3

Técnicas básicas de programación

Introducción

El objetivo de este capítulo es enseñar las técnicas elementales necesarias para escribir programas con el Z80. Presentaremos aquí conceptos nuevos, como son los de organización de registros, bucles y subrutinas. Las técnicas de programación se centrarán exclusivamente en los recursos *internos* del Z80, es decir, en los registros. Desarrollaremos, además, programas reales, en particular programas aritméticos que servirán para ilustrar los conceptos expuestos y para utilizar instrucciones reales. Veremos así qué instrucciones hay que emplear para manipular la información entre la memoria y el μP, y dentro de esta última. En el capítulo siguiente analizaremos con todo detalle las instrucciones de que dispone el Z80. El capítulo 5 irá dedicado al estudio de las técnicas de direccionamiento, y el 6, al manejo de la información *fuera* del Z80 o, lo que es lo mismo, a las técnicas de entrada y salida.

En este capítulo aprenderemos, sobre todo, con la práctica. Al estudiar programas de complejidad creciente aprenderemos la función de las diferentes instrucciones y de los registros, y aplicaremos los conceptos ya desarrollados. No obstante, hay uno muy importante —el de técnicas de direccionamiento—

que, por su aparente complejidad, no abordaremos hasta el capítulo 5.

Tras esta breve introducción, pasaremos sin más a escribir algunos programas, empezando por los aritméticos. La figura 3.1 recoge el "modelo para el programador" de los registros del Z80.

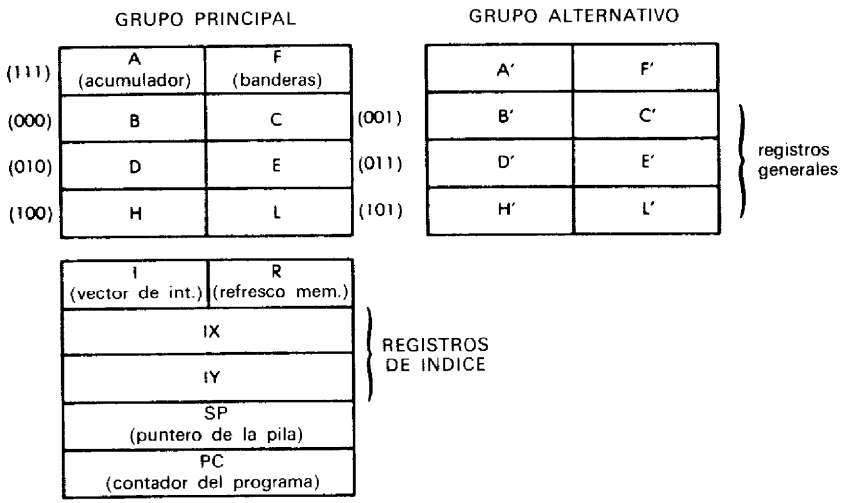

Figura 3.1
Registros del Z80.

Programas aritméticos

Son programas aritméticos los de suma, resta, multiplicación y división. Los presentados aquí funcionarán con enteros, bien positivos binarios, bien en notación de complemento a dos, correspondiendo en este último caso el bit de la izquierda al signo (véase en el capítulo 1 la descripción de la notación en complemento a dos).

SUMA DE 8 BITS

Vamos a sumar dos operandos de 8 bits denominados OP1 y OP2, almacenados, respectivamente, en las direcciones de memoria ADR1 y ADR2. Llamaremos a la suma RES, y la almacenaremos en la dirección ADR3. Todo esto queda recogido en la figura 3.2. El programa encargado de realizar la suma es el siguiente:

Instrucciones	*Comentarios*
LD A, (ADR1)	CARGAR OP1 EN A
LD HL, ADR2	CARGAR LA DIRECCION DE OP2 EN HL
ADD A, (HL)	SUMAR OP2 A OP1
LD (ADR3), A	DEJAR EL RESULTADO RES EN ADR3

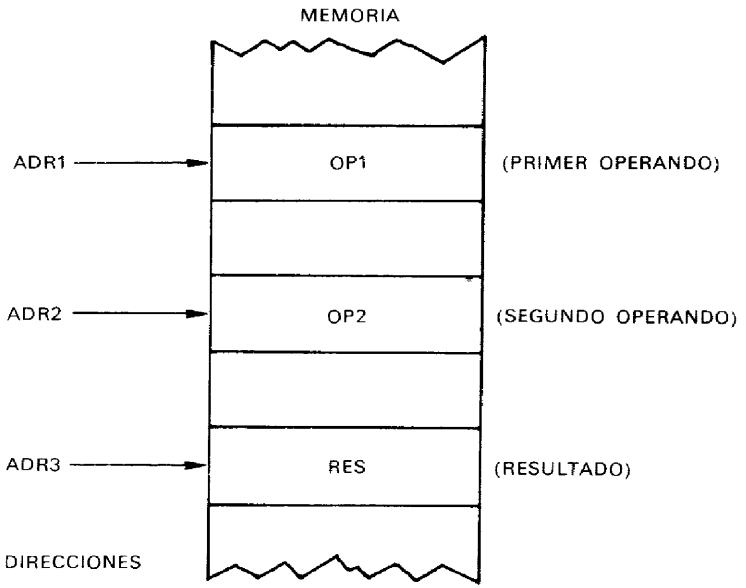

Figura 3.2
Suma de 8 bits; RES = OP1 + OP2.

Este es nuestro primer programa. Las instrucciones están a la izquierda, y los comentarios a las mismas, a la derecha. Examinémoslo con más detenimiento. Se trata de un programa de cuatro instrucciones (se llama *instrucción* a cada una de las líneas, y se ha expresado aquí de forma simbólica). El programa *ensamblador* traducirá cada una de esas instrucciones a uno, dos, tres o cuatro bytes, aunque este extremo no es de nuestra incumbencia.

En la primera línea se especifica que el contenido de ADR1 se cargue en el acumulador A. Como se ve en la figura 3.2, ese contenido es el primer operando "OP1"; por tanto, el resultado de la primera instrucción es la transferencia de OP1 desde la memoria al acumulador; el movimiento se ilustra en la figura 3.3. "ADR1" es una representación simbólica de la dirección real de 16 bits que se encuentra en la memoria. El símbolo ADR1 se definirá en otra parte el programa; podría, por ejemplo, definirse como igual a la dirección "100".

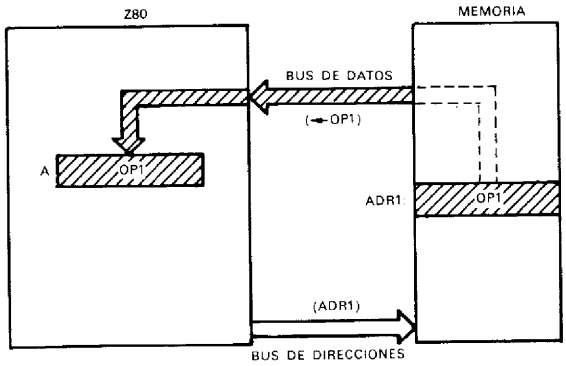

Figura 3.3
LD A, (ADR1): se carga OP1 de la memoria.

Esta instrucción de *carga* da lugar a una *operación de lectura* de la dirección 100 (véase la figura 3.3), cuyo contenido recorrerá el *bus* de datos hasta el acumulador, en el que queda depositado. Recordaremos del capítulo anterior que las operaciones lógicas y aritméticas actúan sobre el acumulador como uno de los operandos fuente (diríjase al mencionado capítulo para más detalles). Como queremos sumar los dos valores OP1 y OP2, debemos empezar por cargar OP1 en el acumulador; hecho esto, podremos sumar los contenidos del mismo, es decir, OP1 y OP2. El campo de la derecha de la instrucción se llama campo de *comentario*. El programa ensamblador lo ignora durante la traducción, pero de todas formas se incluye en el programa por razones de legibilidad. Para entender lo que hace el programa, es de capital importancia usar buenos comentarios (es lo que se llama *documentar* un programa).

En este caso el comentario se explica a sí mismo; el valor de OP1, que se encuentra en la dirección ADR1, se carga en el acumulador A.

El resultado de esta primera instrucción se ilustra en la figura 3.3. La segunda instrucción del programa es:

```
LD HL, ADR2
```

Obliga a "cargar ADR2 en los registros H y L". Para leer en la memoria el segundo operando OP2, antes debemos colocar su dirección en un par de registros del Z80, como H y L. A continuación podemos sumar el contenidio de la posición de memoria cuya dirección está en H y L al acumulador.

```
ADD A, (HL)
```

Como se ve en la figura 3.2, el contenido de la posición de memoria ADR2 es OP2, nuestro segundo operando. El conteni-

do del acumulador es en este momento OP1, el primer operando. Como resultado de la ejecución de esta instrucción, se toma OP2 de la memoria y se suma a OP1, como ilustra la figura 3.4.

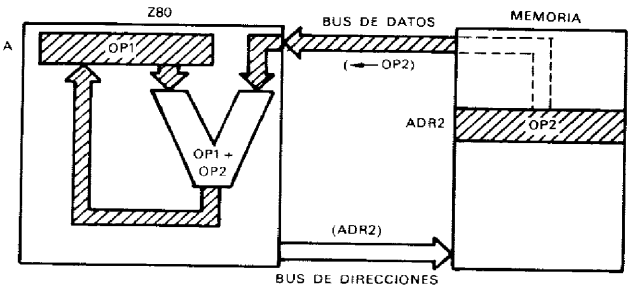

Figura 3.4
ADD A, (HL).

La suma se deposita en el acumulador. El lector recordará que, en el caso del Z80, el resultado de las operaciones aritméticas se deposita en el acumulador. En otros procesadores puede depositarse en otros registros o volver a la memoria.

La suma de OP1 y OP2 está, pues, en el acumulador. Para completar el programa no tenemos más que transferir el contenido de aquél a la posición de memoria ADR3 para almacenar el resultado en la posición especificada. Esta operación es la que realiza la cuarta instrucción del programa:

```
LD (ADR3), A
```

Esta instrucción carga el contenido de A en la dirección ADR3. El efecto de la misma queda ilustrado en la figura 3.5.

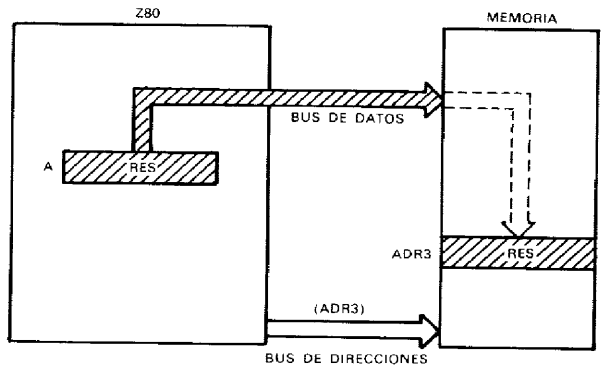

Figura 3.5
LD (ADR3), A (guardar el acumulador en la memoria).

Antes de la ejecución de la operación ADD, el acumulador contiene OP1 (véase la figura 3.4). Tras la suma se ha escrito en el mismo un nuevo resultado, que es "OP1 + OP2". Recuérde-

se que el contenido de cualquier registro interno del microprocesador, al igual que el de cualquier posición de memoria, permanece invariable tras una operación de lectura. En otras palabras, leer el contenido de un registro o de una posición de la memoria no altera ese contenido. Este cambia únicamente en las operaciones de *escritura*. En el caso que nos ocupa, los contenidos de las posiciones de memoria ADR1 y ADR2 permanecen invariables a lo largo de todo el programa. Sin embargo, tras la instrucción ADD, el contenido del acumulador sí se modifica, porque se ha escrito en él la salida de la ALU; en consecuencia, su contenido anterior se pierde.

En lugar de ADR1, ADR2 y ADR3 pueden utilizarse direcciones numéricas reales. Para trabajar con direcciones simbólicas es preciso utilizar lo que se llaman "seudoinstrucciones", que especifican el valor de tales direcciones simbólicas para que el programa ensamblador pueda reemplazarlas por las direcciones físicas verdaderas. Esas seudoinstrucciones podrían ser:

```
ADR1 = 100H
ADR2 = 120H
ADR3 = 200H
```

Ejercicio 3.1: *Consultando exclusivamente la tabla de instrucciones del final del libro, escríbase un programa que sume dos números almacenados en las posiciones de memoria LOC1 y LOC2 y deposite el resultado en la posición LOC3. Compárese el programa con el que acaba de estudiarse.*

SUMA DE 16 BITS

El programa de suma de 8 bits sirve únicamente para sumar números de 8 bits, es decir, números comprendidos entre 0 y 255, si se trabaja en notación binaria absoluta. En la práctica, casi siempre es preciso trabajar con números de 16 bits o más y, por tanto, recurrir a la *precisión múltiple*. Presentaremos aquí algunos ejemplos de operaciones aritméticas en 16 bits que fácilmente podrán extrapolarse a 24, 32 o más (siempre múltiplos de 8). Supondremos que el primer operando está almacenado en las posicions de memoria ADR1 y ADR1-1. Como OP1 es esta vez un número de 16 bits, necesitará dos posiciones de 8 bits. De la misma forma, OP2 se almacena en ADR2 y ADR2-1. El resultado se deposita en las direcciones ADR3 y ADR3-1. La figura 3.6 ilustra todo esto; H denota la mitad superior (bits 8 a 15) y L la inferior (bits 0 a 7).

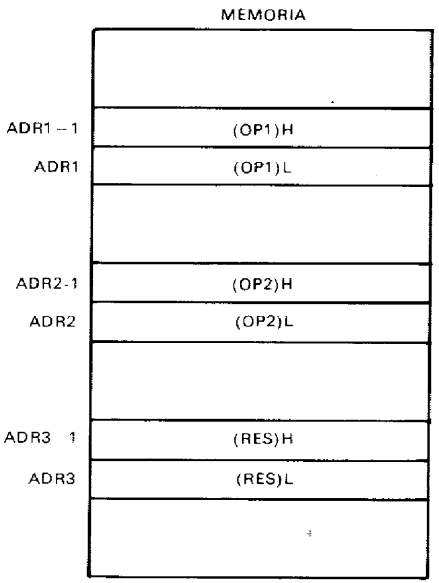

Figura 3.6
Operandos de la suma de 16 bits.

Este programa sigue la misma lógica general que el anterior. En primer lugar, se suman las dos mitades inferiores de los dos operandos, porque el microprocesador sólo puede operar de 8 en 8 bits. El acarreo que pudiera generar esa suma se almacena automáticamente en el bit interno de acarreo ("C"). A continuación se suman las dos mitades de orden superior de los dos operandos y el acarreo, y el resultado se guarda en la memoria. El programa es el siguiente:

```
LD   A, (ADR1)     CARGAR LA MITAD INFERIOR
                   DE OP1
LD   HL, ADR2      CARGAR DIRECCION DE LA MI-
                   TAD INFERIOR DE OP2
ADD  A, (HL)       SUMAR OP1 Y OP2 (INFERIORES)
LD   (ADR3), A     ALMACENAR EL RESULTADO (IN-
                   FERIOR)
LD   A, (ADR1-1)   CARGAR LA MITAD SUPERIOR
                   DE OP1
DEC  HL            DIRECCION DE LA MITAD SUPE-
                   RIOR DE OP2
ADC  A, (HL)       (OP1 + OP2) SUPERIOR + ACA-
                   RREO
LD   (ADR3-1), A   ALMACENAR EL RESULTADO
                   (SUPERIOR)
```

Las primeras cuatro instrucciones del programa son idénticas a las utilizadas para la suma de 8 bits de la sección anterior y dan lugar a la suma de las mitades menos significativas (bits 0 a 7) de OP1 y OP2. La suma, llamada "RES", se deposita en la posición de memoria ADR3 (véase la figura 3.6).

