

indescomp

CODIGO MAQUINA zx spectrum

para principiantes



LENGUAJE MAQUINA

ZX
spectrum

para
principiantes

indescomp

Copyright (c) 1982 Beam Software

Editado por INDESCOMP, S.A.
Pº de la Castellana, 179 - 28016 MADRID

* Todas las consultas relativas a este libro deberán ser dirigidas
a INDESCOMP, S.A.

Traduce e Imprime: CONORG, S.A.

I.S.B.N.: 84-86176-07-7
Depósito Legal: M-26663-1984

CONTENIDOS

BUSCANDO TU CAMINO EN EL LENGUAJE-MAQUINA

Comenzando por el principio	5
Conceptos básicos sobre el lenguaje-máquina	11
La forma de contar de los computadores	18
De cómo se representa la información	24
Una ojeada sobre la CPU	30
Todo está muy bien, pero ¿Cómo se pasa un programa en lenguaje-máquina?	39
De cómo la CPU usa sus extremidades (Registros)	43
Marcando números a una mano	51
Banderas y sus usos	58
Contando y descontando a una y dos manos	64
Aritmética a una mano	69
Operadores Lógicos	75
Apañándose las con números a dos manos	79
Manipulando números a dos manos	83
Manipulando la pila	91
Aritmética a dos manos	95
Lazos y Saltos	99
Uso de subrutinas en lenguaje-máquina	106
Operaciones con bloques	109

INSTRUCCIONES DEL Z80 QUE SE USAN MENOS FRECUENTEMENTE

Canje de registros	115
Bits: Mirar, poner y reponer	117
Rotaciones y desplazamientos	119
Dentro y fuera	122
Representación BCD	126
Interrupciones	127
Instrucciones de reanudación (RST)	128

PROGRAMANDO TU SPECTRUM

Planteando tu programa en lenguaje-máquina	130
Particularidades del ZX Spectrum	135

PROGRAMA MONITOR

Introduciendo al monitor de rutinas en lenguaje-máquina	145
Cargador Hexadecimal	155

PROGRAMA: LAS RANAS URBANAS

Pasando el Programa	161
Etapa Primera: Bases de datos	164
Etapa Segunda: Inicialización	172
Etapa Tercera: Tráfico regular	176
Etapa Cuarta: Coche de policía	181
Etapa Quinta: Rana	185
Etapa Sexta: Control	190

APENDICES

Tabla de entrada de las teclas del Spectrum	227
Direcciones en la zona de pantalla	228
Repertorio de símbolos del Spectrum	229
Tabla de conversión de Hexadecimal a Decimal	230
Tabla resumen de operaciones con banderines	234
Instrucciones del Microprocesador Z80 por orden alfabético de nemónimos	236
Instrucciones del Microprocesador Z80 ordenados por código de operación	240

COMENZANDO POR EL PRINCIPIO

Este libro está diseñado para introducirte en la programación del

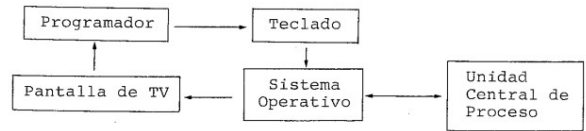
"SINCLAIR ZX SPECTRUM", usando el lenguaje de la máquina
y el lenguaje ensamblador.

Puede ser que abordes en este libro, sin tener ideas precisas de qué va eso
del lenguaje-máquina. No importa.

Puede incluso que ni sepas qué es el lenguaje-máquina.
O que no te hayas percatado que existe una diferencia esencial
entre lenguaje-máquina y lenguaje ensamblador; ni por supuesto,
seas consciente de la enorme diferencia que ambos tienen
con el lenguaje BASIC.

No te importe; ni te preocupes; ni te asustes por la *jerga*:
nosotros te explicaremos cada cosa, todas las cosas, y paso a paso.

Para empezar, observemos la forma en que un computador trabaja:



Con este diagrama intentamos mostrarte que hay una barrera entre el programador
y el cerebrito del SPECTRUM: la unidad central de proceso
que abreviarémos con las siglas CPU, correspondientes a

Central
Processing
Unit

No es posible con un procesamiento normal, que el programador le diga directamente
a la CPU lo que desea hacer. Es necesario que el sistema operativo TRAMITE
tus deseos.

En las máquinas de Sinclair, se ha escogido para la CPU la pastilla Z80A
que es la versión más rápida del ya popular, procesador Z80. Los procesadores más
ampliamente utilizados para microcomputadores son cuatro:

- el Z80,
- el 6502,
- el 6809,
- y el 8088.

Con mucho, la pastilla más popular es la Z80.

*Estoy seguro de que no constituirá ninguna sorpresa si te confirmamos
que el Z80A no comprende ni una sola palabra de ¡BASIC!
Desde luego no se ha diseñado ninguna CPU que pueda comunicarse
directamente con una persona que hable en cristiano.*



Si lo analizas durante bastante tiempo, te darás cuenta que sería muy difícil -si no imposible- dar a la pastilla de un computador una instrucción que tuviera significado para una persona. Quitá la parte superior de tu Sinclair (si te atreves) y échala una mirada a la cucaracha cercana al altavoz...

es la CPU Z80A

Es obvio, que esa cucaracha que hay dentro del computador, sólo puede responder a las señales eléctricas que le envíen el resto de las cucarachas.

¿QUE ES LENGUAJE-MAQUINA?

La pastilla Z80, ha sido diseñada de forma que puede aceptar simultáneamente señales en 8 de las patitas que posee.

Los diseñadores de la cucaracha Z80 la construyeron, de manera que las diferentes combinaciones de esas 8 señales INSTRUYAN al Z80 para llevar a cabo las diferentes funciones que debe conocer.

Teniendo bien claro que lo que realmente sucede es pura electrónica; vamos a adoptar un convenio para representar estas señales, por ejemplo:

-Si hay señal en una de las patitas de la cucaracha escribiremos un 1, y escribiremos un 0 si no hay señal.

Una combinación de las 8 señales la representaremos algo así como 0011 1100 y representará una instrucción para la CPU.

Lo cual, dista mucho de parecerse a eso de:

¡ LET A = A + 1 !

Y sin embargo, esto es de lo que va el llamado lenguaje-máquina. El nombre ya lo dice: es un lenguaje para máquinas.

Cada fabricante de las diferentes pastillas ha empleado un diferente "lenguaje" para su producto.

En este momento puede que te estés preguntando:

Si esto es el lenguaje-máquina: ¿por qué molestarme? ¿por qué no beneficiarme del trabajo ajeno que me permite programar en un lenguaje mucho más fácilmente comprensible, como es el BASIC y el COBOL?

La razón estriba en las ventajas primordiales del lenguaje-máquina, que son:

- UNA EJECUCION MAS RAPIDA DEL PROGRAMA
- UN USO MAS EFICAZ DE LA MEMORIA
- UNOS PROGRAMAS MAS CORTOS (que ocupan menos espacio en memoria)
- UNA ABSOLUTA LIBERTAD FRENTE AL SISTEMA OPERATIVO



Todas estas ganancias son el resultado directo de programar en un lenguaje que la CPU puede entender, sin tener que traducirlo primero.

Cuando tú programas empleando el lenguaje BASIC, el programa que realmente está ejecutando la máquina es el llamado "Sistema operativo", que sí está escrito en lenguaje-máquina ...(por otras personas).

En lenguaje normal, ese programa es algo así como:

OTRA

- Mire la siguiente instrucción a procesar
- Tradúscala en la correspondiente serie de instrucciones en lenguaje-máquina.
- Cumplimente cada instrucción
- Guarde el resultado si así se exige
- Retorne a la instrucción llamada OTRA.

Si estás *elucubrando* sobre dónde el computador encuentra el sistema operativo, no sigas, está en la ROM.

En otras palabras, el sistema operativo está incorporado dentro del SPECTRUM. ROM es la abreviatura de Read Only Memory:

Es aquella zona de la memoria, cuyo contenido no puede cambiarse y en la que SOLO puedes LEER (que como sabes es mirar o "guipar" lo que cada celdilla de memoria contiene).

¡Un programa escrito en BASIC, puede ser hasta 60 veces más lento que un programa escrito directamente en lenguaje-máquina!

La traducción toma su tiempo

(que se lo digan al traductor de este libro).

Y además, las instrucciones en lenguaje-máquina que resultan de la traducción, son normalmente menos eficaces que las escritas directamente en lenguaje-máquina. De la misma forma que es normalmente más rápido conducir uno mismo, que usar el transporte público: Tú puedes atajar por donde sabes, en lugar de seguir la ruta del transporte público que debe ante todo preocuparse del CASO GENERAL de los usuarios.

No obstante, nosotros somos los primeros en admitir que la programación en lenguaje-máquina también tiene sus inconvenientes.

Las principales desventajas del lenguaje-máquina son:

- LOS PROGRAMAS SON DIFÍCILES DE LEER Y DE "DEPURAR"
- ES IMPOSIBLE ADAPTARLOS A OTROS COMPUTADORES
- SON PROGRAMAS MÁS LARGOS (en el número de instrucciones)
- LOS CÁLCULOS ARITMÉTICOS SON DIFÍCILES

Esto significa que TÚ debes tomar una decisión muy ponderada sobre el método de programación que vas a utilizar en cada caso particular:

-Un programa muy largo para una aplicación de contabilidad, deberá escribirse en un lenguaje preparado para tratar con números y en el que los programas puedan ser fácilmente modificados cuando así se precise.



Por otro lado, no hay nada tan frustrante como un juego de marcanitos, o de saltos de rana, escrito en BASIC -cuando lo has logrado completar, resulta que es increíblemente lento.

Son tus propias necesidades, la cantidad de memoria que haya en tu computador, el tiempo de respuesta que precises, el tiempo de que dispongas para desarrollar el programa, y otras circunstancias las que determinarán la elección del lenguaje de programación.

En resumen, el lenguaje-máquina, es una serie de señales eléctricas que la unidad central de proceso puede comprender y que nosotros podemos representar por números.

¿QUE ES LENGUAJE ENSAMBLADOR?

Es bastante comprensible, que si el lenguaje-máquina únicamente fuera representado por números, muy poca gente sería capaz de escribir programas en ese lenguaje. Porque después de todo, quién puede sacar partido de un programa que pareciera algo así como:

0 0 1 0 0 0 0 1
0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
etc...



Afortunadamente, podemos inventar una serie de nombres para cada uno de estos grupos de unos y ceros. El lenguaje ensamblador es la representación de este lenguaje-máquina de manera que sea más legible para los humanos; sólo hay una diferencia muy pequeña entre el lenguaje ensamblador y el lenguaje-máquina:

-El lenguaje ensamblador es de un nivel inmediatamente superior al lenguaje máquina. Los humanos lo podemos leer más fácilmente que el lenguaje-máquina; los computadores no pueden leer lenguaje ensamblador. Por cada instrucción escrita en lenguaje ensamblador hay una instrucción que cumple la misma función escrita en lenguaje-máquina, y viceversa. Es decir, hay una relación BIUNIVOCAL entre ellas, podemos decir que el lenguaje ensamblador es equivalente al lenguaje-máquina.

El lenguaje ensamblador utiliza nuestra capacidad *nemotécnica* para fomentar la legibilidad de un programa.

Por ejemplo, a estas alturas, la instrucción

INC HL

puede que no signifique mucho para tí, pero por lo menos puedes leerlo, son letras. Si además se te dijera que "INC" es la abreviatura acordada para "incrementar" y que "HL" es el nombre de una variable; observando esa instrucción, seguro que tendrías el *pálpito* de lo que realiza:

Aumenta el valor de HL.

Esa misma instrucción en lenguaje-máquina es

0 0 1 0 0 0 1 1

y obviamente, también puedes "leer" esa instrucción, porque en cierta manera puedes leer los números, pero no va a significar absolutamente nada para tí, a menos que tengas una tabla donde puedas consultar su equivalente, o que tu cerebro tenga una capacidad retentiva mayor que la de un computador.



El lenguaje ensamblador puede ser traducido directamente al lenguaje-máquina por tí (consultando una tabla), o por un programa (consultando una tabla).

Un programa así, se llama ensamblador. Puedes imaginártelo como un programa que realiza la tediosa tarea de traducir un programa en lenguaje ensamblador a la correspondiente secuencia de instrucciones en lenguaje-máquina que el cerebrito del SPECTRUM comprenderá.

Y por supuesto, comprendemos que un ENSAMBLADOR para el ZX Spectrum está disponible en el mercado. Sin embargo, esos ensambladores exigen normalmente 6K de memoria, por lo que en una máquina con 16K, no tiene mucha utilidad. La pantalla del Spectrum ocupa 7K de memoria, y después de cargar el ensamblador puedes encontrarte que solamente quedan libres unas 4K de memoria para el programa que tú deseas ensamblar (lo que significará aproximadamente, 1/2K de programa en lenguaje-máquina),

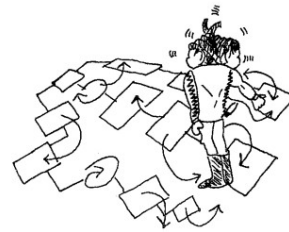
La alternativa que se te ofrece frente a un programa ensamblador es que tú realices la traducción de los nemónimos del lenguaje ensamblador a los códigos del lenguaje-máquina, utilizando las tablas que *te prestamos* al final de este libro.

Es duro, es frustrante al principio. Está lleno de inconvenientes. Pero es una práctica maravillosa para tu desarrollo ! y te dará una magnífica comprensión de cómo la CPU del Spectrum funciona.

Te recomendaríamos, de hecho, que intentarás escribir programas cortos en lenguaje-máquina de esta forma:

Escribiéndolos en lenguaje ensamblador y traduciendo los
a lenguaje-máquina, a mano.

Antes de que te comparas un programa ensamblador.



*En inglés nemonics

En español, como tú quieras; pero ten en cuenta que el *nemimos* se refiere a la memoria como en *memotécnico* y "nimos" a nombres como homónimo.



RESUMEN

CPU

Es la abreviatura para la *Unidad Central de Proceso* del computador. Es la pastilla electrónica que efectúa todas las tareas de control y de cálculo de un computador.

LENGUAJE-MAQUINA

El lenguaje que *entiende* la CPU. Para la CPU del Spectrum, es el lenguaje máquina correspondiente al microprocesador Z80, y que está constituido por unas 200 instrucciones.

LENGUAJE BASIC

Es un lenguaje de programación diseñado para que sea comprensible por los humanos. Cuando un computador obedece un *comando* en BASIC, necesita previamente *traducir* este comando en una serie de instrucciones en *lenguaje-máquina*. Los programas en BASIC son, por tanto, considerablemente más lentos que los programas en lenguaje-máquina.

...Pero son más fáciles de redactar.

LENGUAJE ENSAMBLADOR

Cada *instrucción* en lenguaje-máquina tiene una *representación taquigráfica y nemotécnica* para que pueda ser más fácilmente comprendida por las personas. Por ejemplo, la instrucción en lenguaje-máquina 0 0 1 1 0 1 1 0 tiene su equivalente en lenguaje ensamblador con "HALT" (que indica a la máquina una DETENCIÓN temporal en el programa).

PROGRAMA ENSAMBLADOR

El programa que *traduce* las instrucciones escritas en lenguaje ensamblador (fácilmente leídas y comprendidas por las *personas*) en instrucciones en lenguaje-máquina (facilísimamente leídas y comprendidas por el *computador*).

ROM (Memoria en la que sólo se puede leer).

Zona de la *memoria* de un computador donde está alojado el extenso programa escrito en lenguaje-máquina que *determina* la operación del computador. En la terminología inglesa no se le denomina software ni hardware sino *firmware*: porque está FIRMEMENTE incorporado a los circuitos del computador, de manera que permanece allí incluso cuando apagamos el computador y nos vamos.

Para el Spectrum, la ROM está en el lenguaje-máquina del Z80, y la grabaron específicamente con él. La ROM del Spectrum ocupa desde las posiciones de memoria que van de la 0 a la 16383. *Únicamente* puedes *guiar* el contenido de esas posiciones; mientras que en el resto de la memoria RAM puedes *mirar* lo que contiene cualquiera de las posiciones y *meter* (o escribir) lo que desees.



¿Qué es la CPU?

Si deseamos comunicarnos con el computador, tenemos que saber la clase de órdenes que acepta y el lenguaje que el *cerebrillo* de la máquina (la CPU) habla.

Si ignoramos la clase de información que la CPU comprende, no podemos realmente instruir al computador para que efectúe tareas tan singulares como la de ser nuestro contrario al ajedrez o ser un contable que cuide nuestras cuentas. La CPU no es un enorme misterio. A mí me gusta imaginarla como una *compañera* solitaria, colocada en el interior de tu Spectrum, y a la que se le pide constantemente que esté haciendo cosas.

Sobre todo cálculos.

Pero la pobre "compa" no posee ni un trozo de papel ni un lápiz para anotar lo que está pasando. ¿Cómo lo hace?

El diseño de la CPU:

A estas alturas, yo debo contarte la forma en que los diseñadores del Z80 ven las cosas, y cómo se supone que la CPU debe tratarlas. La CPU ha sido diseñada para efectuar labores extremadamente simples, pero está capacitada para hacerlas muy rápidamente.

Hemos mencionado antes, que la CPU no tiene incluso ni lápiz ni papel, y eso es una parte del diseño de la CPU. Cualquier número que ella (no pueda recordar) tiene que llevarlo a un "cajetín" para guardarlo de forma segura.

Vamos a mirar un ejemplo:

Digamos que tú quieres que la CPU determine la hora en Nueva York, sabiendo la hora que es en Londres.

Dado que la CPU no conoce absolutamente nada, lo primero de todo: tendremos que decirle cual es la hora en Londres (digamos las 10 de la mañana).

La CPU no tiene sitio para guardar esta información, y no sabe lo que le vamos a preguntar a continuación, por lo que mete esa información en un cajetín; digamos el cajetín número 1.

Luego tendremos que decirle la diferencia de hora entre Nueva York y Londres; digamos 5 horas de antelación; y ella lo pondrá en el cajetín número 2.

Ahora viene la ocasión para calcular, ella mira el cajetín número 1, recaba el número que allí hay, recaba el número del cajetín número 2, y realiza el cálculo; luego pone el resultado, digamos en el cajetín número 3.

10 - 5 = 5 el resultado por supuesto es "las 5 de la mañana".



Todas estas carreras entre cajetines, sumas, restas y demás, serían extremadamente tediosas si la CPU tuviera que hacerlo todo en *su cabeza*; por lo que exactamente hace lo que tú o yo haríamos: contar con nuestros dedos (de la mano y de los pies si es preciso). Las *manos* y los *pies* de la CPU se denominan REGISTROS. La pastilla Z80 de tu Spectrum es singular en que tiene bastantes "manotas" y "pinreles", pero ya llegaremos a eso más adelante.

Para ilustrar cómo la CPU calcula la diferencia de tiempo en el ejercicio anterior, llamemos a una de las manos de la CPU "MANO A".

¿Cómo manipula la CPU el contenido del cajetín número 1 y del cajetín número 2?

La secuencia que te presentamos ahora es casi exactamente lo que la CPU realmente haría:

- Marcar el valor del cajetín número 1 con los dedos de su MANO A.
- Restar el contenido del cajetín 2 de lo que tiene en su MANO A.
- Mirar los dedos de su MANO A y guardar el valor que indican en el cajetín 3.

De esto, podemos deducir algunas conclusiones fenomenales:

1. La CPU no sería capaz de trabajar con un número como 11,53 —únicamente podría tratar con números ENTEROS.
2. La CPU estaría limitada en sus cálculos al máximo número que pudiera marcar con sus dedos.

Eso es verdad

Nuestro principal consuelo es, que la CPU tiene un montón de *manos y pies*, y además puede manejar cada uno de ellos separadamente, y puede contar hasta 255 usando sólo los 8 *dedos* de la MANO A.