El posible acarreo ("0" ó "1") que pudiera resultar de la suma se almacena automáticamente en el bit de acarreo C del registro de estado (registro F). Si los dos números generan acarreo, el bit C será igual a "1"; en caso contrario, su valor será "0".

Las cuatro instrucciones siguientes son también básicamente iguales a las del programa de 8 bits a que nos estamos refiriendo. En este caso sirven para sumar las mitades más significativas (o mitades superiores, es decir, los bit 8 a 15) de OP1 y OP2 y cualquier acarreo, y almacenar el resultado en ADR3-1.

Tras la ejecución de este programa de 8 instrucciones, el resultado de 16 bits aparece almacenado en las posiciones de memoria ADR3 y ADR3-1. Se observará, no obstante, que hay una diferencia entre las dos partes del programa; en efecto, la *instrucción "ADD"* utilizada en la primera parte (instrucción tercera) no aparece en la segunda. Dicha instrucción suma dos operandos, con independencia del acarreo. En la segunda parte se ha empleado en su lugar otra, llamada "ADC", que suma dos operandos más cualquier acarreo que pudiera haberse producido y es, pues, necesaria para obtener un resultado correcto. Como la suma previamente ejecutada sobre las mitades inferiores puede dar lugar a un acarreo, es preciso tener éste en cuenta en las instrucciones de la segunda parte.

La cuestión que surge inmediatamente es: ¿qué ocurriría si la suma de las mitades superiores de los operandos también diese lugar a un acarreo? Hay dos posibilidades: la primera es suponer que se trata de un error; este programa está pensado para trabajar con resultados de hasta 16 bits, pero no de 17. La otra es incluir instrucciones adicionales para verificar precisamente la posibilidad de que se produzca un acarreo al final del programa. Se trata de la primera de una larga serie de decisiones que ha de tomar el programador.

Nota: Hasta ahora hemos supuesto que la mitad superior de un operando se almacena "encima" de la inferior, es decir, en la dirección de memoria inmediatamente inferior. Pero las cosas no son necesariamente así, y, de hecho, en el Z80 las direcciones se almacenan al revés: primero, la mitad inferior, y a continuación, la mitad superior en la siguiente posición de memoria. Con el fin de trabajar en una convención común para direcciones y datos, es recomendable que también éstos se almacenen

con la parte inferior sobre la superior. La situación se ilustra en la figura 3.7.

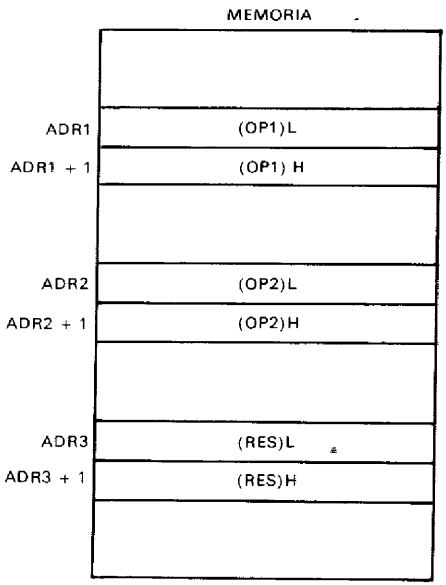

Figura 3.7
Almacenamiento de operandos en orden inverso.

Al trabajar con operandos de varios bytes, es importante tener en cuenta dos convenciones decisivas:

— el orden en que se almacenan los datos en memoria,
— la zona que señalan los apuntadores (byte inferior o byte superior).

Los ejercicios 3.2 y 3.3 están pensados para aclarar estas cuestiones.

Ejercicio 3.2: *Vuelva a escribir el programa de suma de 16 bits con la organización de memoria descrita en la figura 3.7.*

Ejercicio 3.3: *Supóngase ahora que ADR1 no señala hacia la mitad inferior de OP1 (como en las figuras 3.6 ó 3.7), sino hacia la superior (figura 3.8). Vuélvase a escribir el programa teniendo en cuenta esta nueva convención.*

Es el programador quien decide cómo se almacenan los números de 16 bits y también si las referencias de dirección señalan a la mitad inferior de dichos números o a la superior. Es otra decisión que hay que aprender a tomar durante el diseño de algoritmos y estructuras de datos.

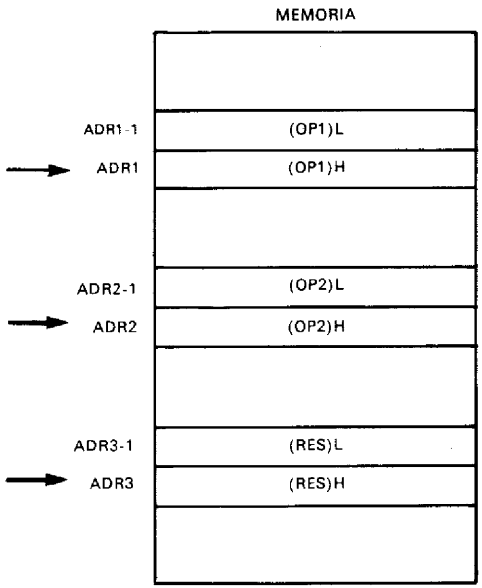

Figura 3.8
Punteros del byte superior.

Los programas que acabamos de examinar son programas tradicionales, que utilizan el acumulador. Veremos a continuación una alternativa al de 16 bits, que trabaja no con el acumulador, sino con algunas de las instrucciones especiales de 16 bits de que dispone el Z80. Supondremos que los operandos están almacenados tal como describe la figura 3.6. El programa es el siguiente:

```
LD   HL, (ADR1) CARGAR HL CON OP1
LD   BC, (ADR2) CARGAR BC CON OP2
ADD  HL, BC     SUMAR 16 BITS
LD   (ADR3), HL ALMACENAR RES EN ADR3
```

Lo primero que llama la atención en este programa es que es mucho más corto que el anterior. Se dice que es más "elegante". *Con ciertas limitaciones, los registros H y L del Z80 pueden utilizarse como un acumulador de 16 bits.*

Ejercicio 3.4: *Con las instrucciones de 16 bits que acabamos de presentar, escríbase un programa de suma para operandos de 32 bits, suponiendo que éstos se almacenan como describe la figura 3.9.*

Respuesta:

```
LD   HL, (ADR1)
LD   BC, (ADR2)
ADD  HL, BC
```

```
LD   (ADR3), HL
LD   HL, (ADR1 + 2)
LD   BC, (ADR2 + 2)
ADC  HL, BC
LD   (ADR3 + 2), HL
```

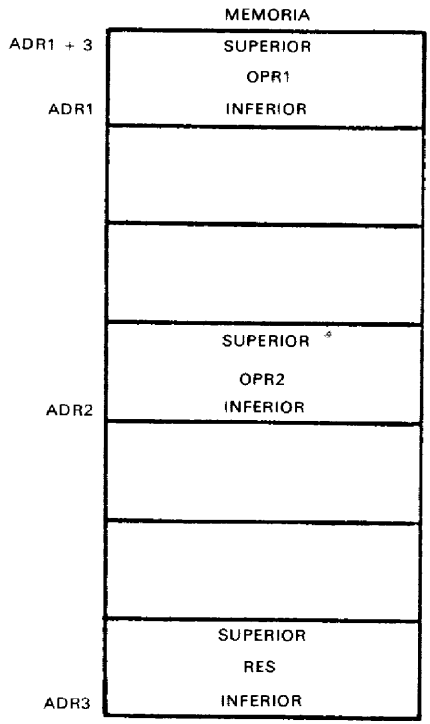

Figura 3.9
Suma de 32 bits.

Ahora que ya sabemos programar la suma binaria, podemos pasar a la resta.

RESTA DE NUMEROS DE 16 BITS

La resta de 8 bits es demasiado sencilla, así que la dejaremos como ejercicio y pasaremos directamente al problema de restar números de 16 bits. Como es habitual, los dos números OP1 y OP2 se almacenan en las direcciones ADR1 y ADR2, suponiendo una disposición de memoria similar a la ilustrada en la figura 3.7. Para restar se sustituye la instrucción ADD por la SBC.

Ejercicio 3.5: *Escribir un programa de resta.*

El programa aparece a continuación, y las rutas seguidas por los datos se muestran en la figura 3.10.

```
LD   HL, (ADR1)   OP1 EN HL
LD   DE, (ADR2)   OP2 EN DE
AND A             ELIMINAR ACARREO
SBC HL, DE        OP1 – OP2
LD   (ADR3), HL   RES EN ADR3
```

El programa es básicamente igual al de suma de 16 bits. Sin embargo, mientras que el Z80 tiene dos tipos de suma en registros dobles —ADD y ADC—, sólo cuenta con una resta: SBC. Como consecuencia de ello, ha habido que introducir dos cambios.

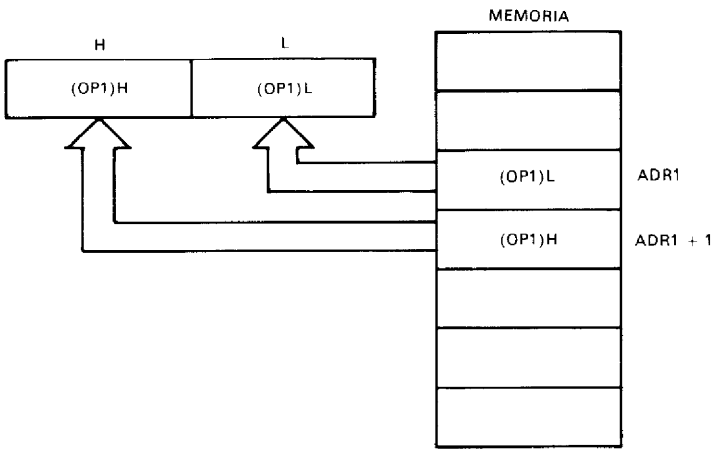

Figura 3.10
Carga de 16 bits: LD HL, (ADR1).

El primero es el uso de SBC en lugar de ADD.

El segundo es la instrucción "AND A", utilizada para eliminar la bandera de acarreo antes de restar. Esta instrucción no modifica el valor de A.

La precaución es necesaria, porque el Z80 dispone de dos formas de suma, con y sin acarreo, en los registros H y L, pero sólo de una resta, SBC, o resta con acarreo, cuando trabaja en el par de registro HL. Como SBC tiene en cuenta automáticamente el valor del bit de acarreo, es preciso poner éste a 0 antes de ejecutar la operación, y eso es precisamente lo que hace la instrucción "AND A".

Ejercicio 3.6: *Escríbase de nuevo el programa de resta sin usar las instrucciones especiales de 16 bits.*

Ejercicio 3.7: *Escríbase el programa de resta para operandos de 8 bits.*

Hay que recordar que, en la aritmética en complemento a dos, el valor final de la bandera de acarreo no tiene significado. Si, como resultado de la resta, se produce una situación de desbordamiento, entrará en juego el bit correspondiente (bit V) del registro de estado, cuyo valor podrá verificarse.

Los ejemplos que acabamos de estudiar corresponden a sencillas sumas y restas binarias, pero es necesario trabajar en otras formas de representación aritmética, y en particular en BDC.

Aritmética BCD

SUMA BCD DE 8 BITS

El concepto de notación aritmética BCD se expuso en el capítulo 1, pero recordaremos aquí sus características esenciales. Se utiliza, sobre todo, en aplicacions contables, en las que es de rigor conservar en el resultado todas las cifras significativas. En notación BCD se emplea un *nibble* de 4 bits para almacenar una cifra decimal (de 0 a 9), de manera que un byte de 8 bits representa dos cifras BCD *(BCD condensado)*. Veamos ahora cómo se suman dos bytes de dos cifras BCD cada uno.

Para identificar la naturaleza del problema, analizaremos antes algunos ejemplos numéricos.

Sea la suma de "01" y "02":

"01" se representa como	0000 0001
"02" se representa como	0000 0010
El resultado es	0000 0011

que es la representación BCD de "03" (si duda al deducir los equivalentes BCD, consulte la tabla de conversión del final del libro). Este caso ha sido muy fácil. Veamos otro:

"08" se representa como	0000 1000
"03" se representa como	0000 0011

Ejercicio 3.8: *Calcúlese la suma de los dos números de arriba en notación BCD. ¿Qué se obtiene como resultado?*

Respuesta: Si el resultado es "0000 1011", lo que ha obtenido es la suma *binaria* de 8 y 3, es decir, 11 en representación binaria. Lo que ocurre es que "1011" *no es un código BCD*

válido. De lo que se trata es de obtener la representación *BCD* de "11", es decir, 00010001.

El problema radica en que la representación BCD utiliza únicamente las primeras diez combinaciones de cuatro cifras para codificar los símbolos decimales 0 a 9. Las seis posibles combinaciones que quedan no se usan, y "1011" es una de ellas. En otras palabras, siempre que la suma de dos cifras BCD sea mayor que 9, hay que añadir 6 al resultado para saltar por encima los seis códigos que no se usan.

En efecto, al sumar la representación binaria de "6" a 1001:

```
          1011    (resultado binario no permitido)
        + 0110    (+ 6)
          ----
se obtiene: 00010001
```

que es "11" en notación BCD. Por fin hemos obtenido el resultado correcto.

Este ejemplo ilustra una de las dificultades básicas que plantea la notación BCD, a saber: la necesidad de compensar la existencia de seis códigos que no se usan. Para corregir el resultado de la suma binaria se utiliza una instrucción "DAA", llamada "ajuste decimal" (la instrucción suma 6 si el resultado es mayor que 9).

El mismo ejemplo servirá para ilustrar el problema siguiente. El acarreo procede de la cifra BCD inferior (la de la derecha) y pasa a la de la izquierda. Es preciso tener en cuenta este acarreo interno y sumarlo a la segunda cifra BCD, cosa que hace automáticamente la instrucción de suma. Sin embargo, con frecuencia conviene detectar este acarreo interno del bit 3 al 4 ("semiacarreo"), para lo que se emplea la bandera H.

Ilustraremos todo esto con el siguiente ejemplo, un programa que suma los números BCD "11" y "22":

```
LD   A, 11H      CARGAR EL LITERAL BCD "11"
ADD  A, 22H      SUMAR EL LITERAL BCD "22"
DAA              AJUSTE DECIMAL DEL RESULTA-
                 DO
LD   (ADR), A    ALMACENAR EL RESULTADO
```

En este programa hemos introducido un símbolo nuevo —"H"— que, situado dentro del campo del operando de la instrucción, indica que el dato al que sigue se expresa en notación hexadecimal. Las representaciones hexadecimal y BCD de las cifras "0" a "9" son idénticas. En este caso queremos sumar los literales (o constantes) "11" y "22", y almacenar el

resultado en la dirección ADR. Cuando el operando se especifica como parte de una instrucción, como el ejemplo que nos ocupa, se habla de *direccionamiento inmediato* (en el capítulo 5 se analizarán en detalle las diversas formas de direccionamiento). Almacenar el resultado en una dirección especificada, como LD (ADR), A se llama *direccionamiento absoluto* cuando ADR representa una dirección de 16 bits.