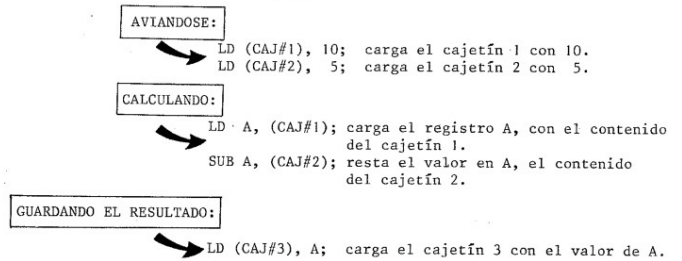
Ya hablaremos en el siguiente capítulo de los detalles de cómo la CPU puede contar hasta más de 8 con cada mano, mientras nosotros sólo logramos contar hasta 10, usando 2 manos. Baste decir por ahora que

con cada mano puede contar hasta 255 y que
con cada pie puede contar hasta más de 64.000!!

El ejercicio de Nueva York - Londres no ha sido escrito todavía en un lenguaje que la CPU comprenda —lo que hemos hecho es describir el **proceso**—.



Para que tengas un anticipo del mundo excitante de la programación en lenguaje-máquina, usemos ahora nemónimos (siglas o abreviaturas), para enseñar a la CPU cada paso:



Estas instrucciones pueden parecer al principio *demasiado lacónicas*, pero después de todo los nemónimos son los nemónimos.

"LD" es la abreviatura de LOAD, de manera que LD A, 1, por ejemplo significaría *cargar* el registro A con un 1: es decir, marcar un 1 con los dedos de la MANO A.

También usamos una imagen bastante gráfica en estos nemónimos mediante los paréntesis:

★ LOS PARENTESIS se usan para indicar que tratamos con el contenido de lo que se mencione dentro del paréntesis.

Es fácil recordar ésto, porque los paréntesis son similares a CONTENEDORES.

Así al seguir los nemónimos anteriores, mandamos a la CPU llevar a los cajetines número 1 y número 2, el 10 y el 5 respectivamente, etc... para obtener el resultado final en el cajetín número 3.

Todo ésto es bastante simple de seguir, y estoy seguro que puedes comprender que mientras estamos haciendo este cálculo, los números de la mano "A" se utilizan para representar la hora en Nueva York; que posteriormente pueden utilizarse para representar el número de empleados de una compañía y que en otro instante cuánto dinero tienes, etc.

Si tú estás acostumbrado al concepto de variables por tus programas en BASIC, debes dejarlos a un lado cuando programes en lenguaje-máquina.

Los dedos de la mano "A" no son una variable en el sentido que se le da en la programación en BASIC. Son simplemente lo que la CPU utiliza para contar. Una de las grandes diferencias al programar en lenguaje-máquina frente a la programación en BASIC es precisamente esta falta de variables.



Puedes haberte dado cuenta que es factible asimilar los cajetines que usamos para guardar información a variables en BASIC, si damos a cada uno un nombre.

Sí. Tienes toda la razón; aunque tampoco son realmente variables, los cajetines pueden ser inmensamente útiles y realizan funciones similares a las de las variables, pero ten bien presente que estos cajetines no son más que celdillas de memoria que reservamos para fines específicos.

La forma en que la CPU se las apaña con los números negativos es diferente, y lo comentaremos más adelante.

¿Y cuando se le acaban las manos a la CPU?

Ahora yo mencionaría que si te encontraras con la CPU en la calle, observarías que es una "tía" con aspecto muy extraño:

*Sus manos tienen 8 dedos, y tiene 8 manos.
Sólo tiene dos pies, pero cada pie tiene 16 dedos.
Y encima es extremadamente ágil con sus pies!*

Esta, por tanto, bien pertrechada para el enorme número de cálculos que le exigimos, tomando nota de todos los números con los dedos de sus manos y de sus pies.

Sin embargo, puede suceder que la CPU no tenga suficientes manos para hacer los cálculos, o que el programador por alguna razón desee parar a la mitad de un cálculo para hacer cualquier otra cosa.

La CPU no puede simplemente guardar la información en cajetines, porque tendría que mantener algunas manos libres justamente para recordar en qué cajetines puso la información.

La CPU Z80 lo soluciona utilizando una PILA, que es como una de esas "agujas" verticales que alguna gente tiene en su mesa de despacho para pinchar las cuentas, papелitos de notas, etc.; primero uno, luego otro encima, y así sucesivamente. Es un excelente sistema de archivo si luego sacas únicamente el papel que está encima de todos; pero es muy inconveniente si quieres sacar el de enmedio, ya que tendrías que barajar todos los papeles de la pila.

Afortunadamente, es un sistema muy conveniente para la CPU porque ella siempre necesita coger el papel que está en la cima de la pila. Cuando quiera que una interrupción obliga a la CPU a parar sus cálculos, ella pone toda la información de sus manos en la pila, y tan pronto como puede reanudar los cálculos, ella saca las piezas de información de la parte superior y continúa con su trabajo.



En la terminología de los computadores, se llama a esta aguja una "pila" (en inglés, *stack*). Cuando ponemos un trozo de información en la pila, lo "empujamos" sobre ella (*push*). Y cuando lo sacamos de ella, "tiramos" de él (*pop* -como si fuera el corcho de una botella-).

La CPU puede "pushar" y "popar" toda clase de información en la pila -por ejemplo, en medio de un cálculo complejo la CPU puede desear "salvar" toda la información de sus muchas manos y pies, lo que implicará muchos "pushados" diferentes. Para recabar la información se necesitarán tantos "popados" equivalentes.

Por razones que mejor conocen los diseñadores del Z80, a nuestra CPU le gusta mantener la pila llena hasta el tope. Lo que significa que cuanta más información se coloca en la pila, más crece ésta HACIA ABAJO.

La principal ventaja de utilizar una pila para guardar temporalmente trozos de información es que la CPU no necesita recordar en qué cajetín está ésta, sabe que es la última pieza de información que colocó en la pila. Naturalmente ha de mantener un orden estricto si ha de meter y sacar de la pila muchos de estos trozos.

¿QUE PUEDE HACER LA CPU?

Yo creo que es conveniente considerar en este momento la clase de instrucciones que los diseñadores pensaron que debían incorporar en la pastilla Z80.

Dado que la CPU tiene que ser capaz de recordar todos sus cálculos mediante los dedos de sus manos y pies, sólo hay dos clases de números que la CPU puede tratar:

- -Números a una mano -i.e.: números que pueda señalar con una sola mano;
- -Números a dos manos -i.e.: aquellos números que pueda presentar con dos manos.

Te será difícil de creer, pero la CPU no puede manejar números mayores que los que puede indicar con 2 manos!

Las instrucciones que la CPU puede cumplimentar son también muy limitadas:

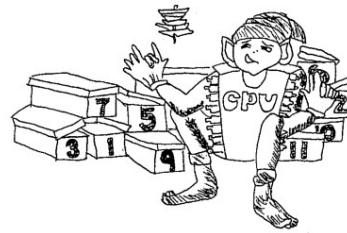
- -Contar números a una mano.
- -Contar números a dos manos.
- -Sumar, restar, aumentar, disminuir, o comparar números a una mano.
- -Sumar, restar, aumentar o disminuir números a dos manos.
- -Varias manipulaciones con números a una mano -e.g.: hallar el negativo de un número.
- -Hacer que la CPU salte a otra parte del programa.
- -Intentar transferir números a una mano desde y hasta el mundo exterior.



Ya sé que pensarás que es un conjunto muy limitado de instrucciones, y a pesar de ello, puedes hacer que la CPU juegue al ajedrez o calcule tu salario!

Observa que hay instrucciones muy simples como la multiplicación, pero que no existen. Si necesitas multiplicar dos números cuando estás programando en lenguaje máquina tendrás que escribir un programa para que pueda hacerse.

Ese es el *meollo* de la diferencia entre lenguaje-máquina y lenguaje BASIC:
el lenguaje-máquina es mucho más lento para redactar programas, porque únicamente puedes hacer las cosas en pasos diminutos.



RESUMEN

REGISTROS

La CPU tiene un cierto número de registros que usa para cálculos. Ocho de ellos pueden verse como las *manos* de la CPU, y 2 de ellos como los *pies* de la CPU. Cada mano tiene 8 *dedos*, mientras que cada pie tiene 16 *dedos*.

CELDILLAS DE MEMORIA

La CPU puede transferir información de una mano a otra, de una mano a la memoria, y de la memoria a una mano. Pueden *reservarse* celdillas determinadas de memoria para que contengan una información específica.

LA PILA

La CPU puede usar una pila para guardar la información que el programador desea. Se coloca información en la pila "empujándola sobre ella", y se recaba la información de la pila "tirando de ella".

INSTRUCCIONES POSIBLES

Las instrucciones que la CPU es capaz de realizar, son las más simples, efectúan transferencias de información y cálculos aritméticos. Todo programa ha de estar constituido por series de estas instrucciones simples.



LA FORMA DE CONTAR DE LOS COMPUTADORES

Hemos dicho que la CPU era capaz de contar hasta 255, empleando sólo 8 dedos.

¿Cómo puede ser eso, si con 10 dedos nosotros sólo podemos contar hasta 10?


Ciertamente, no porque sean más activos (que no lo son), sino porque están más mecanizados en su formación:


¿Por qué al levantar nuestro dedo índice indicamos el mismo valor (= '1') que al levantar nuestro dedo meñique?.


Parece obvio que si quisiéramos, podríamos representar dos números diferentes de esta forma, lo que es muy parecido al caso de considerar distintos el número 001 y el número 100.


La pura verdad es que *los humanos* no somos nada eficaces usando los dedos para contar. La CPU entiende que al no tener un dedo levantado ya posee alguna información, y que cuando el dedo está levantado es una diferente información.

Con sólo dos dedos, es posible ingeniárselas para contar desde 0 a 3 de la manera siguiente:

 00 = 0 Si indicamos como 0 no tener ningún dedo levantado,

 01 = 1 y si tener un dedo levantado significa '1'.

 10 = 2 Esto significa uno-cero, pero no 10

 11 = 3 Significa que elegimos la representación 11 (2 dedos) para el valor 3.

De esta fácil manera, hemos elegido una representación diferente.

Hay una estrecha relación entre esto que hemos visto y la representación binaria. Los dedos de la CPU son como celdillas en la memoria que pueden indicar *alzado* y *bajado* (o '1' y '0' como estarás acostumbrado a ver).

Si añadimos un tercer dedo en nuestro ejemplo podríamos representar todos los números desde el 0 hasta el 7.

Con cuatro dedos sería posible representar todos los números del 0 al 15.

Si no te lo crees sería un buen ejercicio que escribieras todas las posibilidades correspondientes a cuatro dedos.



Con el fin de simplificar la notación de tales números, y evitar la confusión al escribir "11", queriendo significar algo distinto a que 2 bits están alzados; se ha adoptado universalmente el siguiente convenio:

Se indican con las letras de la A a la F los números del 10 al 15.

Esto significa que escribiremos los números decimales del 0 al 15 como:

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

¿A que es simple?

Esta forma de numerar se denomina FORMATO HEXADECIMAL.

Para evitar confusiones, algunos escriben una "H" después de un número hexadecimal v.e.: 10H). La "H" no tiene valor, y sólo sirve para recordarnos que está en formato hexadecimal.

En la programación en lenguaje-máquina, es MUY CONVENIENTE tratar los números en formato hexadecimal;

Pero esto sólo es un convenio y si lo desearas podrías escribir todas tus instrucciones en el formato decimal normal. Pero mejor utiliza el formato hexadecimal porque:

1. Es facilísimo pasar de esta forma a la forma binaria, que nos indica lo que está haciendo cada bit (dedo).
2. Nos proporciona un medio sencillo de ver si los números son a una o a dos manos -ie.: 8 bits o 16 bits.
3. Establece como estándar que todos los números son grupos de dos dígitos (los comentaremos más ampliamente).
4. Este convenio universal y la familiaridad con el sistema hexadecimal, te permitirá leer otros libros y manuales.
5. Como la CPU está preparada para procesar información representada por números binarios, lo que es fastidioso para su lectura por humanos, necesitamos una representación más asequible.

Pero sigue siendo sólo un convenio y no una regla sagrada.

El sistema hexadecimal, como ya dijimos, permite representar los números del 0 al 15 usando sólo 4 bits. Cualquier celdilla de la memoria con 8 bits, o un registro de 8 bits puede describirse por dos grupos de 4 bits.




(Es lo mismo que decir que cualquier combinación de 10 dedos puede representarse con 2 manos de 5 dedos cada una ¿verdad?).

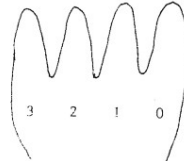
LA RAZON POR LA QUE INSISTIMOS CON LAS CELDILLAS DE MEMORIA DE 8 BITS Y LOS REGISTROS DE 8 BITS, ES QUE CORRESPONDE A LA ESTRUCTURA DEL ZX SPECTRUM.

Todas las celdillas de memoria y todos los registros simples tienen 8 bits, lo cual no es difícil de comprender! (todos los humanos tienen 5 dedos en cada mano).

Tomémonos las cosas, cada una a su tiempo, y vamos a "familiarizarnos" primero con 4 dedos:


$$\begin{aligned} 1111 &= 2^{**3} + 2^{**2} + 2^{**1} + 2^{**0} \\ &= 8 + 4 + 2 + 1 \\ &= \text{Decimal } 15 \\ &= F \text{ (en notación hexadecimal)} \end{aligned}$$

Para aquellos de vosotros con inclinaciones matemáticas, podéis observar que el número que cada dedo representa está multiplicado por 2 a medida que vais hacia la izquierda. Si nosotros numeramos los dedos así:



entonces el valor de cada dedo es "2ⁿ", siendo n el número asociado al dedo. Llamemos a esta mano de 4 dedos "manilla", de la misma forma que un cigarro pequeño es un ...!"pitillo"!

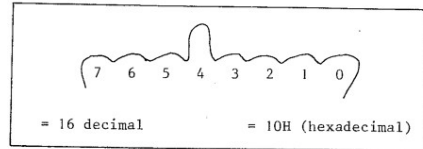
Ejercicio: ¿Cuál es el valor decimal y hexadecimal que los siguientes grupos de 4 bits representan?

	<u>Decimal</u>	<u>Hexadecimal</u>
0010		
0110		
1001		
1010		
1100		

Es importante que te familiarices con la notación hexadecimal, y si todavía no has agarrado el concepto vuelve a leer las últimas páginas, antes de proseguir.

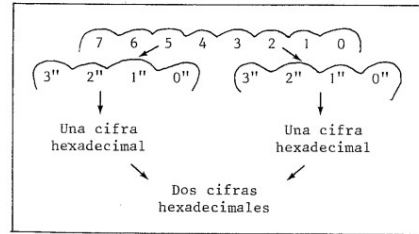


Examinemos lo que sucede si queremos un número mayor de 15. Digamos 16.
 Pues usaremos el siguiente dedo de la izquierda porque nuestras manos
 tenían 8 dedos, o sea, dos *manillas*.



Podemos, por tanto, indicar cada manilla por una de las cifras hexadecimales que representan del 0 al 15, (0-9 & A-F).

De esta forma cualquier *mano de 8 bits* puede escribirse exactamente como *dos manillas hexadecimales*:



La manilla de la izquierda indica 16 veces más que la manilla de la derecha. Es bastante equivalente a la notación decimal: la cifra de las decenas vale diez veces más que la cifra de las unidades.

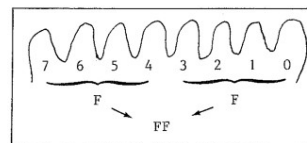
Un número en formato decimal como 15, lo convertimos tan automáticamente a $15 = (1 \times 10) + 5$ que no prestamos la más mínima atención.

Es exactamente lo mismo con notación hexadecimal. Para convertir de hexadecimal a decimal, multiplicamos la cifra hexadecimal de la manilla izquierda por 16 y le sumamos la cifra de la manilla derecha. Con el ejemplo anterior:

$$\begin{aligned} 10H &= (1 \times 16) + 0 \\ &= 16 \text{ Decimal} \end{aligned}$$



Así es como somos capaces de contar hasta 255 utilizando sólo 8 dedos. El número máximo lo obtenemos cuando están levantados todos los dedos:



$$\begin{aligned} FFH &= (F \times 16) + F \\ &= (15 \times 16) + 15 \text{ (en decimal)} \\ &= 255 \text{ (Decimal)} \end{aligned}$$

El número más pequeño es cuando no tenemos levantado ningún dedo:

$$00H = 0 \text{ Decimal}$$

- Observa que todos los números, hasta el mayor, requieren 2 -y sólo 2- cifras para definir el número.

Investiga por tí mismo cualquier combinación de 8 bits y mira a ver si puedes convertirlo a notación hexadecimal y luego a decimal. Al principio puede parecer un poco extraño y difícil pero pronto *le echarás el guante*.

Además, contar en hexadecimal es hacer lo mismo que en decimal:

Decimal: 26 27 28 29 30 etc.

Hexadecimal: 26 27 28 29 2A 2B 2C 2D 2E 2F 30 etc.

Los valores de los números en decimal y en hexadecimal, corresponden a diferentes cantidades.

Te habrás dado cuenta que después de 29H viene 2AH y no 30H.



El siguiente programa en BASIC te permitirá teclear en tu Spectrum un número decimal y convertirlo a su valor en hexadecimal:

```
100 REM conversión decimal a decimohexal
110 PRINT "teclea valor en decimal"
120 INPUT n: PRINT n
130 LET S$ = ""
140 LET n2 = INT (n/16): LET n1 = INT (n - n2*16)
150 LET S$ = CHR$( (n1 <= 9) * (n1 + 48) +
    (n1 > 9) * (55 + n1)) + S$
160 IF n2 = 0 THEN PRINT: PRINT "hexadecimal = "; S$;
    " H":FOR I = 1 TO 200: NEXT I: RUN
170 LET n = n2: GO TO 140
```

Intenta *convertir a mano* los siguientes números y utiliza el programa en BASIC para comprobar tu respuesta.

- i. 16484 la dirección de la celdilla de memoria en que comienza la zona de pantalla.
- ii. 22528 la dirección de la celdilla de memoria en que comienza la zona de atributos de la pantalla del Spectrum.
- iii. 15360 la dirección de la celdilla en la memoria en que comienza el repertorio de símbolos del Spectrum.
- iv. 15616 dirección de la celdilla de memoria en que comienzan la tabla de códigos ASCII en el Spectrum.

RESUMEN

DECIMAL

La notación decimal es un convenio para contar en grupos de 10 unidades cada vez. Las cifras decimales son 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9.

HEXADECIMAL

La notación hexadecimal es un convenio para contar en grupos de 16 unidades cada vez. Las cifras hexadecimales son 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E y F.

Algunas veces se añade una H al final de un valor hexadecimal para recordarnos que está escrito en ese formato. Por ejemplo: 1800H.

CELDILLAS DE MEMORIAS DE 8 BITS

El ZX Spectrum está diseñado para que cada celdilla de memoria contenga 8 bits (dedos). Cada celdilla de memoria puede guardar desde el 0 hasta 255 en decimal. Es más conveniente representarlo en la notación hexadecimal como un número de 2 cifras.



DE COMO SE REPRESENTA LA INFORMACION

Hay una diferencia enorme entre *humanos y computadores* para representar información. La información para los humanos está primordialmente compuesta de signos, cifras y letras (información alfanumérica), mientras que toda la información en un computador está almacenada como grupos de bits.

La palabra *bit* corresponde a *b*inario *d*igito ("0" ó "1"). En el procesador Z80A, estos bits están agrupados en series de 8. A un grupo de 8 bits lo llamamos... octeto (*byte si eres anglofilo*). A esta forma de representar información usando bits se denomina formato binario. Forma la base del lenguaje que el Z80 y la mayoría de los microprocesadores hablan.

Básicamente, dentro del Spectrum hay dos clases de información: la primera es el programa, la segunda son los datos sobre los que el programa opera y que pueden ser números o textos alfanuméricos. Comentaremos ahora estas tres representaciones: PROGRAMA, NUMEROS y ALFANUMERICOS.

REPRESENTACION DE UN PROGRAMA

Un PROGRAMA es una secuencia de instrucciones que damos a la CPU para que efectúe una determinada tarea, que puede *desglosarse* en un cierto número de "subtareas".