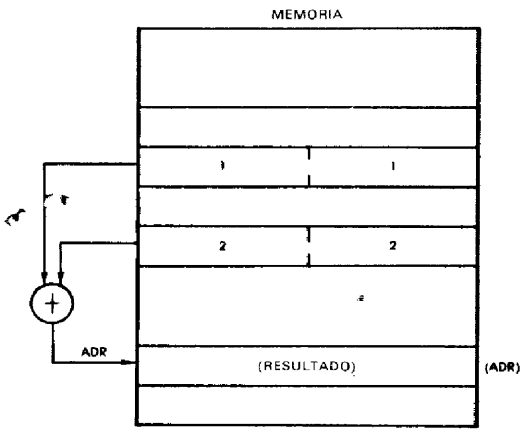

Figura 3.11
Almacenamiento de cifras BCD.

Este programa es análogo al de suma binaria de 8 bits, pero con la nueva instrucción "DAA", de la que mostraremos el funcionamiento con un ejemplo. Empecemos por sumar "11" y "22" en BCD:

$$\begin{array}{rll} & 00010001 & (11) \\ + & 00100010 & (22) \\ \hline = & \underbrace{00110011}_{3\quad 3} & (33) \end{array}$$

El resultado, alcanzado con las reglas de la suma binaria, es correcto.

Sumemos ahora "22" y "39" con las mismas reglas de la suma *binaria:*

$$\begin{array}{rll} & 00100010 & (22) \\ + & 00111001 & (39) \\ \hline = & \underbrace{01011011}_{5\quad ?} & \end{array}$$

"1011" es un *código BCD no permitido*, porque en BCD se utilizan sólo los primeros diez códigos binarios y se saltan los

seis siguientes; por tanto, habrá que hacer aquí lo mismo, es decir, sumar 6 al resultado:

```
  01011011   (resultado binario)
+     0110   (6)
= 01100001   (61)
    6   1
```

que es el resultado BCD correcto.

Ejercicio 3.9: *¿Podría colocarse en el programa la instrucción DAA después de la LD (ADR), A?*

RESTA BCD

A primera vista, la resta en BCD parece una cosa muy complicada. La operación se realiza sumando el *complemento a diez* del número, igual que se sumaba el complemento a dos para realizar la resta binaria. El complemento a 10 se calcula determinando el de 9 y sumando 1, lo que en un microprocesador normal consume de tres a cuatro operaciones; sin embargo, el Z80 dispone de una potente instrucción DAA, que simplfica las cosas.

La instrucción DAA ajusta automáticamente el valor del resultado del acumulador en función del valor de las banderas C, H y N, antes de DAA, a su valor correcto (véase el capítulo siguiente para más detalles sobre DAA).

SUMA BCD DE 16 BITS

Esta operación se realiza exactamente igual que la binaria correspondiente. El programa es el siguiente:

```
LD    A,(ADR1)      CARGAR (OP1) EN A
LD    HL,(ADR2)     CARGAR ADR2 EN HL
ADD   A,(HL)        (OP1 + OP2) INFERIOR
DAA                 AJUSTE DECIMAL
LD    (ADR3), A     ALMACENAR EL RESULTADO (IN-
                    FERIOR)
LD    A, (ADR1 + 1) CARGAR (OP1) H EN A
INC   HL            PUNTERO A ADR2 + 1
ADC   A,(HL)        (OP1 + OP2)  SUPERIOR + ACA-
                    RREO
```

```
DAA                        AJUSTE DECIMAL
LD      (ADR3 + 1), A      ALMACENAR EL RESULTADO
                           (SUPERIOR)
```

RESTA EN BCD EMPAQUETADO

Ya se han descrito los procedimientos elementales de suma y resta BCD. Sin embargo, en la práctica habitual, los números BCD tienen un número variable de bytes. Como ejemplo simplificado de resta en BCD empaquetado supondremos que las dos cantidades localizadas en N1 y N2 tienen el mismo número de bytes BCD, número que se llama CUENTA. La organización de registros y memoria se muestra en la figura 3.12. El programa es el siguiente:

```
BCDPAK  LD      B, CUENTA
        LD      DE, N2
        LD      HL, N1
        AND     A           ELIMINAR ACARREO
MENOS   LD      A, (DE)     BYTE N2
        SBC     A, (HL)     N2 – N1
        DAA
        LD      (HL), A     ALMACENAR EL RE-
                            SULTADO
        INC     DE
        INC     HL
        DJNZ    MENOS       DECREMENTAR B Y
                            REPETIR HASTA B = 0
```

N1 y N2 son las direcciones en que están almacenados los números BCD, direcciones que se cargan en los pares de registros DE y HL:

```
BCDPAK  LD      B, CUENTA
        LD      DE, N2
        LD      HL, N1
```

A continuación, antes de la primera resta, hay que eliminar el bit de acarreo. Ya se ha dicho que hay varias formas de hacerlo. Una de ellas es:

```
AND A
```

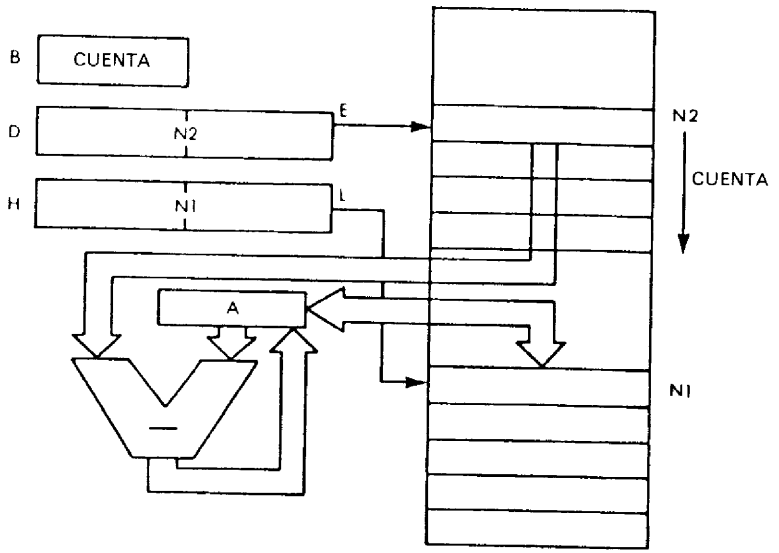

Figura 3.12
Resta en BCD empaquetado N1 ← N2 − N1.

El primer byte de N2 se carga en el acumulador y se le resta el primero de N1. A continuación se utiliza la instrucción DAA para obtener el valor BCD correcto:

```
MENOS   LD      A, (DE)
        SBC     A, (HL)
        DAA
```

El resultado se almacena en N1:

```
LD  (HL), A
```

Por último, se incrementan los punteros de los bytes en curso:

```
INC  DE
INC  HL
```

Se decrementa el contador y se ejecuta el bucle de resta hasta que alcance el valor "0":

```
DJNZ  MENOS
```

La instrucción DJNZ es una instrucción especial del Z80 que decrementa el registro B y salta —si no es 0— en una sola operación.

Ejercicio 3.10: *Compárese el programa que acaba de presentarse con el de suma binaria de 16 bits. ¿Dónde está la diferencia?*

Ejercicio 3.11: *¿Son intercambiables los papeles de DE y HL? (Un consejo: cuidado con SBC.)*

Ejercicio 3.12: *Escríbase un programa de resta en BCD de 16 bits.*

BANDERAS BCD

En notación BCD, la bandera de acarreo que aparece como resultado de una suma indica que el resultado es superior a 99. La situación es diferente a la que se daba en complemento a dos, porque las cifras BCD están representadas en auténtica notación binaria; por el contrario, la presencia de bandera de acarreo tras una resta indica un defecto.

TIPOS DE INSTRUCCIONES

Hemos utilizado ya dos tipos de instrucciones del microprocesador: instrucciones LD, que cargan el acumulador a partir de direcciones de memoria o almacenan su contenido en direcciones especificadas; se llaman instrucciones de *transferencia de datos.*

Instrucciones *aritméticas*, como ADD, SUB, ADC y SBC, que ejecutan operaciones de suma y resta. Pronto veremos en este mismo capítulo otras instrucciones de la ALU.

Pero hay otros tipos que todavía no hemos usado: se trata de las instrucciones de *salto*, que modifican el orden de ejecución del programa; recurriremos a esta clase de instrucciones en el próximo ejemplo. Conviene observar que a las instrucciones de *salto* se les llama también de *bifurcación* en situaciones condicionales, es decir, en puntos del programa en los que se toma una decisión lógica. Como sugiere el nombre de *bifurcación*, la presencia de esa instrucción escinde el camino único del programa en dos divergentes.

ultiplicación

Analicemos a continuación un problema aritmético más complicado: la multiplicación de números binarios. Antes de redactar el algoritmo de la operación, examinaremos una multiplicación decimal corriente; sea la de 12 por 23:

```
    12   (multiplicando = MPD)
×   23   (multiplicador = MPR)
-------
    36   (producto parcial)
+ 24
-------
= 276    (resultado final)
```

La operación se lleva a cabo multiplicando la cifra de la derecha del multiplicador por el multiplicando, es decir: "3" × "12"; el producto parcial es "36"; a continuación se multiplica la siguiente cifra del multiplicador, "2", por "12", y el resultado, "24", se suma al producto parcial.

Pero todavía falta una operación: 24 se escribe *desplazado una posición hacia la izquierda* (decimos que 24 se *desplaza a la izquierda* una posición, pero igual podríamos haber dicho que el producto parcial 36 se *desplaza una posición a la derecha*).

Los dos números, correctamente desplazados, se suman, y así se obtiene el producto 276. Es algo muy sencillo, y las cosas ocurren exactamente de la misma forma en notación binaria.

Veamos, por ejemplo, la multiplicación de 5 por 3:

```
(5)     101    (multiplicando
(3)   × 011    (multiplicador)
      -----
      101      (producto parcial)
     101
    000
    -----
(15)  01111    (resultado final)
```

Realizaremos la multiplicación exactamente igual que en el ejemplo que acabamos de ver. La representación formal del algoritmo aparece en la figura 3.13. Se trata de un diagrama de flujo —el primero del libro—, y lo analizaremos con cierto detalle.

El diagrama de flujo es una representación simbólica del algoritmo que acabamos de seguir. Cada rectángulo representa una orden que debe ejecutarse, y que habrá que transformar en una o más instrucciones del programa. En cada uno de los símbolos romboidales hay que llevar a cabo una comprobación, ya que son *puntos de bifurcación* del programa. Si el resultado de la comprobación es afirmativo, la bifurcación conduce el flujo del programa a una posición determinada; en caso negativo, lo conduce a una posición diferente. La idea de bifurcación se expondrá más adelante, en el propio programa. El lector deberá, por el momento, estudiar atentamente el diagrama de flujo hasta que adquiera la completa seguridad de que represen-

ta efectivamente el algoritmo de multiplicación. Obsérvese que del rombo inferior parte una flecha que se dirige al superior; se debe a que esa porción del diagrama debe ejecutarse ocho veces, una por cada bit del multiplicador. Esta disposición del programa que provoca el reinicio de una misma operación en un mismo punto es lo que se llama un *bucle*.

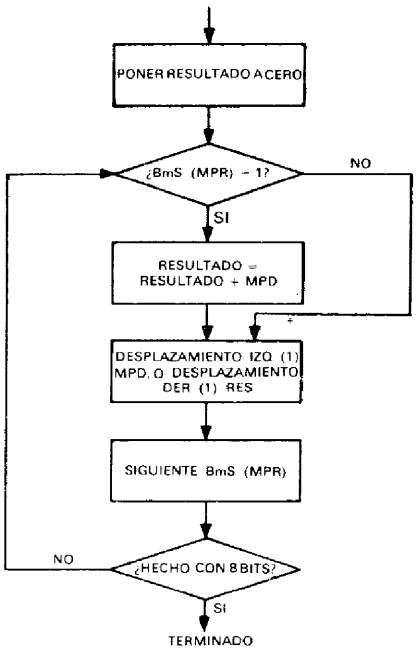

Figura 3.13
Diagrama de flujo del algoritmo básico de multiplicación.

Ejercicio 3.13: *Ejecútese la multiplicación binaria de "4" por "7" utilizando el diagrama de flujo y verificando el resultado, que debe ser "28". Pruébese de nuevo si el que se obtiene es otro, porque sólo quien sea capaz de seguir el diagrama hasta dar con el valor correcto estará en condiciones de transformarlo en un programa.*

MULTIPLICACION 8 POR 8

Vamos, por fin, a transformar el diagrama de flujo en un programa para el Z80, programa que aparece completo en la figura 3.14 y que estudiaremos en detalle. Como se recordará del capítulo 1, programar consiste en este caso en "traducir" el diagrama de flujo de la figura 3.13 al programa de la 3.14. Cada

uno de los recuadros del diagrama dará lugar a una o más instrucciones.

Se supone que MPR y MPD han recibido ya un valor concreto.

```
MPY88   LD    BC,(MPRAD)    CARGAR MULTIPLICADOR EN C
        LD    B,8           B ES EL CONTADOR DE BIT
        LD    DE,(MPDAD)    CARGAR EL MULTIPLICANDO EN E
        LD    D,0           BORRAR D
        LD    HL,0          PONER EL RESULTADO A 0
MULT    SRL   C             DESPLAZAR EL BIT DEL MULTIPLICADOR AL ACARREO
        JR    NC,NOADD      VERIFICAR EL ACARREO
        ADD   HL,DE         SUMAR MPD AL RESULTADO
NOADD   SLA   E             DESPLAZAR MPD A LA IZQUIERDA
        RL    D             LLEVAR BIT A D
        DEC   B             DECREMENTAR CONTADOR DE DESPLAZAMIENTO
        JP    NZ,MULT       REPETIR SI CONTADOR ≠ 0
        LD    (RESAD),HL    ALMACENAR EL RESULTADO
```

Figura 3.14
Programa de multiplicación 8 × 8.

La primera casilla del diagrama es la de *inicialización*, necesaria porque antes de nada hay que poner a 0 una serie de registros o posiciones de memoria para que el programa trabaje en ellos. Los registros que se utilizarán en el programa de multiplicación aparecen en la figura 3.15.

Se utilizan tres pares de registros del Z80. El multiplicador de 8 bits se supone que reside en la posición de memoria, MPRAD; el multiplicando, MPD, ocupa la dirección MPDAD, y los dos se cargarán en los registros C y E, respectivamente (véase la figura 3.15). El registro B trabaja como contador.

Los registros D y E albergan el multiplicando a medida que se desplaza de bit en bit hacia la izquierda.

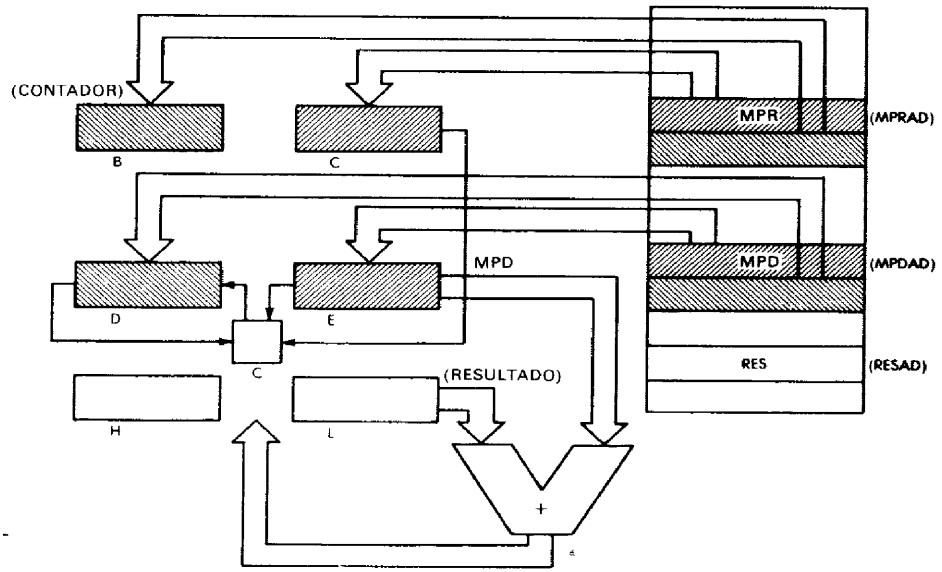

Figura 3.15
Registros usados en la multiplicación 8 × 8.

Obsérvese que, aunque en un principio basta con cargar C y E, hay que prever 16 bits para que también puedan cargarse B y D a partir de la memoria; se ponen, respectivamente, a "8" y "0".

Por último, hay que tener en cuenta que el resultado de una multiplicación de 8 por 8 bits puede ocupar hasta 16 bits, porque $2^8 \times 2^8 = 2^{16}$; por tanto, hay que reservar para el resultado dos registros, que son los H y L, como indica la figura 3.15.

El primer paso es cargar los registros B, C y E con los contenidos adecuados e iniciar el resultado (el producto parcial) al valor "0", tal como especifica el diagrama de flujo de la figura 3.13. Todo ello se consigue con las siguientes instrucciones:

```
MPY88   LD   BC,(MPRAD)
        LD   B,8
        LD   DE,(MPDAD)
        LD   D,0
        LD   HL,0
```

Las tres primeras instrucciones cargan MPR en el par de registros BC, el valor "8" en el registro B y MPD en el par de registros DE, respectivamente. Como MPR y MPD son palabras de 8 bits, se cargan, de hecho, en los registros C y E, mientras que las palabras de la memoria que les siguen pasan a B y D. La situación se ilustra en las figuras 3.16 y 3.17. La siguiente instrucción pone a 0 el contenido de D.

En este programa de multiplicación, el multiplicando se desplaza hacia la izquierda antes de sumarlo al resultado (recuérdese que también se puede desplazar el resultado a la derecha, como indica la cuarta casilla del diagrama de flujo de la figura 3.13). A cada paso, el multiplicando MPD se desplaza hacia el registro D, que, por tanto, debe iniciarse al valor "0", operación que lleva a cabo la instrucción cuarta. La quinta pone a 0 de una vez los contenidos de los registros H y L.

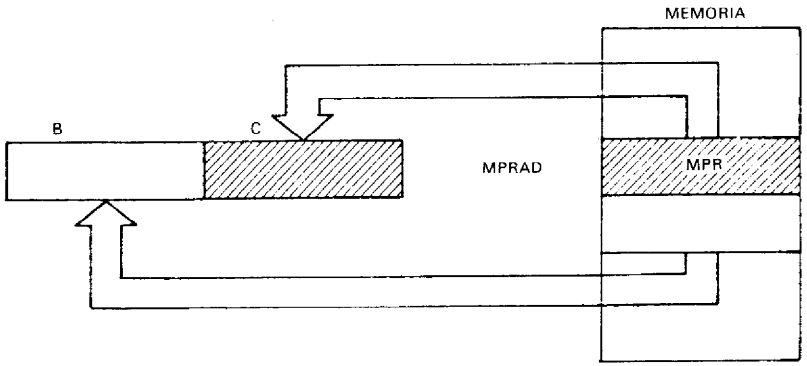

Figura 3.16
LD BC, (MPRAD).

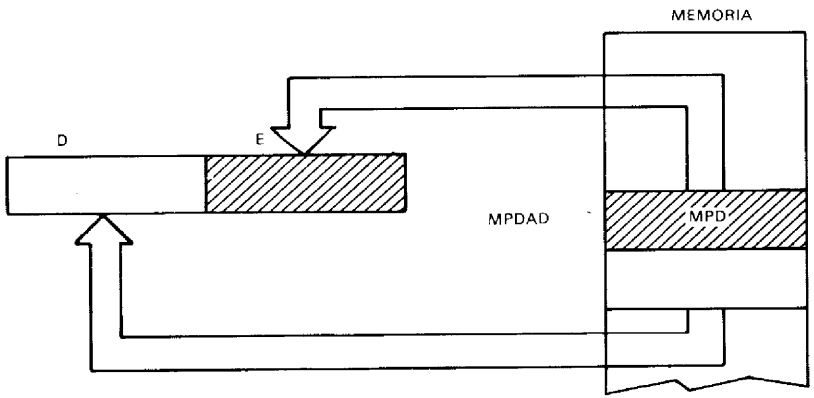

Figura 3.17
LD DE, (MPDAD).

El siguiente paso del diagrama de flujo consiste en poner a prueba el bit menos significativo (el de la derecha) del multiplicador, MPR. Si resulta ser "1", el valor de MPD se añade al resultado parcial; en caso contrario, no se añade. Para ello hacen falta tres instrucciones:

```
MULT   SRL    C
       JR     NC, NOADD
       ADD    HL, DE
```

El primer problema a resolver es la verificación del bit menos significativo del multiplicador contenido en el registro C. Podemos usar para ello la instrucción BIT del Z80, que permite comprobar cualquier bit de cualquier registro, pero en este caso lo que nos interesa es crear un programa con un bucle lo más sencillo posible. Para utilizar la instrucción BIT, tendríamos que comprobar, primero, el bit 0; luego, el 1, y así sucesivamente hasta llegar al 7, lo que exigiría una instrucción diferente cada vez, algo incompatible con un solo bucle. Para acortar el programa hay que buscar otro camino, y en este caso hemos decidido trabajar con una instrucción de *desplazamiento*.

Nota: Hay una forma de utilizar la instrucción BIT y un bucle, pero exigiría que el programa se modificase a sí mismo, una práctica que, por el momento, evitaremos.

SRL es un nuevo tipo de operación que se ejecuta dentro de la unidad aritmética y lógica. Significa *desplazamiento lógico a la derecha*. Tras un desplazamiento lógico a la derecha, aparece un "0" en la posición del bit 7; por el contrario, tras un *desplazamiento aritmético a la derecha*, el bit que ocupa la posición 7 adopta el mismo valor que antes tenía dicha posición. En el próximo capítulo se describirán las diversas operaciones de desplazamiento. El resultado de la instrucción SRL C viene mostrado en la figura 3.15 por una flecha que sale del registro C y se dirige hacia el cuadrado que designa el bit de acarreo ("C"). En este punto, el bit de la derecha del MPR estará en el bit de acarreo C, el que se comprueba.

La instrucción siguiente, "JR NC, NOADD", es una operación de *salto* que significa: "si no hay acarreo" (NC), saltar a la dirección NOADD (etiqueta). Si el contenido del bit de acarreo es "0" (no hay acarreo), el programa salta a la dirección NOADD; si el contenido de C es "1" (hay acarreo), no se produce bifurcación, y se ejecuta la siguiente instrucción de la secuencia, en este caso "ADD HL, DE".

Esta instrucción dice que hay que sumar los contenidos de D y E a los de H y L, y dejar el resultado en H y L. Como E contiene el multiplicando, MPD (véase la figura 3.15), resulta que la instrucción suma dicho multiplicando al resultado parcial.

En este punto, con independencia de que MPD se haya sumado o no al resultado, hay que desplazar el multiplicando a la izquierda (cuarto recuadro del diagrama de flujo de la figura 3.13). Para ello se usa la instrucción:

```
NOADD  SLA E
```

SLA significa "desplazamiento aritmético a la izquierda". Como ya hemos explicado, hay dos tipos de operaciones de desplaza-

miento: lógico y aritmético. Este es el aritmético. En el caso de desplazamiento a la izquierda, SLA especifica que el bit de la parte derecha del registro (el menos significativo) sea "0", como en el caso de SRL que vimos antes.

Supongamos, por ejemplo, que el contenido inicial del registro E fuera 00001001. Tras la instrucción SLA, ese contenido será 00010010, y el bit de acarreo valdrá 0.

Pero, como se ve en la figura 3.15, lo que nos interesa es desplazar el bit más significativo (BMS) de E directamente a D (este movimiento viene ilustrado por la flecha que va de E a D); sin embargo, no hay ninguna instrucción para desplazar un doble registro, como el D y E, de una vez. Una vez desplazados los contenidos de D y E, el bit de la izquierda habrá "caído" en el bit de acarreo; por tanto, hay que recoger ese bit y desplazarlo al registro D, y para ello sirve la instrucción siguiente:

```
RL D
```

RL es también una operación de desplazamiento, pero de distinto tipo. Significa "rotación circular a la izquierda". En una operación de *rotación circular*, al contrario que en una de *desplazamiento*, el bit que llega al registro es el contenido del bit de acarreo C (véase la figura 3.18), que es justamente lo que nos interesa. El contenido de C se carga en el extremo derecho de D, lo que, de hecho, equivale a transferir el bit izquierdo de E.

Esta secuencia de dos instrucciones se muestra en la figura 3.19. Como se ve, el bit identificado por X en la posición más significativa de E pasa primero al bit de acarreo y a continuación a la posición menos significativa de D.

En este punto, como indica el diagrama de flujo de la figura 3.13, hay que señalar el siguiente bit de MPR y comprobar si es el octavo. Esto se consigue decrementando el contador de bits del registro B (figura 3.15). De decrementar el registro, se encarga la instrucción:

```
DEC B
```

que es una instrucción de decremento de resultado evidente.

Por último, es preciso comprobar si el contador ha disminuido hasta 0, lo que se consigue examinado el valor del bit Z. Como recordará el lector, la bandera Z (0) indica si la operación aritmética previa (DEC, por ejemplo) ha producido un resultado nulo. Obsérvese, sin embargo, que DEC HL, DEC BC, DEC DE, DEC IX y DEC SP no afectan a la bandera Z mencionada. Si el contador no es "0", quiere decir que la operación no ha terminado, y que hay que ejecutar una vez más

el bucle del programa, de lo que se encarga la instrucción siguiente:

JP NZ, MULT

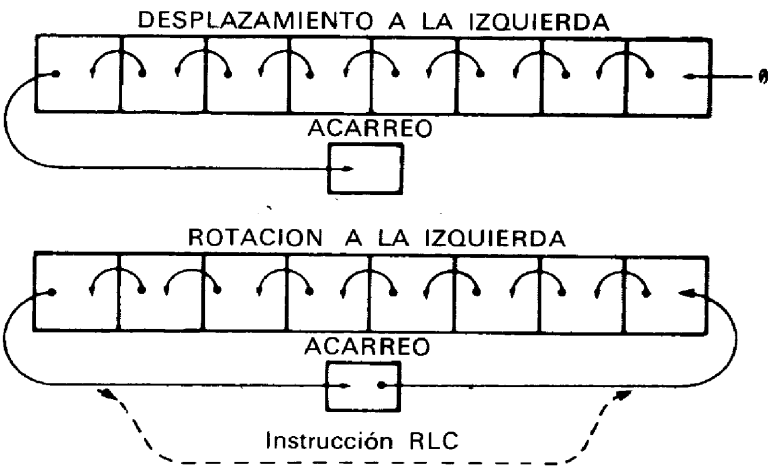

Figura 3.18
Desplazamiento y rotación circular.

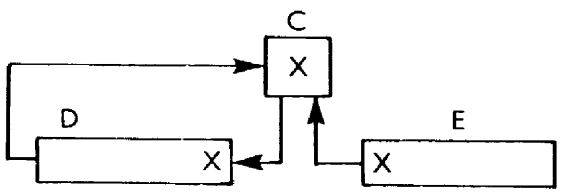

Figura 3.19
Desplazamiento de E a D.

Se trata de una instrucción de salto, la cual especifica que siempre que el bit Z no sea 0 (NZ significa no cero), hay que saltar a la posición MULT. De esta forma se cierra el *bucle del programa*, que se ejecutará una y otra vez hasta que el valor de B se reduzca a 0. En ese momento, el bit Z adquirirá un valor no nulo, y la instrucción JP NZ dejará de actuar, lo que dará lugar a que se ejecute la instrucción siguiente de la secuencia, a saber:

LD (RESAD), HL

Esta instrucción almacena el contenido de H y L, es decir, el resultado de la multiplicación, en la dirección RESAD. Obsérvese que la instrucción transfiere los contenidos de ambos registros a dos posiciones de memoria consecutivas: RESAD y RESAD + 1. Almacena 16 bits de una vez.

Ejercicio 3.14: *¿Sería capaz de escribir el programa de multiplicación que acabamos de estudiar sustituyendo la instrucción SRL C por la BIT (descrita en el capítulo siguiente)? ¿Qué inconveniente tiene?*

Tratemos ahora de mejorar el programa, si tal cosa es posible.

Ejercicio 3.15: *¿Puede sustituirse JR por JP al final del programa? En caso afirmativo, ¿qué ventaja tiene el cambio?*

Ejercicio 3.16: *¿Puede utilizarse DJNZ para acortar el final del programa?*

Ejercicio 3.17: *Estúdiense las instrucciones LD D, 0 y LD HL, 0 del principio del programa; ¿pueden sustituirse por*

```
XOR  A
LD   D, A
LD   H, A
LD   L, A
```

En caso afirmativo, ¿qué efecto ejercerían sobre el tamaño (número de bytes) y la velocidad?

En la mayor parte de los casos, el programa que acabamos de desarrollar será un subrutina que tendrá como instrucción final RET (*return*, vuelta). El mecanismo de subrutina se explicará más adelante en este mismo capítulo.

UN EJERCICIO IMPORTANTE

El que acabamos de ver es el primer programa realmente importante del libro. Incluye instrucciones muy diversas: de transferencia (LD), aritméticas (ADD), lógicas (SRL, SLA, RL) y de salto (JR, JP) y cuenta, además, con un bucle de siete instrucciones que empiezan en la dirección MULT y se ejecutan varias veces seguidas. Para aprender a programar es imprescindible entender perfectamente este programa. Es más largo que los sencillos programas aritméticos de los capítulos anteriores, y resulta imprescindible examinarlo con detenimiento. A continuación se propone un ejercicio de la mayor importancia, y es decisivo que el lector lo realice por completo y correctamente antes de seguir, porque será la única demostración de que ha asimilado los conceptos expuestos con anterioridad. Quien lo resuelva correctamente tendrá la seguridad de haber comprendi-

do cabalmente la forma en que el microprocesador manipula la información, la desplaza entre los registros y la memoria, y la procesa. Quien no haga el ejercicio o quien no lo resuelva correctamente es muy probable que tropiece con dificultades al escribir sus propios programas. Aprender a programar exige práctica; por tanto, tome un papel, o utilice la figura 3.20, y haga el

Ejercicio 3.18: *Siempre que se escribe un programa es preciso comprobarlo a mano, para tener la seguridad de que proporciona resultados correctos, y eso es justamente lo que vamos a hacer ahora; la finalidad de este ejercicio es rellenar la tabla de la figura 3.20.*

ETIQUETA	INSTRUCCION	B	C	C (ACARREO)	D	E	H	L

Figura 3.20
Tabla del ejercicio de multiplicación.