En el Z80 todas las instrucciones se representan internamente, por uno o varios octetos. Lógicamente a las instrucciones representadas por un octeto las llamamos instrucciones "*cortas*". Y las instrucciones *largas* son las representadas por dos o más octetos.

Como el Z80 es un microprocesador de 8 bits, solamente puede operar con un octeto a la vez. Y si requiere más de uno *los recaba sucesivamente* de la memoria. Por tanto, una instrucción de un sólo octeto será cumplimentada generalmente mucho más rápidamente que una instrucción de 2 ó 3 octetos. Como regla general, es más productivo escribir un programa en lenguaje-máquina utilizando instrucciones de un sólo octeto, cuando sea posible.

Puedes mirar ahora el repertorio de instrucciones que hay en el Apéndice y echar una ojeada sobre las instrucciones CORTAS y LARGAS. No te inquietes si no las puedes entender; ya las comentaremos en profundidad más adelante.



Representación de enteros

Como comentamos anteriormente, dado el diseño del Z80, no podemos tener un número como 11,53. La CPU puede únicamente trabajar con números enteros. Además, al usar sólo 8 dedos (8 bits), sólo representaremos los números en la escala de 0 a 255.

v.g.: el decimal "255" se representará por FFH
(cuya representación binaria es 1 1 1 1 1 1 1 1).

Pero, ¿qué pasa con los números negativos?

Representación de números con signo

Recuerda que un octeto es una mano con 8 dedos y que un número se representa según los dedos que estén alzados.

Obviamente, para representar un número entero con signo en formato binario, tendremos que buscar algún convenio para distinguir los positivos de los negativos. Digamos que adoptamos el siguiente:

★ SE CONSIDERARA UN NUMERO NEGATIVO CUANDO LA CPU TENGA SU PULGAR LEVANTADO.
(En la jerga de los computadores, el pulgar es el dedo colocado en el extremo izquierdo de la mano y se denomina bit 7).

Nos quedan solamente 7 dedos para representar el valor del número; lo que significa que el número máximo que podemos usar ya no es 255. De hecho, la mitad de los números que podemos tener en una sola mano (1 octeto) serán negativos y la otra mitad de ellos serán positivos [todo depende de si el pulgar está alzado o no].

La gama total de números posibles en una mano si adoptamos el convenio anterior será, por tanto, de -128 a +127. (Observa que la gama total de números sigue siendo todavía 256).

Y ahora, ya está liado el pastel: ¿Cuándo una mano con el pulgar levantado representa un entero positivo lo suficientemente grande, y cuándo corresponde a un número negativo?

La respuesta es: cuando te de la gana. Tú tienes que hacer la elección: el número puede corresponder a los positivos entre 0 y 255, o a los enteros positivos y negativos entre -128 y +127, pero no las dos cosas a la vez. A tí, programador, te toca decidir el convenio que estás utilizando en un momento dado.



Ya hemos decidido que *alzar el pulgar* significa que el número es negativo, y tenerlo *bajado* es positivo. (Pero, ¿nos basta con esto?).

Pues no. Necesitamos decidir cuál de las 127 posibilidades que tenemos con los restantes 7 dedos corresponde a -1, cuál a -2, etc...

Necesitamos una representación para los números negativos, y de forma que cuando sumemos un número a su opuesto obtengamos 0. Como un ejercicio, pensemos cuál es el número que sumado a 1 nos da 0.

Usted perdona: obviamente es -1, y ya sabemos que el pulgar -bit 7- ha de estar alzado. Por tanto,

0000	0001
1???	????
0000	0000

¿cuál será?

Vamos a intentar con 1 0 0 0 0 0 0 1 -en otras palabras, lo mismo que +1, pero con pulgar alzado. Para comprobar si es -1 *intentemos sumar* este valor a +1.

0000	0001
1000	0001
1000	0010

La suma no es la respuesta correcta. Hubiera sido correcta si la respuesta fuera 0 0 0 0 0 0 0 0. Necesitamos buscar un número que a partir del 1 *que nos llevamos* en la primera columna, se logre que todos los colocados a la izquierda se conviertan en 0.

Puedes intentarlo por tí mismo, pero verás que el único número que cumple esas condiciones es 1 1 1 1 1 1 1 1 (FFH en hexadecimal).

Para confirmarlo, sumemos:

0000	0001
1111	1111
0000	0000

y nos llevamos
"una" a....

¿Habría una forma para determinar la regla general que nos permita hallar el negativo de cualquier número? Del ejemplo parece deducirse que tenemos que hallar el opuesto de un número y añadir un 1 al final.

Intentemos comprobar esta regla con otro número; por ejemplo 3:

es decir 3 =	0 0 0 0	0 0 1 1
opuesto =	1 1 1 1	1 1 0 0
añadir 1 =	1 1 1 1	1 1 0 1

(FDH)

Sumémosle el número obtenido al 3, y veamos lo que pasa:

y llevamos una

0	0	0	0	0	0	1	1
1	1	1	1	1	1	0	1
<hr/>							
-	0	0	0	0	0	0	0

funciona!

Hemos encontrado una manera de representar números negativos

-01 = FF
-02 = FE
-03 = FD
...y así sucesivamente.

El mayor número positivo es:

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

= 7F = 127 decimal

y el mayor negativo será:

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

= 81 = -127 decimal

La comprobación real de la regla es ver si aplicándosela a un número negativo volvemos a obtener el positivo opuesto. Intentémoslo con -3, que como resultó anteriormente es FDH

Número	1	1	1	1	1	0	1
Opuesto	0	0	0	0	0	1	0
Suma 1	=	0	0	0	0	0	1

= 3 y sigue funcionando

Es por tanto, una representación que funciona correctamente, y la podemos aplicar para obtener el negativo de cualquier número.

Números negativos con 16 bits.

El mismo razonamiento se aplica a los números a 2 manos (de 16 bits), y en los que sólo necesitemos alzar el pulgar de una sola mano para indicar si el número es negativo o no. Por convenio se utiliza el bit 7 del octeto de mayor peso.

Convenio

Hablando en términos informáticos diremos que es una representación en COMPLEMENTO A DOSES. En uno de los apéndices de este libro encontrarás las tablas de complemento a doses de los números negativos, (también se dice *complemento a dos*).

Pero recuerda que sólo es un convenio; eres tú quien todavía tiene que decidir en cada momento si los números que estás usando quieren representar números en la gama 0 a 255, o números en la gama -128 a +127.



Ejercicio

- i. Si 127 (0 1 1 1 1 1 1 1) es el mayor número positivo que puede ser representado con este convenio: ¿cómo representarías -128?
- ii. Encuentra el número positivo de mayor valor usando 16 bits (2 manos u octetos) y el mayor número negativo de 16 bits.
- iii. Encuentra el complemento a dos del número más pequeño de 16 bits. ¿Por qué es 8000H?

Representación de datos alfanuméricos

Algunas veces en lenguaje-máquina no queremos que los números sean instrucciones para el procesador, ni que sean números para cálculos; podemos simplemente querer que representen cifras o letras, v.g.: el título de tu último programa "el programa n^o. 1 del mundo".

Nuestro convenio para representar datos alfanuméricos, es decir: *listras de símbolos*, es bastante directo: todos los símbolos pueden representarse con una sola mano (en un código de 8 bits).

● En el mundo de los ordenadores hay dos convenios internacionales para representar datos alfanuméricos: el código ASCII y el código EBCDIC:

★ ASCII corresponde a *Americano Estándar Código Información Intercambio* y se utiliza universalmente en la industria de microcomputadores.

★ EBCDIC es una variación de ASCII utilizada por IBM que corresponde a *Extendido Binario Codificado Decimal Intercambio Código*.

En el ZX Spectrum, los símbolos alfanuméricos corresponden al convenio ASCII, excepto para el símbolo de *la libra* 61H y el de *arrobás* (7FH). En el apéndice correspondiente hay una tabla de conversión del código ASCII. Compárala con la tabla del apéndice A de tu manual del Spectrum.

Tecllea esto: PRINT CHR\$ 33
y aparecerá "!" porque el signo de admiración se representa internamente por 21H (que es decimal 33).

Para tu ayuda:

Te hemos mostrado que la mano de la CPU puede mostrar diferentes cosas:

Puede ser

- una instrucción para la CPU
- un número entre 0 y 255.
- un número entre -128 y +127
- una parte de un número a 2 manos
- un símbolo alfanumérico.

Todo eso es cierto, y depende de tí -el programador- recordar lo que se supone que la CPU tiene en su mano.



RESUMEN

CONTENIDO DE LA MEMORIA

La memoria del Spectrum puede guardar programas, números y textos; tal y como deseamos. No hay forma de decir *cuál es cuál* examinando sólo el contenido de una única celdilla de memoria.

PROGRAMAS

Las instrucciones de un programa se guardan en memoria como secuencias de octetos. Algunas instrucciones requieren sólo un octeto, mientras que otras precisan hasta 4 octetos.

NUMEROS

Cada celdilla de memoria puede utilizarse para guardar números enteros positivos o números enteros con signo, según *nosotros elijamos*. La gama de números puede ser, por tanto, de 0 a 255 ó de -128 a +127.

NUMEROS NEGATIVOS

Se ha adoptado un convenio para números con signo (+ ó -):
si el bit 7 está alzado (on), el número es negativo;
si el bit 7 está bajado (off) el número es positivo.

Complemento a doses.

El complemento a doses de cualquier número en representación binaria es su opuesto. Se obtiene *invirtiendo* el número binario (cambiando unos por ceros y ceros por unos) y añadiendo "1".



Introducción

Hemos dicho que *el cerebrito* del Spectrum es la CPU: el microprocesador Z80A. Es una versión más rápida del procesador Z80, fabricado bajo licencia de Zilog Inc.

La única diferencia entre el Z80 y el Z80A, es que el primero *circula a una velocidad* de 2 Mhz/s (Megaciclos por segundo), mientras que el segundo lo hace a un *ritmo* de 3.5 Mhz/s. "El ritmo del reloj" es simplemente una medida de *cua*n *rápido* la CPU realiza sus cálculos. En el Spectrum, se generan por segundo 3,5 millones de impulsos de reloj, es decir, un impulso de reloj cada 0,00000286 de segundo.