La respuesta puede escribirse directamente en la figura o en un papel aparte. De lo que se trata es de indicar el contenido de cada uno de los registros que intervienen en el programa tras la ejecución de cada una de las instrucciones. En la figura 3.20

aparecen todos los que forman parte del programa de la figura 3.14, que son, de izquierda a derecha: B y C, el acarreo C, D y E, y H y L. En la parte izquierda de la figura se anotan la etiqueta, si existe, y la instrucción que va a ejecutarse. En la parte derecha se anota el contenido de cada uno de los registros tras la ejecución de la instrucción que figura a la izquierda. Si el contenido de un registro no se conoce (es indefinido), se señala tal circunstancia con un trazo. Como ejemplo, empezaremos a rellenar las primeras filas de la tabla:

ETIQUETA	INSTRUCCION	B	C	C	D	E	H	L
		--	--	-	--	--	--	--
MPY88	LD BC,(0200)	00	03	-	--	--	--	--

Figura 3.21
El programa de multiplicación tras una instrucción.

Suponemos en este caso que estamos multiplicando "3" (MPR) por "5" (MPD).

La primera instrucción que se ejecuta es "LD BC, (MPRAD)". El contenido de la posición de memoria MPRAD se carga en los registros B y C. Hemos supuesto que MPR es igual a 3, es decir, "00000011". Tras la ejecución de esta instrucción, el contenido del registro C pasa a ser "3". Obsérvese que la instrucción también carga B con lo que siga a MPR en la memoria. La instrucción siguiente se ocupa de ello y carga en B el valor "8", como ilustra la figura 3.22. Por el momento, los contenidos de D y E y H y L están indefinidos. La instrucción LD no perturba al bit de acarreo, de tal manera que también está indefinido el contenido del mismo.

ETIQUETA	INSTRUCCION	B	C	C	D	E	H	L
		--	--	-	--	--	--	--
MPY88	LD BC,(0200)	00	03	-	--	--	--	--
	LD B,08	08	03	-	--	--	--	--

Figura 3.22
El programa de multiplicación tras dos instrucciones.

La situación tras la ejecución de las primeras cinco instrucciones del programa (justo antes de MULT) se ilustra en la figura 3.23.

La instrucción SRL lleva a cabo un desplazamiento lógico a la derecha, de manera que el bit de ese extremo de MPR pasa al bit de acarreo. Como se ve en la figura 3.24, el contenido de MPR tras el desplazamiento es "0000 0001". El bit de acarreo C vale ahora "1". La operación no ha afectado a los demás

registros. Ahora, debe continuar usted mismo rellenando la tabla.

Al final de este capítulo, en la figura 3.42, se recoge una segunda repetición.

ETIQUETA	INSTRUCCION	B	C	C	D	E	H	L
		--	--	-	--	--	--	--
MPY88	LD BC,(0200)	00	03	-	--	--	--	--
	LD B, 08	08	03	-	--	--	--	--
	LD DE,(0202)	08	03	-	00	05	--	--
	LD D, 00	08	03	-	00	05	--	--
	LD HL,0000	08	03	-	00	05	00	00

Figura 3.23
El programa de multiplicación tras cinco instrucciones.

ETIQUETA	INSTRUCCION	B	C	C	D	E	H	L
		--	--	-	--	--	--	--
MPY88	LD BC,(0200)	00	03	-	--	--	--	--
	LD B, 08	08	03	-	--	--	--	--
	LD DE,(0202)	08	03	-	00	05	--	--
	LD D, 00	08	03	-	00	05	--	--
	LD HL,0000	08	03	-	00	05	00	00
MULT	SRL C	08	01	1	00	05	00	00
	JR NC,0114	08	01	1	00	05	00	00
	ADD HL,DE	08	01	0	00	05	00	05
NOADD	SLA E	08	01	0	00	0A	00	05
	RL D	08	01	0	00	0A	00	05
	DEC B	07	01	0	00	0A	00	05
	JP NZ,010F	07	01	0	00	0A	00	05

Figura 3.24
Un pase del bucle.

La figura 3.40 recoge el listado completo de los contenidos de todos los registros y banderas del Z80. En la figura 3.41 aparece un listado decimal y hexadecimal.

OTRAS POSIBILIDADES DE PROGRAMACION

El programa que acabamos de desarrollar puede escribirse de forma distinta. Como norma general, el programador debe estar en condiciones de modificar y mejorar cualquier progra-

ma. En este caso se ha desplazado el multiplicando antes de sumar, pero matemáticamente hubiera sido lo mismo desplazar el resultado una posición a la derecha antes de sumarlo al multiplicando. De hecho, se trata de un ejercicio interesante.

Ejercicio 3.19: *Escríbase un programa de multiplicación de 8 × 8 con el mismo algoritmo ya visto, pero desplazando el resultado una posición a la derecha, en lugar de hacer lo mismo hacia la izquierda con el multiplicando. Compárese con el programa anterior, y determínese si el nuevo tratamiento será o no más rápido. Las velocidades de ejecución de las instrucciones del Z80 se recogen en el capítulo siguiente.*

PROGRAMA DE MULTIPLICACION MEJORADO

El programa que acabamos de estudiar es una traducción directa del algoritmo. Sin embargo, *para programar con eficacia hay que dedicar atención al detalle*, para ver si puede acortarse el programa o acelerarse su velocidad de ejecución. Veamos ahora algunas opciones que mejoran el programa de multiplicación.

Paso 1

Una posibilidad de mejora consiste en aprovechar mejor las instrucciones del Z80. Así, las instrucciones penúltima y antepenúltima pueden reemplazarse por una sola:

```
DJNZ MULT
```

Se trata de una instrucción especial del Z80 de "salto automático" que decrementa el registro B y bifurca a una posición determinada si no vale "0". Hablando estrictamente, la instrucción no equivale por completo a las otras dos:

```
DEC B
JP NZ, MULT
```

porque especifica un *desplazamiento*, y sólo puede darse un salto dentro del intervalo − 126 a + 129. Sin embargo, en este caso debemos saltar a una posición alejada tan sólo unos pocos bytes, por lo que la mejora es factible. El programa resultante aparece en la figura 3.25.

```
MPY88B  LD      DE, (MPDAD)
        LD      BC, (MPRAD)
        LD      B, 8            CONTADOR DE
                                BIT
        LD      HL, 0
MULT    SRL     C
        JR      NC, NOADD
        ADD     HL, DE
NOADD   SLA     E
        RL      D
        DJNZ    MULT
        LD      (RESAD), HL
        RET
```

Figura 3.25
Programa de multiplicación mejorado, paso 1.

Paso 2

Como puede observarse en el programa inicial de la figura 3.14, se emplean tres operaciones de desplazamiento diferentes: el multiplicador se desplaza a la derecha, a continuación se desplaza a la izquierda el multiplicando MPD en dos operaciones, un desplazamiento a la izquierda del registro E más una permutación circular a la izquierda del D. Todo esto consume tiempo. Hay un "truco" habitual en la programación de la multiplicación, que se basa en el hecho de que cada vez que el multiplicador se desplaza un bit a la derecha, en el registro multiplicador queda libre otro bit. Así, suponiendo que el desplazamiento se produce hacia la derecha —caso del ejemplo anterior— queda libre un bit a la izquierda. Se observa también que el primer producto parcial (o "resultado") utiliza, como máximo, 9 bits. Si al principio hubiésemos reservado para el resultado un solo registro, podríamos haber utilizado la posición que deja libre el multiplicador para almacenar el noveno bit de aquél.

Tras el siguiente desplazamiento de MPR, el tamaño del producto parcial vuelve a aumentar en un bit, de tal manera que puede reservarse al principio un solo registro para el producto parcial y utilizar la posición libre que se produce al desplazar MPR; por tanto, para mejorar el programa, asignaremos MPR y RES a un par de registros. Lo ideal sería desplazarlos juntos en una sola operación, pero el Z80 sólo es capaz de desplazar 8 bits de una vez; como la mayor parte de los microprocesadores de 8 bits, no dispone de las instrucciones necesarias para desplazar 16 bits. (ADD HL,HL desplazan los 16 bits de HL una posición a la izquierda.)

Pero queda otro recurso. El Z80 —como el 8080— dispone de instrucciones especiales de adición de 16 bits que ya hemos

utilizado en este libro. Si el multiplicador y el resultado están almacenados en los registros apareados H y L, podemos usar la instrucción.

```
ADD   HL, HL
```

que suma los contenidos de H y L a sí mismos. Sumar un número a sí mismo equivale a duplicarlo, y en el sistema binario, duplicar un número equivale a desplazarlo hacia la izquierda una posición; por tanto, hemos realizado un desplazamiento de 16 bits de una sola vez. Por desgracia, el desplazamiento se ha producido hacia la izquierda y no hacia la derecha, como queríamos, pero no hay problema.

Conceptualmente, MPR puede desplazarse tanto a la izquierda como a la derecha. Hemos utilizado el algoritmo de desplazamiento a la derecha, porque es el que se usa en la suma convencional, pero no hay necesidad de hacer así las cosas. La operación de suma es conmutativa y admite la inversión del orden, por lo que da lo mismo desplazar MPR a la izquierda.

Para sacar partido a este desplazamiento simulado de 16 bits tendremos que desplazar MPR a la izquierda, por lo que éste residirá en el registro H, y el resultado en el L. La configuración de registros resultante se ilustra en la figura 3.26.

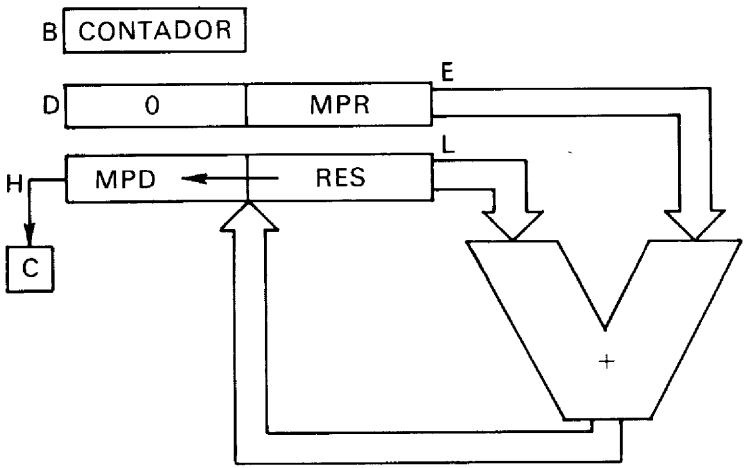

Figura 3.26
Registros del programa mejorado de multiplicación.

El resto del programa es básicamente igual al anterior. Aparece en la figura 3.27.

Al comparar este programa con el anterior se observa que se ha reducido la longitud del bucle de multiplicación (el número de instrucciones entre MULT y el salto). Este progra-

ma tiene menos instrucciones, y, en principio, avanzará con más rapidez, lo que demuestra la importancia de almacenar la información en los registros idóneos.

```
MUL88C  LD      HL,(MPRAD-1)
        LD      L,0
        LD      DE,(MPDAD)
        LD      D,0
        LD      B,8             CONTADOR
MULT    ADD     HL, HL          DESPLAZA-
                                MIENTO A LA
                                IZQUIERDA
        JR      NC, NOADD
        ADD     HL, DE
NOADD   DJNZ    MULT
        LD      (RESAD), HL
        RET
```

Figura 3.27
Programa mejorado de multiplicación, paso 2.

El diseño de programas "directo" da, por lo general, resultados que funcionan, pero que no son *óptimos*. Es, pues, importante aprender a sacar el máximo partido a los registros e instrucciones disponibles. Los ejemplos que hemos visto constituyen un enfoque racional de la selección de registros e instrucciones con vistas a optimizar la eficacia.

Ejercicio 3.20: *Calcúlese la velocidad de multiplicación con el último programa. Supóngase que tiene lugar una bifurcación en el 50 por 100 de los casos. El número de ciclos consumidos por cada una de las instrucciones aparece en el capítulo siguiente; la frecuencia del reloj será de 2 MHz (un ciclo = 0,5 μs).*

Ejercicio 3.21: *Obsérvese que hemos utilizado el par de registros D y E para albergar el multiplicando. ¿Cómo sería el programa anterior si hubiésemos utilizado el par B y C? (Una pista: sería necesario hacer una modificación al final.)*

Ejercicio 3.22: *¿Por qué hay que molestarse en poner a 0 el registro D al cargar MPD en E?*

Por último, vamos a prestar atención a un detalle que puede parecer irritante al programador no familizarizado con el Z80. Como habrá observado el lector, para cargar MPD en E a partir de la memoria es preciso cargar simultáneamente los dos registros D y E desde la dirección de memoria, porque, a menos que la dirección esté contenida en los registros H y L, no hay forma de traer un solo byte directamente y cargarlo en el

registro E; es una peculiaridad heredada del primitivo 8008, que carecía de direccionamiento directo. Con algunas mejoras, esa peculiaridad pasó al 8080, y mejoró todavía más en el Z80, en el que pueden traerse directamente 16 bits desde una dirección dada, pero no 8 bits, salvo hacia el registro A.

Ahora, una vez solucionado este problema un tanto misterioso, pasaremos a programar una multiplicación más complicada.

MULTIPLICACION DE 16 × 16

Para poner a prueba todo lo que ya hemos aprendido, vamos a multiplicar dos números de 16 bits, aunque supondremos que el resultado precisa únicamente 16 bits para que quepa en un par de registros.

Este, como en el primer ejemplo de multiplicación, pasará a los registros H y L (véase la figura 3.28). El multiplicando MPD reside en los registros D y E.

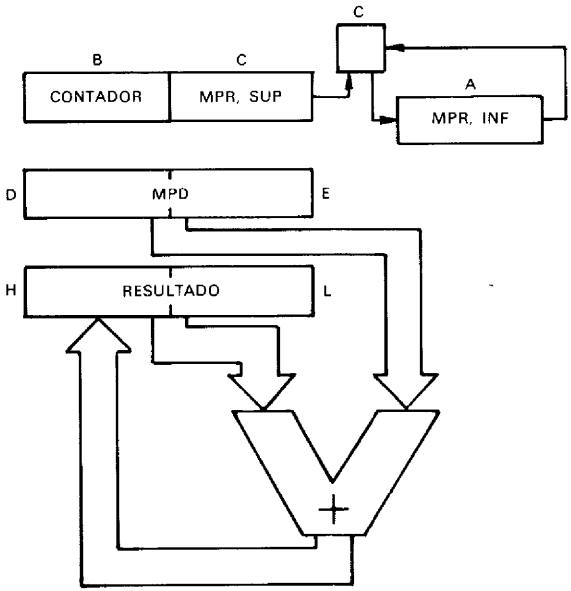

Figura 3.28
Registros de la multiplicación de 16 × 16.

Es tentador situar el multiplicador en los registros B y C, pero si queremos aprovechar la instrucción DJNZ, el registro B debe reservarse para el contador; en consecuencia, la mitad del multiplicador estará en el registro C, y la otra mitad, en el A (véase la figura 3.28). El programa de multiplicación aparece en la figura 3.29.

```
MUL16   LD      A, (MPRAD + 1)      MPR, SUPERIOR
        LD      C, A
        LD      A, (MPRAD)          MPR, INFERIOR
        LD      B, 16D              CONTADOR
        LD      DE, (MPDAD)         MPD
        LD      HL, 0
MULT    SRL     C                   DESPLAZAMIENTO DERECHA MPR, SUPERIOR
        RRA                         ROTACION CIRCULAR DERECHA MPR, INFERIOR
        JR      NC, NOADD           VERIFICAR ACARREO
        ADD     HL, DE              SUMAR MPD AL RESULTADO
NOADD   EX      DE, HL
        ADD     HL, HL              DOBLE DESPLAZAMIENTO MPD IZQUIERDA
        EX      DE, HL
        DJNZ    MULT
        RET
```

Figura 3.29
Programa de multiplicación de 16 × 16.

El programa es análogo al que hemos desarrollado antes. Las primeras seis instrucciones (desde la etiqueta MUL16 a la MULT) inician los registros con los contenidos necesarios. El que las dos mitades de MPR deban cargarse en operaciones separadas constituye una complicación adicional. Se supone que MPRAD señala la parte inferior de MPR en la memoria; la parte superior ocupa la posición secuencial siguiente (naturalmente, puede utilizarse la convención contraria). Una vez leída la parte superior de MPR en A, debe transferirse a C:

```
LD   A,(MPRAD + 1)
LD   C, A
```

Por último, la parte inferior de MPR puede leerse directamente en el acumulador:

```
LD   A, (MPRAD)
```

El resto de los registros —B, D, E, H y L— se inician de la forma habitual:

```
LD   B, 16D
LD   DE, (MPDAD)
LD   HL, 0
```

Hay que realizar un desplazamiento de 16 bits sobre el multiplicador, lo que exige dos operaciones independientes de desplazamiento o de rotación circular sobre los registros C y A:

```
MULT  SLR  C
      RRA
```

Tras el desplazamiento de 16 bits, el bit de la derecha de MPR, es decir, el BmS (bit menos significativo), ocupa el bit de acarreo C, en el que puede verificarse:

```
JR   NC, NOADD
```

Como es habitual, el multiplicando no se suma al resultado si el bit de acarreo es "0", pero sí se añade si es "1".

```
ADD  HL, DE
```

A continuación hay que desplazar el multiplicando MPD una posición hacia la izquierda.

Sin embargo, el Z80 no dispone de ninguna instrucción que permita desplazar simultáneamente los contenidos de los registros D y E una posición a la izquierda, y tampoco es posible sumar a sí mismos esos contenidos, que, por tanto, deben transferirse a H y L, duplicarse y devolverse otra vez a D y E. De todo esto se encargan las tres instrucciones siguientes:

```
NOADD   EX    DE, HL
        ADD   HL, HL
        EX    DE, HL
```

Por último, se reduce el contador B y se produce un salto al principio del bucle si esa reducción no lo lleva a "0".

```
DJNZ  MULT
```

Como siempre, pueden pensarse otras formas de organizar los registros para obtener, o no obtener, programas más cortos.

Ejercicio 3.23: *Cargar el multiplicador en los registros B y C, colocar el contador en A, escribir el programa de multiplicación correspondiente y discutir las ventajas o inconvenientes de esta organización de los registros.*

Ejercicio 3.24: *En el programa original de multiplicación de 16 bits de la figura 3.29, ¿habría alguna forma de desplazar MPD, contenido en los registros D y E, sin transferirlo a los H y L?*

Ejercicio 3.25: *Escríbase un programa de multiplicación de 16 × 16 bits que detecte resultados de más de 16 bits. Se trata de una sencilla mejora del programa básico.*

Ejercicio 3.26: *Escríbase un programa de multiplicación de 16 × 16 bits que admita resultados de 32 bits. La organización de registros sugerida aparece en la figura 3.30. Recuérdese que el resultado inicial tras la primera suma del bucle sólo necesitará 16 bits y que el multiplicador dejará un bit libre por cada iteración posterior.*

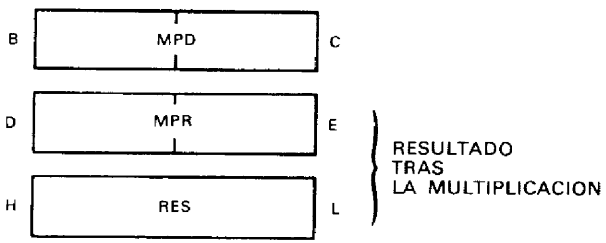

Figura 3.30
Multiplicación de 16 × 16 con resultado de 32 bits.

Pasemos ahora a la última de las operaciones aritméticas usuales: la división.

División binaria

El algoritmo de la división binaria es análogo al utilizado para la multiplicación: el divisor se resta, sucesivamente, de los bits de orden superior del dividendo; tras cada resta, se usa el resultado en lugar del dividendo inicial; simultáneamente, se incrementa cada vez en 1 el valor del cociente. A veces, el resultado de la resta es negativo, y a esa situación se le llama *sobrepasamiento;* para solucionarla, se restaura el resultado parcial, sumándole el divisor otra vez (naturalmente, hay que reducir simultáneamente en 1 el cociente). A continuación se desplazan un bit a la izquierda el cociente y el dividendo, y se repite el algoritmo. El diagrama de flujo se ilustra en la figura 3.31.

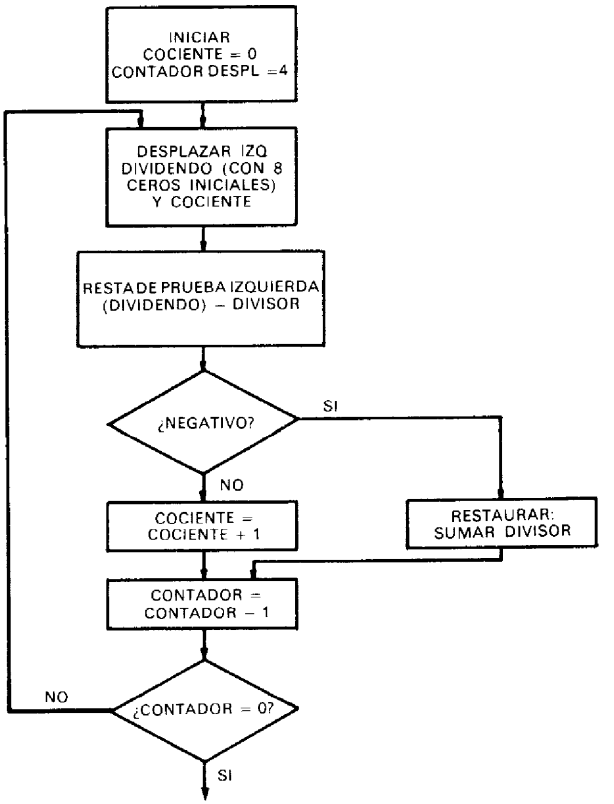

Figura 3.31
Diagrama de flujo de la división binaria de 8 bits.

Este método es el llamado de *restauración.* Hay una variante de ejecución más rápida llamada *sin restauración.*

DIVISION 16 POR 8

Examinemos, como ejemplo, una división de 16 × 8, que deja un cociente de 8 bits y un resto del mismo formato. La figura 3.32 recoge la disposición de los registros.

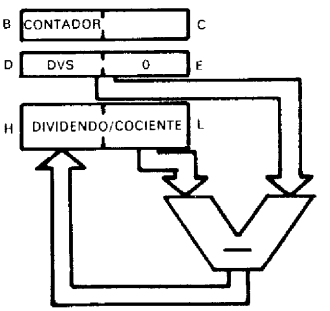

Figura 3.32
Registros de la división 16/8.

El programa aparece en la figura 3.33:

```
DIV168    LD      A,(DVSAD)      CARGAR DIVI-
                                 SOR
          LD      D,A            EN D
          LD      E,0
          LD      HL,(DVDAD)     CARGAR DIVI-
                                 DENDO DE 16
                                 BITS
          LD      B,8            INICIALIZAR
                                 EL CONTADOR
DIV       XOR     A              BORRAR BIT C
          SBC     HL,DE          DIVIDEN-
                                 DO - DIVISOR
          INC     HL             COCIENTE =
                                 COCIENTE + 1
          JP      P,NOADD        VERIFICAR SI
                                 EL RESTO ES
                                 POSITIVO
          ADD     HL,DE          RESTAURAR SI
                                 ES NECESARIO
          DEC     HL             COCIENTE =
                                 COCIENTE - 1
NOADD     ADD     HL,HL          DESPLAZAR
                                 DIVIDENDO A
                                 LA IZQUIERDA
          DJNZ    DIV            BUCLE HASTA
                                 QUE B = 0
          RET
```

Figura 3.33
Programa de división 16/8.

Las primeras cinco instrucciones cargan el divisor y el dividendo, respectivamente, en los registros correspondientes, y, además, inicializan el contador, en el registro B, al valor 8. Obsérvese que el registro B constituye el alojamiento idóneo del contador cuando se utiliza la instrucción especial del Z80, DJNZ:

```
DIV168    LD  A,(DVSAD)
          LD  D,A
          LD  E,0
          LD  HL,(DVDAD)
          LD  B,8
```

A continuación se resta el divisor del dividendo. Como hay que usar una instrucción SBC (no hay resta de 16 bits sin

acarreo), es preciso poner el acarreo a "0" antes de realizar la operación, cosa que puede hacerse de varias formas; el acarreo puede anularse con instrucciones como las siguientes:

```
XOR A
AND A
OR A
```

En este caso se ha utilizado XOR:

```
DIV   XOR A
```

Ahora puede efectuarse la resta:

```
SBC   HL, DE
```

Se anticipa que la resta dejará un resto positivo, paso que se conoce como "resta de prueba" (véase el diagrama de flujo de la figura 3.31); por tanto, el cociente se incrementa en 1. Si la resta falla (es decir, si el resto es negativo), será preciso reducir a continuación en 1 el cociente:

```
INC   HL
```

Se verifica el resultado de la resta:

```
JP   P, NOADD
```

Si el resto es positivo o 0, es que el resultado ha sido correcto, y no es preciso almacenarlo. El programa salta a la dirección NOADD. En caso contrario, el dividendo en curso debe ponerse con su valor anterior sumándole el divisor, a la vez que se resta 1 al cociente. Las siguientes instrucciones se encargan de ejectuar estos pasos:

```
ADD   HL, DE
DEC   HL
```

Por último, se desplaza a la izquierda el dividendo resultante, como anticipación a la siguiente resta de prueba. Se reduce el contador B y se comprueba si vale "0". Mientras esto no ocurra, se ejecuta el bucle:

```
NOADD   ADD   HL, HL
        DJNZ  DIV
        RET
```

Ejercicio 3.27: Verifíquese manualmente el funcionamiento de este programa de división cumplimentando la tabla de la figura 3.34, tal como se hizo en el ejercicio 3.18 con la multiplicación. Obsérvese que no es preciso introducir en esta tabla el contenido de D, porque no se modifica nunca.

ETIQUETA	INSTRUCCION	B	H	L

Figura 3.34
Tabla para el programa de división.

DIVISION DE 8 BITS

El programa que se propone aquí sigue un procedimiento de restauración, y deja en A un cociente complementado. Sirve para efectuar divisiones de 8 bits por 8 bits sin signo.

```
E ES EL DIVIDENDO
C ES EL DIVISOR
A ES EL COCIENTE
B ES EL RESTO

DIV88     XOR     A        BORRAR EL ACUMULADOR
          LD      B, 8     CONTADOR DEL BUCLE
LOOP88    RL      E        PERMUTACION CIRCULAR DE CY EN ACC-DIVIDENDO
          RLA              SALIDA DE CY
          SUB     C        DIVISOR DE LA RESTA DE PRUEBA
```

```
JR      NC, $ + 3     RESTA CORRECTA
ADD     A, C          RESTAURAR
                      ACUM, PONER CY
DJNZ    LOOP88
LD      B, A          PONER EL RESTO
                      EN B
LD      A, E          OBTENER EL CO-
                      CIENTE
RLA                   DESPLAZAR EL
                      ULTIMO BIT DEL
                      RESULTADO
CPL                   BITS DE COM-
                      PLEMENTO
RET
```

Nota: El símbolo "$" de la sexta instrucción representa el valor del contador de programa.

DIVISION SIN RESTAURACION

El programa siguiente lleva a cabo una división de un entero de 16 bits por otro de 15 bits mediante una técnica sin restauración. IX señala el dividendo e IY el divisor (no cero). (Véase la figura 3.35.)

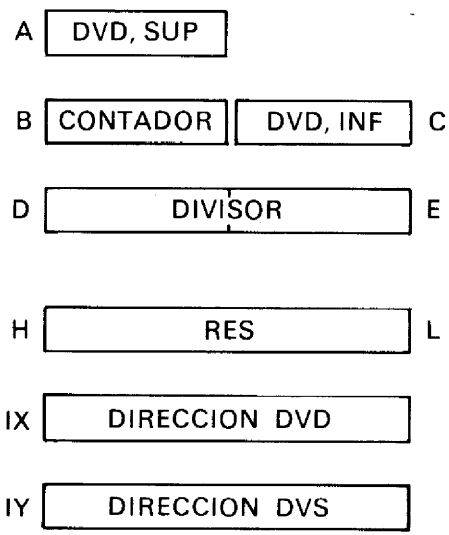

Figura 3.35
Registros de la división sin restauración.

El registro B, inicialmente de valor 16, es el contador.
A y C contienen el dividendo.
D y E contienen el divisor.
H y L contienen el resultado.
El dividendo de 16 bits se desplaza hacia la izquierda mediante las instrucciones:

```
RL    C
RLA
```

El resto se desplaza hacia la izquierda mediante la intrucción:

```
ADC   HL, HL.
```

El cociente final queda en B, C y el resto en HL. El programa continúa.

```
DIV16     LD      B,(IX + 1)
          LD      C,(IX)
          LD      D,(IY + 1)
          LD      E,(IY)
          LD      A, D
          OR      E              PARTE SUPERIOR DEL (DIVISOR) O PARTE INFERIOR DEL (DIVISOR)
          JR      Z, ERROR       VERIFICAR SI DIVISOR = 0
          LD      A, B           OBTIENE (DVD) SUP
          LD      HL,0           BORRAR RESULTADO
          LD      B, 16D         CONTADOR
TRIALSB   RL      C              ROTACION CIRCULAR RESULTADO + ACC IZQ
          RLA
          ADC     HL, HL         DESPLAZAR A LA IZQUIERDA. NO PONE ACARREO
          SBC     HL, DE         MENOS DIVISOR
```

```
NULL    CCF                 BIT DE RESULTADO
        JR      NC, NGV     ¿ACUMULADOR NEGATIVO?
PTV     DJNZ    TRIALSB     ¿CONTADOR CERO?
        JP      DONE
RESTOR  RL      C           ROTACION CIRCULAR RESULTADO + ACC IZQ
        RLA
        ADC     HL, HL      COMO ARRIBA
        AND     A
        ADC     HL, DE      RESTAURAR SUMANDO DVSR
        JR      C, PTV      RESULTADO POSITIVO
        JR      Z, NULL     RESULTADO CERO
NGV     DJNZ    RESTOR      ¿CONTADOR CERO?
DONE    RL      C           DESPLAZAR BIT DE RESULTADO
        RLA
        ADD     HL, DE      RESTO CORRECTO
        LD      B, A        COCIENTE EN B, C
        RET
```

Ejercicio 3.28: *Compárese el programa anterior con el siguiente, que utiliza una técnica de restauración:*

```
DIVIDENDO EN AC
DIVISOR EN DE
COCIENTE EN AC
RESTO EN HL

DIV16   LD      HL, 0       BORRAR ACUMULADOR
```

```
        LD      B, 16D      PONER CONTADOR
LOOP16  RL      C           ROTACION CIRCULAR ACC-RESULTADO
        RLA
        ADC     HL, HL      DESPLAZAMIENTO A LA IZQUIERDA
        SBC     HL, DE      DIVISOR RESTA DE PRUEBA
        JR      NC, $ + 3   RESTA CORRECTA
        ADD     HL, DE      RESTAURAR ACUMULADOR
        CCF                 CALCULAR BIT DEL RESULTADO
        DJNZ    LOOP16      EL CONTADOR NO ES CERO
        RL      C           DESPLAZAR EL ULTIMO BIT DEL RESULTADO
        RLA
        RET
```

Nota: El símbolo "$" de la séptima instrucción significa "posición en curso".