La instrucción más rápida que la CPU puede efectuar *toma* 4 impulsos de reloj, mientras que la *más* lenta requiere 21 impulsos. Lo que significa que, aún realizando todas las instrucciones como si fueran de las más lentas, en cada segundo se llevarían a cabo alrededor de 160.000 instrucciones!

El aspecto físico del cerebro

El microprocesador del Spectrum es una pastilla de silicio con 40 patillas numeradas de la 1 a la 40. Estas patillas son las *vías de acceso* del procesador, por las que se comunica con el resto de los *circuitos* del computador. Por ejemplo, el procesador coge la energía que consume mediante la patilla 11; los impulsos de reloj mediante la patilla 6; señala direcciones mediante las patillas 1 a 5 y 30 a 40, y recibe y envía datos mediante las patillas 7 a 15 exceptuando la 11. El resto de los *pines* son para intercambiar señales de control.

Puede que estés totalmente perdido en este momento. Pero no te asombres, es realmente una ventaja para nosotros que no sepamos la estructura interna de la máquina, y que no necesitemos saberlo para usar toda su capacidad. Lo mismo sucede con una calculadora. La estructura física de la máquina es "transparente" al usuario, (en otras palabras, no la vemos). Únicamente estamos interesados en la estructura lógica del calculador, o en este caso del microprocesador Z80, y cómo podemos emplearla para nuestros fines.



El aspecto lógico del cerebro

El Z80 puede dividirse en 5 partes funcionales, a saber:

- i. la unidad de control
- ii. el registro de instrucciones
- iii. el contador de programa
- iv. la unidad aritmética
- v. los 24 registros del usuario (las manos y pies utilizables de la CPU).

UNIDAD DE CONTROL

Podemos ver a la unidad de control como *supervisora* de los procesos en la CPU. Su labor es sincronizar y coordinar la entrada, el tratamiento, y el resultado de la tarea que la CPU ha sido encargada de realizar; vengán las instrucciones del programa en ROM o del propio programa del usuario.

EL REGISTRO DE INSTRUCCIONES

Es una *mano* que la CPU utiliza para anotar la *instrucción corriente* que toca realizar. La tarea completa -especificada por el programa- debe residir en alguna parte de la memoria, sea en la ROM o en la RAM (Random Access Memory). Debes recordar que un programa es una secuencia de instrucciones. Por lo tanto, para *seguir* un programa, la unidad de control tiene que *ir, coger y traer* cada instrucción de la memoria y colocarla en la mano llamada registro de instrucciones.

CONTADOR DE PROGRAMA

Realmente es uno de los *pies* de la Z80 que le dice a la CPU donde está la siguiente parte del programa (la dirección de la siguiente celdilla de memoria de la que la unidad de control debe *recabar* la instrucción). Es como un *capataz* del almacén de instrucciones, anotando la situación de la instrucción que toca *cumplimentar* a continuación.

UNIDAD ARITMETICOLÓGICA

Es la *calculadora* interna de la CPU. Puede efectuar tanto operaciones aritméticas como operaciones lógicas. De todas las funciones aritméticas básicas que tú y yo conocemos, sólo puede realizar sumas y restas simples, incrementar (añadir 1) y decrementar (restar 1), pero no la multiplicación ni la división. Esta unidad puede además comparar números a una mano, o efectuar operaciones con los bits, tales como desplazar *los dedos* alrededor de sí mismos, mantener determinados dedos alzados o bajados, etc.

Como consecuencia de los cálculos que la ALU (Arithmetic and Logic Unit) realiza, se ve afectado el estado de diferentes *banderines* del "registro de banderines". Se comentará con más detalle en breve.



REGISTROS DE USUARIO

Son las *manos y pies* de la CPU, que tú -el programador- puedes controlar.

Hay 24 registros de usuario dentro del microprocesador Z80; algunos son manos, y algunos son pies.

Las *imágenes* que hemos adoptado de manos, pies y cajetines, facilitan, y son una buena representación de lo que está pasando; pero los *expertos* en computadoras tienden a mirarte desdeñosamente si dices: "*entonces el computador traspasó la información de su mano derecha a su mano izquierda*". Así que daremos ahora los nombres apropiados para las manos y pies de la CPU, de manera que cuando *encares* una situación parecida seas capaz de decir:

"LD B, A"

Para comenzar, los expertos se refieren a las manos y pies de la CPU como REGISTROS. Habíamos mencionado antes que la CPU tiene 8 manos: se denominan A, B, C, D, E, F, H y L. En nuestro mundo la definición de *mano es algo con 8 dedos*.

La CPU tiene 2 pies: se llaman IX e IY. La definición de un *pie es algo con 16 dedos*.

Los nombres de las manos y pies son bastante fáciles de recordar; ya que si un registro tiene como nombre sólo una letra debe ser una mano (y contener 8 bits), mientras que si tiene dos letras debe ser un pie (esto es, tiene 16 bits).

¿Notaste la suave transición de dedos a bits, y de manos y pies a registros? En menos que canta un gallo te habremos acostumbrado a la terminología adecuada.

Realmente, las dos manos restantes de la CPU después de D, E, F, no reciben el nombre de G y H -como uno podía esperar- sino de H y L.



La forma convencional de representar todos estos registros es la siguiente:

A	F
B	C
D	E
H	L
IX	
IY	

Observa que F está *emparejado* con A, pero quitando eso el resto es bastante natural. La razón de emparejar los registros de esta manera es que, a veces, es posible construir un *pie a base de manos*.

Después de todo, si la definición de pie es algo con 16 bits, puede ser que podamos vacilar de vez en cuando y utilizar 2 manos de 8 bits para hacer el trabajo de un pie. Solemos hablar por tanto, de pares de registros tales como BC, DE y HL.

Hay una razón para que el par de registros HL se llame HL, en lugar de llamarse GH u otra cosa. Para ayudar a que recuerdes cual de los dos registros contiene el dígito mayor y cual contiene el dígito inferior:

Es como si desearas representar los números del 0 al 100 con tus manos y pies. Fácilmente puedes establecer que tus dedos de las manos representan los números de 0 a 10, y que los dedos de tus pies (suponiendo que seas lo suficientemente ágil) también. Una forma de marcar el número 37 sería marcar 3 con los dedos de la mano, y 7 con los de los pies. Pero tiene que haber un *acuerdo* con tu interlocutor sobre cual es la cifra de las decenas y cual es la cifra de las unidades; de otra manera, cualquiera puede pensar que estás queriendo decir 73 en lugar de 37.

La H en HL corresponde a ALTO y la L a BAJO. Así que... ningún inglés puede confundirse, porque para decir "alto" dicen "High" y para decir "bajo" dicen "Low". (No es necesario aclarar que los microprocesadores hispanos están por venir).

Este diagrama con los registros emparejados también sirve para indicar qué registro de las otras parejas contiene el dígito mayor: B en BC - D en DE.

Porque todos los altos y los bajos se tratan siempre en el mismo orden.

Los pies (IX e IY) también tienen un nombre especial: se llaman registros *indiciales*. Lo que tiene mucho que ver con el hecho de que se emplean para organizar la información de manera análoga al *índice de un libro*. También puedes verlos como punteros (palos largos que servían para apuntar o señalar cosas en la pizarra), dirigidos hacia una celdilla de memoria. (También verás que se llaman *indexados*, y recuerda que index e índice es lo mismo).



OK, ahora que comprendes la terminología, vamos a matizar algunos puntos:

EL ACUMULADOR (Registro A)

Este registro de 8 bits (un solo octeto) es el registro más importante del Z80. Su nombre data de las primeras generaciones de computadores cuando únicamente había sólo un registro que se utilizaba para acumular los resultados parciales de las operaciones.

Sin embargo, aunque hemos avanzado desde esa primera generación, el acumulador continúa utilizándose ampliamente para las operaciones aritméticas y lógicas. De hecho, la mayoría de los procesadores todavía están diseñados de forma que muchas operaciones sólo pueden efectuarse utilizando el registro A. También es verdad en la cucaracha Z80, y el registro A es un registro preferente. Puedes mirarlo como la *mano derecha* de la CPU, de la misma forma que la mayoría de la gente puede efectuar algunas tareas más fácilmente con su mano diestra que con su mano zurda.

El registro "F"

Observa, por favor, que "AF" no se trata habitualmente como una pareja de registros. La F en este caso sirve para indicar el *flamear de banderas* (y Flag es banderín en inglés). Es una mano con 8 dedos, en la cual, cada dedo sujeta una banderita que cuando está alzada indica que sí se cumple una determinada condición y cuando está bajada que no se cumple. Como lo ampliaremos en otro capítulo; alza ahora un banderín o hazte un nudo en el pañuelo para recordarlo.

El par de registros HL.

De las tres parejas de registros (BC, DE, HL), la pareja HL es probablemente la más importante. Además de dar al usuario la opción de usarlo como dos registros independientes, o como una pareja de registros; el Z80 está diseñado de manera que hay ciertas operaciones aritméticas con *números a dos manos* que solamente pueden efectuarse utilizando este par de registros. A causa de este particular *privilegio* desde el punto de vista circuital, las operaciones generales con pares de registros, son normalmente más rápidas utilizando la pareja HL. Esto hace que prefiramos utilizar en lenguaje-máquina el par de registros HL.

¿Podemos considerar al registro HL como el *pie derecho* de la CPU?
Salvo que uno sea zocato, es bastante aproximado.



Yo pensé que podía ser la ocasión propicia para mencionar que la CPU también tiene un grupo de manos de reserva. Realmente no tanto como un conjunto de manos de repuesto (de acuerdo, el conjunto de registros alternativos, si prefieres la terminología al uso), sino un conjunto de recambio de guantes de trabajo.

Pero son unos guantes de plástico tan acartonados que de hecho retienen la forma de tu mano cuando te los quitas. Si por ejemplo, has marcado el número 3 con tu mano, subiendo el meñique y el anular, al quitarte tus guantes, éstos todavía mantienen la forma de la mano con esos dos dedos alzados!

Sin ninguna duda, ya puedes pensar como utilizar esos guantes:

puedes tomar nota de un número mientras llevas unos guantes, canjeas los que llevas por los de reserva, y el número antiguo todavía estará ahí siempre que lo necesites -en el par de guantes que te has quitado!

Claro que tendrás que volver a intercambiar los guantes para poder utilizar la información que los guantes retienen.

La CPU tiene guantes de reserva para cada par de manos (pero no para los pies. ¿Quién lleva guantes en los pies?)

Pero no son intercambiables entre manos, así como no se puede poner un guante derecho en la mano izquierda.

Si representamos todos los registros, tendríamos:

A - F	↔	A' - F'
B - C	↔	B' - C'
D - E	↔	D' - E'
H - L	↔	H' - L'
IX		
IY		

Observa que el conjunto de guantes que estás llevando, tiene el mismo nombre que la mano correspondiente, mientras que el conjunto de reserva está indicado con un apóstrofe y la misma letra.

Las instrucciones siempre hacen referencia a lo que están haciendo las manos, no al par de guantes que llevan. Así, aunque señalemos el grupo de reserva con un apóstrofe, no existe una instrucción como LD A', l. La CPU, siempre y únicamente trabaja con sus manos, no con sus guantes.



Las únicas instrucciones en que el grupo de registros alternativos está involucrado, son las de "canjear ahora los guantes", por ejemplo:

1. LD A, (CAJ#1)	; Carga A con el contenido del cajetín 1.
2. EX AF, AF'	; EX es abreviatura de EXCHANGE (lo que pone ; en la ventanilla de cambio de moneda) ; Canje de guantes en registros A y F.
3. LD A, (CAJ#2)	;
4. EX AF, AF'	; Otro intercambio de guantes.
5. LD A, (CAJ#3)	;

Este ejemplo intenta demostrar el concepto del conjunto de registros alternativos. Intenta determinar qué sucede.

(CAJ 1) = 1
(CAJ 2) = 2
(CAJ 3) = 3

Te presentamos lo que sucede después de cada instrucción:

	Registro A	Registro A'
1.	1	No se sabe
2.	No se sabe	1
3.	2	1
4.	1	2
5.	3	2

★ Es bastante simple ¿no te parece?

Te darás cuenta que el intercambio de registros es particularmente útil cuando *se te acaban las manos*, (se te acaban los registros), y no quieres liberar tus manos o pies guardando lo que esté marcado en ellas, en la pila o en la memoria. Profundizaremos en este punto más adelante.

¿Todavía hay más registros?

Sí señor, todavía hay más registros, pero probablemente no los utilices a menudo.

EL PUNTERO DE LA PILA

El puntero de la pila es otro de los pies que la CPU tiene (un registro para direcciones; con dos octetos).



Siempre apunta al lugar donde está la *cima* de la pila. Y la pila *crece*, y *crece hacia abajo*, desde las celdillas de memoria con una dirección mayor hacia las celdillas de memoria con direcciones menores.

Habitualmente no tienes que hacer absolutamente nada con el puntero de la pila, cuando estás programando en lenguaje-máquina.

La CPU se preocupa de ello y actualiza este puntero cada vez que tú *empujas* (PUSH) o *tiras* de (POP) información en la pila.

A menudo, se comete el error de olvidar que hay que volver a coger (POP) el valor que has apilado (PUSH). Puedes estar seguro que eso hará que tu programa *se la pegue*.

El registro I

Este es el registro para los vectores de interrupción.

En los sistemas basados en el sistema Z80 distintos al Spectrum, este registro normalmente se utiliza para guardar la dirección de comienzo de una tabla de direcciones que trate las diferentes respuestas ante una posible interrupción.

Sin embargo, en el Spectrum no se utiliza de esta forma; y el registro I está involucrado en generar las señales de cuadro de televisión. Es improbable que tengas que usar este registro.

El registro R

El registro R es un registro para refrescar la memoria.

Está incorporado en el Z80 para refrescar automáticamente las memorias dinámicas. A medida que el Z80 está haciendo su trabajo, la información almacenada en memoria dinámica, que no ha sido *consultada* recientemente, *se desvanecerá* (porque todo "condensador" se va descargando). A menos que estas celdillas de memoria sean refrescadas (recargadas), la información almacenada en ellas desaparecerá.

El registro R se comporta como un simple contador que se incrementa cada vez que se produce una lectura en la memoria. El valor del registro R recorre repetidamente desde 0 a 255.

Por eso se puede -mediante los circuitos- garantizar que todas las partes de la memoria se *refrescan* adecuadamente.

Pero no te preocupes: tú no necesitarás saber nada de eso.

Es algo que a Mr. Sinclair ya le preocupó cuando diseñó el Spectrum.

Nosotros simplemente utilizamos su computador sin inquietarnos por esto.

Desde el punto de vista de la programación, puedes considerar que el registro R está relacionado únicamente con los circuitos. Pero algunas veces puedes utilizarlo como un medio de obtener un número aleatorio entre 0 y 255. Te mostraremos esta forma de utilización posteriormente.



RESUMEN

REGISTROS DEL USUARIO

Hay 8 registros principales de 8 bits en la CPU (A, B, C, D, E, F, H y L) y dos registros de 16 bits (IX e IY). Los registros de 8 bits sólo tienen una letra por nombre, mientras que los registros de 16 bits tienen dos letras.

PAREJAS DE REGISTROS

Seis de los ocho registros de 8 bits pueden en algunas circunstancias ser utilizados en parejas para operar con números de 16 bits.

Son los pares de registros: BC, DE y HL. El nombre HL sirve para recordarnos cual es el octeto de mayor orden (H) y cual es de menor orden (L).

REGISTROS PREFERENTES

El Z80 está diseñado de manera que algunas de las instrucciones de 8 bits sólo pueden efectuarse con el registro A; mientras que algunas instrucciones de 16 bits sólo pueden hacerse con la pareja HL.

CONJUNTO DE REGISTROS ALTERNATIVOS

Los 8 registros principales de 8 bits pueden canjearse con otro conjunto alternativo de registros.

Los valores contenidos en los registros principales son guardados por la CPU en el conjunto alternativo, cuando se *dicta* la instrucción de canjear registros. Se puede seguir operando con el conjunto principal de registros, pero los valores guardados en el conjunto alternativo no pueden ser consultados ni manipulados hasta que se produzca un nuevo canje de registros.



TODO ESTO ESTA MUY BIEN,

PERO ¿COMO SE PASA UN PROGRAMA EN LENGUAJE-MAQUINA?

Ya has oído bastante sobre la CPU y la notación hexadecimal y seguirás pareciéndote que no viene al caso. No explica cómo realmente se rula (RUN) un programa en lenguaje-máquina.

El ZX Spectrum está realmente siguiendo programas en lenguaje-máquina todo el tiempo! (cuando está enchufado). Lo que pasa es que no somos conscientes de ello. Incluso cuando no está haciendo nada, esperando algo en el teclado o mostrando la pantalla desnuda, el computador Spectrum está ocupado, *currando* bajo el control de un programa en lenguaje-máquina.

Ese programa es el que está grabado en la pastilla ROM y al que nos referimos como "sistema operativo". Por ejemplo, la parte del programa que está rulando cuando tú simplemente estás sentado mirando a la pantalla, lleva a cabo las siguientes tareas: *explora* el teclado para ver si pulsas algo; *observa* que no has pulsado ninguna tecla; *presenta* la pantalla vacía correspondiente y *vuelve* a explorar el teclado.

Incluso, cuando tú estás pasando un programa en BASIC, la CPU todavía sigue las instrucciones del programa en lenguaje-máquina. Este programa es de la clase "intérprete" como ya te hemos explicado: él *estudia* tu siguiente instrucción en BASIC; la *convierte* a lenguaje-máquina; *efectúa* esa instrucción, y *vuelve* a interpretar la siguiente instrucción.

Todo eso deja de ser oíerto cuando tú pasas tu propio programa en lenguaje-máquina!

Libertad total frente al sistema operativo!

Utilizando la función "USuaRio" cedemos totalmente el control de la CPU a las instrucciones que hallamos colocado a partir de la dirección USR. La CPU interpretará lo que allí encuentre como instrucciones válidas en lenguaje-máquina y las cumplimentará.

Puede ser bastante espantoso -ya que puedes perder todo lo que tienes guardado en memoria- si perdieras el control. Un error, una letra errónea, y tendrás que apagar el Spectrum y comenzar de nuevo desde el principio.

No hay mensajes de error que te avisen de lo que has hecho equivocadamente, ni comprobación de la sintáxis en una instrucción incorrecta. De manera que si cometes el error más mínimo, puedes desperdiciar las horas de trabajo que empleaste en *meter* tu programa!

Al final de este libro hemos incluido un programa en BASIC que te permitirá meter y corregir programas en lenguaje-máquina. Una vez que has cargado este programa



en tu Spectrum, guárdalo en cinta, ya que es más que probable que pierdas el control de tu programa en lenguaje-máquina como mínimo, una vez. Por otro lado, no tengas temor a experimentar (no puedes estropear el computador con ningún programa en lenguaje-máquina que metas). Lo peor que puede suceder es que tengas que apagar tu Spectrum y volver a encenderlo.

Justamente ahora te abriremos el apetito con un programa en lenguaje-máquina de lo más simple. Carga el "Monitor de rutinas en lenguaje-máquina" que se encuentra al final de este libro y EJECUTALO

El programa te preguntará por la dirección en dónde cargarlo. Significa dónde quieres que el código máquina *resida*. Con este programa no puedes usar una dirección inferior a 31500, así que elijamos 32000. *Teclea* el número 32000 y luego *pulsa* (ENTER).

La pantalla te muestra ahora:

Comando o línea (###):

Esto significa que el programa está esperando que le dictes un **comando** o un número de línea en código máquina.

Tecleemos "1"; luego un 'blanco'; luego "C" y luego '9'. Es como dar una línea en BASIC numerada con 1, pero es una línea en código máquina. Si todo está OK, pulsa ENTER. La pantalla te mostrará ahora la línea metida:

1 C9

y en el fondo de la pantalla te recordará:

"Comando o línea (###):

En este momento no quieres añadir ninguna línea más, así que dictaremos ahora un comando.

Teclea el comando que hace volcar (DUMP) el código máquina de tu listado en la dirección que has especificado, a saber la 32000. (Y luego pulsa ENTER).