Operaciones lógicas

La otra clase de instrucciones que puede ejecutar la ALU son las *instrucciones lógicas:* AND, OR y OR exclusivo (XOR). También podrían incluirse aquí las operaciones de desplazamiento y rotación circular, utilizadas ya repetidamente, y la instrucción de comparación, que en el Z80 se llama CP. El uso individual de AND, OR y XOR se describirá en el capítulo 4.

Vamos ahora a desarrollar un breve programa para comprobar si una posición de memoria dada llamada LOC contiene el valor "0", el "1" o algún otro.

Utilizaremos en el programa la instrucción de comparación y realizaremos una serie de comprobaciones lógicas. Según el resultado de la comparación, se ejecutará uno u otro segmento del programa.

El programa es éste:

```
                LD   A,(LOC)   LEER EL CARAC-
                               TER DE LOC
                CP   00H       COMPARAR CON
                               0
                JP   Z,CERO    ¿ES A 0?
                CP   01H       COMPARAR CON
                               1
                JP   Z,UNO
NOENCONTRADO...
CERO            ...
UNO             ...
```

La primera instrucción, "LD A,(LOC)", lee el contenido de la posición de memoria LOC y la carga en el acumulador. El contenido es el carácter que deseamos comprobar, y su valor se compara con 0 mediante la instrucción:

```
CP 00H
```

La instrucción compara el contenido del acumulador con el valor hexadecimal "00", es decir, con el binario "0000 0000". Esta instrucción de comparación pone el bit Z del registro de estado al valor "1", si el resultado es afirmativo, lo que se comprueba mediante la instrucción:

```
JP Z,CERO
```

La instrucción de salto comprueba el valor del bit Z. Si el resultado de la comparación ha sido positivo, Z valdrá uno y se efectuará el salto a la dirección CERO. Si el resultado es negativo, se ejecutará la instrucción siguiente de la secuencia:

```
CP 01H
```

De la misma manera, la instrucción de salto bifurcará a la posición UNO si la comparación es positiva. Si ninguna de las comparaciones fuesen positivas, se ejecutaría la instrucción que ocupa la posición NOENCONTRADO.

```
                JP   Z,UNO
NOENCONTRADO...
```

La finalidad de este programa es poner de relieve el valor de la instrucción de comparación seguida de salto. Esta combinación podrá utilizarse en muchos de los programas que vienen a continuación.

Ejercicio 3.29: *Búsquese en el capítulo siguiente la definición de la instrucción LD A,(LOC) y examínese su efecto sobre las banderas en caso de que hubiere alguno. ¿Es imprescindible la segunda instrucción de este programa (CP 00H)?*

Ejercicio 3.30: *Escríbase un programa que lea el contenido de la posición de memoria "24" y bifurque a la dirección llamada "ESTRELLA", si en dicha posición se encuentra el símbolo "*". La representación binaria de "*" es "00101010".*

Resumen de instrucciones

Hemos estudiado casi todas las instrucciones importantes del Z80 utilizándolas; hemos transferido valores entre la memoria y los registros; hemos realizado operaciones aritméticas y lógicas con esos valores; los hemos verificado, y, según el resultado obtenido, hemos ejecutado unas u otras porciones del programa. También hemos aprovechado las instrucciones "automáticas" especiales del Z80, como DJNZ, para acortar programas. Más adelante recurriremos a otras instrucciones automáticas, como LDDR, CPIR o INIR.

Se ha sacado el máximo partido de las características peculiares del Z80, como las instrucciones para registros de 16 bits que simplifican los programas (téngase en cuenta que tales características no existen en el 8080, del que el Z80 es una versión optimizada).

Ya hemos hablado de una estructura llamada bucle, y a continuación estudiaremos otra muy importante: la subrutina.

Subrutinas

Conceptualmente, una subrutina no es sino un bloque de instrucciones al que el programador ha adjudicado un nombre. Desde un punto de vista práctico, toda subrutina empieza con una instrucción especial, llamada *declaración de subrutina*, que la identifica como tal al ensamblador, y termina con otra,

llamada *retorno (return)*. Veremos primero el funcionamiento de una subrutina dentro de un programa, para aprender a apreciar su valor, y a continuación pasaremos a la realización práctica de la misma.

La utilización de una subrutina se muestra en la figura 3.36. El programa principal está representado a la izquierda, y la subrutina, a la derecha. Las líneas del primero se ejecutan una tras otra hasta que aparece una instrucción "CALL SUB" o *llamada a subrutina*, que transfiere el control a ésta, de manera que la primera instrucción que se ejecuta tras CALL SUB es la primera de las que componen la subrutina en cuestión, como muestra la flecha 1 de la figura.

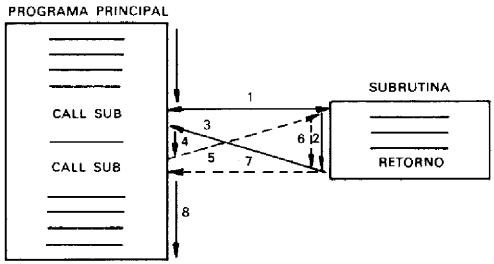

Figura 3.36
Llamadas a una subrutina.

A continuación se ejecuta el subprograma de la subrutina de la misma forma que cualquier otro programa. Supondremos en principio que la subrutina no incluye, a su vez, otras llamadas. La última instrucción de la subrutina es RETORNO, lo que provoca la devolución del control al programa principal. Tras dicha instrucción, la primera que se ejecuta es la que sigue a CALL SUB en el mencionado programa principal, hecho que muestra la flecha 3 de la figura. A continuación prosigue de la forma normal la ejecución del programa principal (flecha 4).

Dentro del programa principal surge una segunda instrucción CALL SUB, y tiene lugar una segunda transferencia, simbolizada por la flecha 5. Esto significa que la subrutina vuelve a ejecutarse, una vez más, tras la nueva llamada.

En el momento en que aparece la instrucción RET dentro de la subrutina se ejecuta la instrucción del programa principal que sigue a la CALL SUB, como muestra la flecha 7, y tras ella el resto de dicho programa (flecha 8).

El efecto de las instrucciones especiales CALL SUB y RET está, por tanto, claro. Falta por saber qué utilidad tienen las subrutinas.

Lo más valioso de una subrutina es que puede llamarse desde cualquier punto del programa principal y utilizarse cuantas veces sea necesario *sin necesidad de volver a escribirla*. De

esta forma se ahorra espacio de memoria y tiempo de programación, con la consiguiente simplificación del diseño de programas.

Ejercicio 3.31: *¿Cuál es el principal inconveniente de la subrutina?*

Respuesta: El inconveniente del trabajo con subrutinas se deduce fácilmente con sólo examinar el flujo de control entre ellas y el programa principal: *la velocidad general de ejecución es más baja*, porque es preciso ejecutar las instrucciones especiales CALL SUB y RETURN.

REALIZACION PRACTICA DEL MECANISMO DE LA SUBRUTINA

Vamos a ver de qué forma se tratan en el interior del microprocesador las dos instrucciones especiales CALL SUB y RET. El efecto de CALL SUB es tomar la instrucción siguiente de una nueva dirección. Como se recordará (y si no se recuerda deberá repasarse el capítulo 1), la dirección de la instrucción que debe ejecutarse a continuación de la que está en curso se encuentra en el contador del programa PC. De esto se deduce que CALL SUB modifica el contenido del PC, en el que carga la dirección de comienzo de la subrutina. Pero, *¿basta con eso?*

Para responder a esa pregunta, consideremos la segunda de las instrucciones especiales: RET. Esta instrucción determina la vuelta a la instrucción que sigue a CALL SUB, lo que sólo es posible si su dirección se ha conservado en algún sitio. Dicha dirección es el valor del contador del programa en el momento en que se llega a CALL SUB, porque el contador del programa se incrementa automáticamente cada vez que se usa (repasar el capítulo 1). Esta es precisamente la instrucción que debe conservarse para ejecutar más adelante RET.

El problema que se plantea es dónde conservar esa dirección de retorno, que debe permanecer en un sitio en el que no pueda borrarse de ninguna manera.

Antes de seguir, vamos a analizar la situación que plantea la figura 3.37: la subrutina 1 contiene una llamada a otra subrutina 2 o SUB2. El mecanismo expuesto debe funcionar también en este caso. Naturalmente, el número de llamadas internas no tiene por qué estar limitado a dos, y puede ser cualquiera N. En general, cada vez que se encuentre una nueva llamada CALL, el mecanismo de ejecución debe volver a almacenar el contador

del programa, lo que significa que hacen falta al menos 2N posiciones de memoria para este mecanismo. Además, hay que volver, primero, desde SUB2 y, a continuación, desde SUB1. En otras palabras, lo que hace falta es una estructura capaz de conservar el orden cronológico en que se han almacenado las direcciones.

Dicha estructura tiene un nombre, y ya hemos hablado de ella: es *la pila*. La figura 3.39 recoge el contenido real de la pila durante las sucesivas llamadas a las subrutinas. Examinemos primero el programa principal. La primera llamada se encuentra en la dirección 100: CALL SUB1. Supongamos que, en el microprocesador, la llamada a la subrutina utiliza 3 bytes (RST es una excepción); por tanto, la siguiente dirección secuencial no es "101", sino "103"; la instrucción CALL utiliza las direcciones "100", "101" y "102"; como la unidad de control del Z80 "sabe" que se trata de una instrucción de 3 bytes, el valor del contador del programa cuando la llamada haya sido decodificada en su totalidad será "103". El efecto de la llamada será cargar el valor "280" —dirección de partida de SUB1— en el contador del programa.

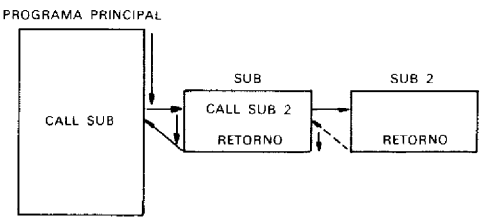

Figura 3.37
Llamadas internas.

Ya estamos en condiciones de estudiar el efecto de la instrucción RET y el funcionamiento del mecanismo de la pila. La ejecución avanza dentro de SUB2 hasta que encuentra la instrucción RET en el momento 3. El efecto de RET no es sino transferir la cabecera de la pila al contador del programa. En otras palabras, el contador recupera el valor que tenía antes de la entrada a la subrutina. La parte superior de la pila es, en nuestro ejemplo, "303". La figura 3.39 indica que, en el momento 3, el valor "303" ha pasado de la pila al contador del programa. Como resultado, la ejecución de la instrucción avanza a partir de la dirección "303". En el ciclo 4 se encuentra la instrucción RET de SUB1. El valor de la cabecera de la pila es "103", que pasa al contador del programa. Como consecuencia, el programa se ejecuta, a partir de la posición de memoria "103", dentro del programa principal, que es precisamente el efecto deseado. La figura 3.39 demuestra que en el ciclo 4 la pila está de nuevo vacía. El mecanismo funciona.

Este mecanismo de llamada a subrutinas actúa hasta que la pila alcanza su dimensión máxima, y por eso los primitivos microprocesadores con pilas de 4 u 8 registros estaban limitados a 4 u 8 niveles de llamadas a subrutinas.

Obsérvese que en las figuras 3.37 y 3.38 las subrutinas se han simbolizado a la derecha del programa principal; ello obedece únicamente a razones de claridad de la representación, porque las subrutinas se escriben exactamente igual que las instrucciones normales del programa. En la hoja de papel en que aparece el listado completo del programa, las subrutinas pueden colocarse al principio del texto, en el centro del mismo o al final, y se identifican por la declaración de subrutina que las precede. Las instrucciones especiales indican al ensamblador que lo que sigue debe tratarse como una subrutina. Estas *seudoinstrucciones* del ensamblador se estudiarán en el capítulo 10.

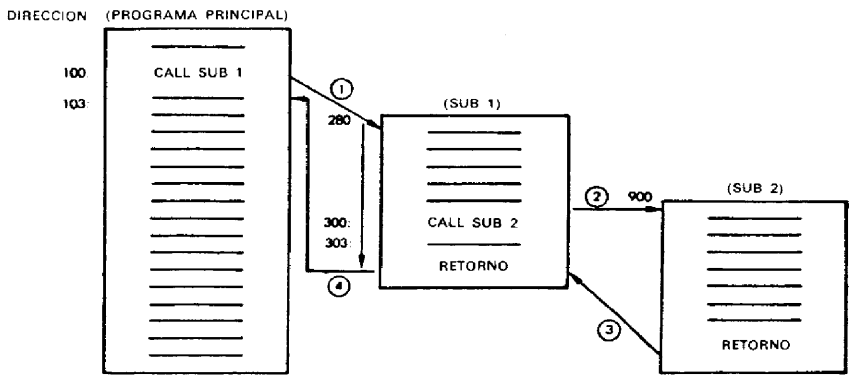

Figura 3.38

Llamadas a subrutinas.

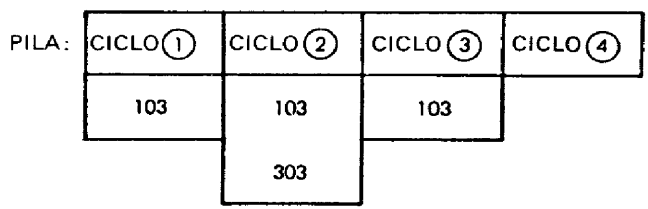

Figura 3.39
Estado de la pila a lo largo del tiempo.

SUBRUTINAS DEL Z80

Ya se han expuesto los conceptos básicos relativos a las subrutinas. Sabemos que hace falta una pila para que funcionen. El Z80 dispone de un puntero de pila de 16 bits, de manera que la pila puede residir en cualquier lugar de la memoria y albergar hasta 64K (1K = 1024), suponiendo que

estén disponibles para ese fin. En la práctica, el programador define, antes de escribir el programa, la dirección de partida de la pila y su dimensión máxima, reservándose, en consecuencia, la parte necesaria de la memoria.

La instrucción de llamada a subrutinas del Z80 es CALL, y existe en dos versiones: llamada directa o incondicional —CALL DIRECCION—, que es la que ya se ha descrito, y llamada condicional, peculiar del Z80, y en virtud de la cual se llama una subrutina si se satisface determinada condición. Por ejemplo, CALL NZ, SUB1 llamará la subrutina 1 si la bandera Z es 0 en el momento de la verificación. Es una instrucción potente, porque muchas llamadas a subrutinas son condicionales, es decir, sólo se producen si se cumple una condición específica.

CALL CC, NN sólo se ejecuta si es cierta la condición especificada por "CC"; CC es un conjunto de tres bits (bits 3, 4 y 5 del código de operación) capaz de especificar hasta ocho condiciones, que corresponden a cada una de las cuatro banderas "Z", "C", "P/V" y "S", que pueden ser cero o no cero.

Hay también dos tipos de instrucciones de vuelta: RET y RET CC.

RET es la instrucción de vuelta básica. Ocupa un byte, y hace que los dos bytes superiores de la pila vuelvan a instalarse en el contador del programa. Es incondicional.

RET CC tiene el mismo efecto, pero sólo se ejecuta si las condiciones especificadas por CC son ciertas. Los bits condicionales son los mismos de la instrucción CALL que acaban de describirse.

Además, hay dos tipos especializados de retorno que sirven para acabar rutinas de interrupción: RETI y RETN. Se describirán en el capítulo de instrucciones del Z80 y en el de interrupciones.

Hay, por fin, otra instrucción especializada análoga a una llamada a subrutina, pero que sólo permite al programa desviarse a una de ocho posiciones de partida localizadas en la página cero. Se trata de la instrucción RST P, una instrucción de 1 byte que almacena automáticamente el contador del programa en la pila y desvía el programa a la dirección especificada en el campo de tres bits P; éste corresponde a los bits 3, 4 y 5 de la instrucción, multiplicados por ocho.

En otras palabras, si los bits 3, 4 y 5 son "000", el salto se producirá a la posición 00H. Si son "001", el salto será a 08H, etcétera, y así hasta 111, que provoca la bifurcación a la posición 38H. La instrucción RST es muy eficaz en términos de velocidad, porque tiene un solo byte, aunque a cambio de saltar únicamente a ocho posiciones en la página cero; además, estas

direcciones de la página cero están separadas nada más que por ocho bytes. Se trata de una instrucción procedente del 8080 que se usa mucho para interrupciones, como se describirá en el capítulo correspondiente. No obstante, el programador puede utilizarla para cualquier otro fin, y debe considerarse como una posible llamada a una subrutina especializada.

EJEMPLOS DE SUBRUTINAS

Casi todos los programas desarrollados hasta el momento, y la mayor parte de los que vamos a desarrollar, se escribirían normalmente como subrutinas. Así, el programa de multiplicación es normal que se use en muchos puntos de un programa general; por tanto, para clarificar y facilitar el desarrollo de programas, conviene definir una subrutina llamada, por ejemplo, MULT; al final de la misma no hay más que añadir la instrucción RET.

Ejercicio 3.32: *Si se usa MULT como subrutina, ¿"dañará" algunos de los registros o banderas internos?*

RECURRENCIA

Se llama recurrencia a la llamada a una subrutina desde ella misma. Si se ha comprendido el mecanismo de ejecución práctica de subrutinas, podrá responderse a la siguiente pregunta:

Ejercicio 3.33: *¿Es posible que una subrutina se llame a sí misma? (En otras palabras, ¿funcionará todo correctamente si una subrutina se llama a sí misma?) Si no está seguro de la respuesta, dibuje la pila y ocúpela con las direcciones sucesivas; observe a continuación los registros y la memoria (véase ejercicio 3.18) y determine si hay algún problema.*

Las interrupciones se estudiarán en el capítulo 6, dedicado a las técnicas de entrada y salida. Todos los retornos, con excepción de los que proceden de interrupciones, son instrucciones de un byte; por su parte, todas las llamadas —excepto RST— son instrucciones de tres bytes.

Ejercicio 3.34: *Consulte en el capítulo siguiente los tiempos de ejecución de las instrucciones CALL y RET. ¿Por qué el retorno de una subrutina es mucho más rápido que la llamada a la misma? (Una pista: si la respuesta no parece obvia, repásese una vez más el funcionamiento de la pila del mecanismo de subrutinas y analícense las operaciones internas que deben llevarse a cabo.)*

PARAMETROS DE SUBRUTINAS

Cuando se llama a una subrutina, normalmente se espera que actúe sobre ciertos datos. Así, en el caso de la multiplicación, hay que transmitir a la subrutina dos números para que sean sometidos a esa operación. Como ya vimos en el caso de la rutina de multiplicación, el multiplicando y el multiplicador se encuentran en posiciones de memoria dadas. He aquí, pues, un procedimiento de paso de parámetros: a través de la memoria. Pero hay, además, otras dos técnicas, lo que da lugar a tres métodos:

1. A través de los registros.
2. A través de la memoria.
3. A través de la pila.

Usar los *registros* para transferir parámetros tiene la ventaja, suponiendo que haya registros disponibles, de que no es preciso trabajar con posiciones fijas de memoria, de manera que la subrutina es independiente de la memoria. Si se utiliza una posición de memoria fija, cualquier otro usuario del programa deberá tener mucho cuidado para seguir la misma convención y asegurarse de que esa posición está realmente libre (véase el ejercicio 3.19). Por eso, en muchos casos, se reserva un bloque de posiciones de memoria para transferir parámetros entre varias subrutinas.

Usar la *memoria* tiene la ventaja de la flexibilidad (pueden usarse más datos), pero a cambio de un rendimiento inferior y de tener que ligar la subrutina a un área fija de la memoria.

La ubicación de los parámetros en la *pila* tiene la misma ventaja que el uso de los registros: la independencia de la memoria. La subrutina "sabe" que recibirá, por ejemplo, dos parámetros almacenados en la cabecera de la pila. Por supuesto, también tiene inconvenientes: la pila se satura de datos, con la consiguiente reducción del número de niveles de llamadas a subrutinas. También complica considerablemente el manejo de la pila, y puede exigir el empleo de varias de estas estructuras de datos.

La elección es responsabilidad del programador; pero, en general, se prefiere conservar la mayor independencia posible con respecto a las posiciones reales de memoria.

Si no hay registros disponibles, la pila es una alternativa a considerar. Sin embargo, cuando es necesario pasar a la subrutina mucha información, ésta puede residir directamente en memoria. Una forma elegante de resolver el problema de transmitir un bloque de datos consiste simplemente en transmitir un puntero dirigido a la información (un puntero es la direc-

ción del principio del bloque). El puntero puede pasarse por medio de un registro, o de la pila (hacen falta dos posiciones de pila para almacenar una dirección de 16 bits), o bien por medio de una o varias posiciones fijas de memoria.

Por último, si ninguna de las dos soluciones es aplicable, habrá que acordar con la subrutina una posición fija en memoria (el "buzón de correos").

Ejercicio 3.35: *¿Cuál de los tres métodos mencionados será mejor para las recurrencias?*

BIBLIOTECA DE SUBRUTINAS

La organización de las diversas porciones de un programa en forma de subrutinas identificables tiene la ventaja de que pueden ponerse a punto independientemente y de que se les puede asignar un nombre mnemotécnico. Si pueden utilizarse en segmentos diferentes del programa serán intercambiables y, por tanto, podrán formar parte de una biblioteca de subrutinas. Sin embargo, en programación no existe ninguna panacea, y acostumbrarse a convertir cualquier grupo de instrucciones que se repita por su función en una subrutina dará lugar a un rendimiento escaso. El programador deberá aprender a equilibrar las ventajas con los inconvenientes.

Resumen

Hemos visto en este capítulo cómo manipulan internamente la información las instrucciones del Z80. Los algoritmos traducidos a programas se han hecho cada vez más complicados y, además, han servido para utilizar y explicar los tipos de instrucciones más importantes.

También hemos definido varias estructuras de uso continuo, como bucles, pilas y subrutinas.

El lector deberá tener ya una idea básica de lo que es programar y de las principales técnicas puestas en juego en las aplicaciones normales. Pasemos, pues, a estudiar en detalle cada una de las instrucciones.

```
        A=00 BC=0000 DE=0000 HL=0000 S=0300 P=0100 0100' LD   BC,(0200)
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0200')
        A=00 BC=0003 DE=0000 HL=0000 S=0300 P=0104 0104' LD   B,08
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
        A=00 BC=0803 DE=0000 HL=0000 S=0300 P=0106 0106' LD   DE,(0202)
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0202')
        A=00 BC=0803 DE=0005 HL=0000 S=0300 P=010A 010A' LD   D,00
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
        A=00 BC=0803 DE=0005 HL=0000 S=0300 P=010C 010C' LD   HL,0000
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0000')
        A=00 BC=0803 DE=0005 HL=0000 S=0300 P=010F 010F' SRL  C
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
      C A=00 BC=0801 DE=0005 HL=0000 S=0300 P=0111 0111' JR   NC,0114
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0114')
      C A=00 BC=0801 DE=0005 HL=0000 S=0300 P=0113 0113' ADD  HL,DE
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
        A=00 BC=0801 DE=0005 HL=0005 S=0300 P=0114 0114' SLA  E
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    V   A=00 BC=0801 DE=000A HL=0005 S=0300 P=0116 0116' RL   D
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00 BC=0801 DE=000A HL=0005 S=0300 P=0118 0118' DEC  B
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  N     A=00 BC=0701 DE=000A HL=0005 S=0300 P=0119 0119' JP   NZ,010F
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (010F')
  N     A=00 BC=0701 DE=000A HL=0005 S=0300 P=010F 010F' SRL  C
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V C   A=00 BC=0700 DE=000A HL=0005 S=0300 P=0111 0111' JR   NC,0114
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0114')
Z V C   A=00 BC=0700 DE=000A HL=0005 S=0300 P=0113 0113' ADD  HL,DE
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00 BC=0700 DE=000A HL=000F S=0300 P=0114 0114' SLA  E
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    V   A=00 BC=0700 DE=0014 HL=000F S=0300 P=0116 0116' RL   D
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00 BC=0700 DE=0014 HL=000F S=0300 P=0118 0118' DEC  B
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  N     A=00 BC=0600 DE=0014 HL=000F S=0300 P=0119 0119' JP   NZ,010F
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (010F')
  N     A=00 BC=0600 DE=0014 HL=000F S=0300 P=010F 010F' SRL  C
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00 BC=0600 DE=0014 HL=000F S=0300 P=0111 0111' JR   NC,0114
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0114')
Z V     A=00 BC=0600 DE=0014 HL=000F S=0300 P=0114 0114' SLA  E
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    V   A=00 BC=0600 DE=0028 HL=000F S=0300 P=0116 0116' RL   D
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00 BC=0600 DE=0028 HL=000F S=0300 P=0118 0118' DEC  B
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  N     A=00 BC=0500 DE=0028 HL=000F S=0300 P=0119 0119' JP   NZ,010F
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (010F')
  N     A=00 BC=0500 DE=0028 HL=000F S=0300 P=010F 010F' SRL  C
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00 BC=0500 DE=0028 HL=000F S=0300 P=0111 0111' JR   NC,0114
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0114')
Z V     A=00 BC=0500 DE=0028 HL=000F S=0300 P=0114 0114' SLA  E
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    V   A=00 BC=0500 DE=0050 HL=000F S=0300 P=0116 0116' RL   D
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00 BC=0500 DE=0050 HL=000F S=0300 P=0118 0118' DEC  B
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  N     A=00 BC=0400 DE=0050 HL=000F S=0300 P=0119 0119' JP   NZ,010F
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (010F')
  N     A=00 BC=0400 DE=0050 HL=000F S=0300 P=010F 010F' SRL  C
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00 BC=0400 DE=0050 HL=000F S=0300 P=0111 0111' JR   NC,0114
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0114')
Z V     A=00 BC=0400 DE=0050 HL=000F S=0300 P=0114 0114' SLA  E
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
S   V   A=00 BC=0400 DE=00A0 HL=000F S=0300 P=0116 0116' RL   D
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00 BC=0400 DE=00A0 HL=000F S=0300 P=0118 0118' DEC  B
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  N     A=00 BC=0300 DE=00A0 HL=000F S=0300 P=0119 0119' JP   NZ,010F
       A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (010F')
```

Figura 3.40
Listado completo de la multiplicación.

```
    N   A=00 BC=0300 DE=00A0 HL=000F S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00 BC=0300 DE=00A0 HL=000F S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (0114')
Z V     A=00 BC=0300 DE=00A0 HL=000F S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  C     A=00 BC=0300 DE=0040 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
        A=00 BC=0300 DE=0140 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    N   A=00 BC=0200 DE=0140 HL=000F S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (010F')
    N   A=00 BC=0200 DE=0140 HL=000F S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00 BC=0200 DE=0140 HL=000F S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (0114')
Z V     A=00 BC=0200 DE=0140 HL=000F S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
S       A=00 BC=0200 DE=0180 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
        A=00 BC=0200 DE=0280 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    N   A=00 BC=0100 DE=0280 HL=000F S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (010F')
    N   A=00 BC=0100 DE=0280 HL=000F S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00 BC=0100 DE=0280 HL=000F S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (0114')
Z V     A=00 BC=0100 DE=0280 HL=000F S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V C   A=00 BC=0100 DE=0200 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  V     A=00 BC=0100 DE=0500 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z  N    A=00 BC=0000 DE=0500 HL=000F S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (010F')
Z  N    A=00 BC=0000 DE=0500 HL=000F S=0300 P=011C 011C' LD   (0204),HL
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (0204')
Z  N    A=00 BC=0000 DE=0500 HL=000F S=0300 P=011F 011F' NOP
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
```

Figura 3.40
Listado completo de la multiplicación (continuación).

RESPUESTAS AL EJERCICIO 3.18 (MULTIPLICACION):

```
0100                10          ORG  #0100
0100 0002           20 MPRAD    DEFW #0200
0102 0202           30 MPDAD    DEFW #0202
0104 0402           40 RESAD    DEFW #0204
                    50 ;
0106 ED4B0001       60 MP488    LD   BC,(MPRAD)   ;CARGA MULTIPLICADOR EN C
010A 0608           70          LD   B,8          ;B ES CONTADOR DE BIT
010C ED5B0201       80          LD   DE,(MPDAD)   ;CARGA MULTIPLICANDO EN E
0110 1600           90          LD   D,0          ;INICIALIZA D
0112 210000        100          LD   HL,0         ;PONE A 0 EL RESULTADO
0115 CB39          110 MULT     SRL  C            ;SHIFT AL ACARREO DEL BIT
                   115 ;                          ;MULTIPLICADOR
0117 3001          120          JR   NC,NOADD     ;COMPRUEBA EL ACAREO
0119 19            130          ADD  HL,DE        ;SUMA AL RESULTADO MPD
011A CB23          140 NOADD    SLA  E            ;SHIFT IZQUIERDA DE MPD
011C CB12          150          RL   D            ;GUARDA EL BIT EN D
011E 05            160          DEC  B            ;DECREMENTA EL CONTADOR DE SHIFT
011F C21501        170          JP   NZ,MULT      ;REPETIR SI CONTADOR<>0
0122 220401        180          LD   (RESAD),HL   ;ALMACENA EL RESULTADO
```

Figura 3.41
Programa de multiplicación (Hex).

ETIQUETA	INSTRUCCION	B	C	C (ACARREO)	D	E	H	L
		00	00	0	00	00	00	00
MP488	LD BC,(0200)	00	03	0	00	00	00	00
	LD B, 08	08	03	0	00	00	00	00
	LD DE,(0202)	08	03	0	00	05	00	00
	LD D, 00	08	03	0	00	05	00	00
	LD HL,0000	08	03	0	00	05	00	00
MULT	SRL C	08	01	1	00	05	00	00
	JR NC,0114	08	01	1	00	05	00	00
	ADD HL,DE	08	01	0	00	05	00	05
NOADD	SLA E	08	01	0	00	0A	00	05
	RL D	08	01	0	00	0A	00	05
	DEC B	07	01	0	00	0A	00	05
	JP NZ,010F	07	01	0	00	0A	00	05
MULT	SRL C	07	00	1	00	0A	00	05
	JR NC,0114	07	00	1	00	0A	00	05
	ADD HL,DE	07	00	0	00	0A	00	0F
NOADD	SLA E	07	00	0	00	14	00	0F
	RL D	07	00	0	00	14	00	0F
	DEC B	06	00	0	00	14	00	0F
	JP NZ,010F	06	00	0	00	14	00	0F

Figura 3.42
Dos repeticiones del bucle.