Enhorabuena; ya has metido una instrucción en lenguaje-máquina de un programa!

Puedes comprobar que la ha guardado correctamente, tecleando el comando MEM, seguida de ENTER. Este comando te permite **examinar** el contenido de la memoria, por lo que te preguntará la dirección de comienzo. Teclea 32000 y luego ENTER.

Te mostrará el contenido de las celdillas de memoria, desde la 32000 hasta la 32087. Todas mostrarán 00, excepto la 32000 que te mostrará C9. Pulsa la tecla "M" para volver al principio del programa.

Lo que la instrucción "C9" significa es: RETURN! (vuelta)



Es un poco como montar en bicicleta por primera vez: realmente quieres quedarte suelto por tí mismo, pero tan pronto como has andado un poquitín, quieres "volver" a la seguridad del suelo (o al sistema operativo como es este caso).

Rulemos ahora nuestro programa en lenguaje-máquina. Para pasar cualquier programa en lenguaje-máquina, tienes que haberlo volcado en la memoria y dar el comando RUN seguido de ENTER.

¿Qué pasó? ¿Por qué la pantalla mostró 32000 al final de la pantalla? Esa fue la dirección utilizada como dirección de carga del principio del programa.

★ No olvides que la función de "USR" es ejecutar una subrutina en lenguaje-máquina. Como parte de esa función, el valor de USR al volver del programa que tú tienes en memoria, será el valor de la pareja de registros BC.

La respuesta se basa en la forma en que el sistema operativo del Spectrum (sí, ese mismo) trata la función "USR".

Cuando el sistema operativo encuentra la función "USR" carga la dirección que el usuario especificó, en el par de registros BC -en este caso 32000-.

El valor de "USR", por ejemplo en

LET A = USR 32000

es claro que nos da la respuesta 32000.

Esta peculiaridad de la función "USR" demostrará su utilidad al permitirnos observar lo que sucede al pasar un programa en lenguaje-máquina.

Metamos el siguiente programa en lenguaje-máquina:

OB
C9

La forma de meter este programa de sólo dos instrucciones es la siguiente:

Para meter la línea 1 OB: teclea '1', luego 'blanco', luego '0', luego 'B' y luego pulsa ENTER. Similarmente mete la línea 2 C9. El listado debe mostrarte que has metido las líneas correctamente. Teclea el comando DUMP y luego el comando RUN.

Esta vez, el resultado será 31999! Porque la instrucción OB es "DEC BC" (abreviatura de DECrementar el valor de BC en 1).



Ejercicio:

Experimenta con las instrucciones que operan con el par de registros BC y que puedes consultar en la tabla del final. ¿Puedes deducir lo que las abreviaturas significan?

Ten presente que la última línea de todos tus programas debe ser "C9". Es la instrucción de volver (RETURN), y si lo olvidas, el programa nunca devolverá el control al sistema operativo.

Si esto te sucede, no te preocupes -tu computador no ha sido dañado. Simplemente apágalo y vuelve a cargar todo.

Ejercicio:

Puedes utilizar el comando "mem" para examinar cualquier parte de la memoria. Prueba con diferentes direcciones en las que pienses que vas a *toparte* con algo interesante.



Introducción

Hemos visto que la CPU del ZX Spectrum tiene 24 *manos y pies*. Justamente las operaciones que están permitidas y la facilidad con que son ejecutadas por la CPU, son la *clave* para la programación en lenguaje-máquina de tu Spectrum.

Imagínate por un momento convertido en una CPU:

Probablemente como la mayoría de la gente, eres 'diestro' y puedes hacer cosas con tu mano derecha que no te son fáciles de hacer con tu mano izquierda. Hay además ciertas acciones que son fáciles de realizar de una forma, pero más difíciles de otra. Como coger algo de un estante con tu pie izquierdo y pasarlo a tu mano derecha; es más difícil que hacerlo si utilizas tus manos izquierda y derecha.

Es lo mismo con el lenguaje-máquina: tú puedes efectuar algunas tareas fácilmente de una forma, con más dificultad de otra, y puede que sea imposible según una tercera. Saber cuáles combinaciones de acciones son las permitidas, es la clave del éxito.

La mano de la CPU equivalente a tu mano derecha es el registro A. ¿Lo recuerdas? El acumulador, la mano que surgió como resultado de la herencia genética de los primeros computadores.

Por otro lado, puedes guardar temporalmente lo que tienes en tu mano derecha sobre cualquier otra mano, pie y viceversa. Los *inventores* de los computadores se refieren a esto como direccionamiento de registros. Lo que es un nombre demasiado pomposo para decir transferencia de información de un registro a otro.

Otros ejemplos serían

LD A, B
LD H, E

y así seguirían.

Observa por favor, que LD es el nemónimo (abreviatura) de "cargar" (LOAD) y que cuando en lenguaje ensamblador ves una coma se lee "con". Por tanto LD A, B lo leeríamos como

CARGAR A con B

Una instrucción en lenguaje ensamblador se lee en el mismo *orden* en que se leería una *frase* normal.

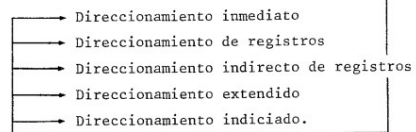


Hay además otras combinaciones o formas distintas al direccionamiento de registros, en que la información puede ser transferida de un registro a otro o de un registro a la memoria.

LAS FORMAS EN QUE PUEDES UTILIZAR LAS EXTREMIDADES DE LA CPU:

Una de las ventajas del procesador Z80 es el gran número de manos y pies, y las posibles combinaciones (modos de direccionamiento) de que dispone.

Echemos un ojo a las combinaciones ofrecidas por el Z80:



Vaya una lista de nombrecitos. No te preocupes, ten confianza y nos los *merendaremos* uno a uno en breve. Y los entenderás, aunque se usen nombres parecidos, v.g.: indexado.

La lista anterior no cubre todas las posibles combinaciones -sólo aquellas que corresponden a los números a una mano-. Tratemos cada una de estas posibles *contorsiones* por turno:

Direccionamiento inmediato

Su forma general es:

LD r, n

(usamos LD como un ejemplo pero vale para otras instrucciones).

Usamos la abreviatura "r" para indicar cualquier registro de 8 bits y la "n" para cualquier número de 8 bits.

El direccionamiento inmediato es una técnica que se emplea sólo para una única mano. En realidad el dato es una parte de la instrucción, esto significa que la CPU puede ejecutar la instrucción inmediatamente que la recibe. No necesita *buscar* en la memoria ninguna otra información para cumplir esta instrucción.

Por ejemplo, para *anotar* el número 215 en la mano A. Estoy seguro de que ya sabes lo bastante de *memónimos* para ser capaz de escribir esto como: LD A, 215 ó LD A, D7H. Te reiteramos que puedes hacer esto con cualquiera de los registros y con cualquier número.



El FORMATO para el direccionamiento inmediato se muestra a continuación:

octeto 1	código de instrucción	(que le dice al computador cuál es la instrucción)
octeto 2	n	(el valor del dato real de esa instrucción).

★ Dado que sólo hay un octeto reservado para el dato, el número que puedes especificar debe estar dentro de la escala 0 - 255. Si no lo entiendes todavía, vuelve a estudiar el capítulo "la forma de contar de los computadores".

Habitualmente utilizamos el direccionamiento inmediato para inicializar los contadores y para definir las constantes que necesitamos en los cálculos.

El direccionamiento inmediato es fácil de usar en lenguaje-máquina. Sin embargo, es el menos flexible de todas las transacciones (modos de direccionamiento) dado que el registro implicado y el dato, han de determinarse en el momento de escribir el programa. La instrucción BASIC equivalente sería

LET A = 5

● Obviamente necesitamos esta clase de instrucción, pero no podríamos escribir programas sólo con esta clase.

El direccionamiento inmediato es muy conveniente pero no resuelve ningún gran problema.

Pero como mínimo empezamos a llegar a algún sitio. Nosotros -los programadores- podemos especificar ahora los números que cargamos en cada registro.

Direccionamiento de registros

Ya hemos comentado este modo anteriormente. Su formato general es LD r, r (u otra instrucción distinta a LD).

Esta técnica sólo hace referencia a dos manos; es pasar información de una mano a otra.

La CPU permite el paso de información entre dos manos cualesquiera, excepto la mano "F" (que no debiera considerarse nunca como mano, ya que es el registro de "banderines" y no guarda números en la forma normal).

Las instrucciones con direccionamiento de registros sólo necesitan un octeto.

Las instrucciones de esta clase no solamente son cortas (un octeto) sino que también son muy rápidas. El tiempo necesario para ejecutarlas es el correspondiente a 4 impulsos de reloj; o sea: menos de 1 microsegundo en el Spectrum.



* Hay una *regla* al escribir programas en lenguaje-máquina: Las transacciones (transferencias de registros a registros) de mano a mano, deberán utilizarse siempre que sea posible, para *mejorar* la eficacia del programa tanto en tiempo como en ocupación de memoria.

Direccionamiento indirecto de registros

LD (rr), A ó LD A, (rr) ó LD (HL), n

Esta potente clase de instrucciones *provoca* la transferencia de datos entre la CPU y la celdilla de memoria a la que *apunta* el contenido de uno de los pares de registros de 16 bits (los pies).

El direccionamiento indirecto de registros es más rápido que el direccionamiento indirecto ordinario, dado que la CPU no necesita coger la dirección de la memoria.

Sin embargo, debemos cargar anteriormente el registro, por lo que el direccionamiento indirecto de registros *sólo* es ventajoso cuando el programa utiliza muchas veces la misma dirección, o una muy cercana. Por ejemplo:

	LD HL, FORMA	; carga HL con el comienzo de la tabla denominada <i>FORMA</i> .
<u>LAZO</u>	LD A, (HL)	; recaba un dato
	INC HL	; mueve el puntero hacia adelante.
	continúa LAZO	
		hasta acabar con la tabla <u>FORMA</u>

Direccionamiento extendido

LD A, (nn) ó LD (nn), A

En el direccionamiento extendido, la instrucción del programa suministra a la CPU una determinada dirección mediante dos octetos. Si la transacción es desde o hasta el acumulador, la transferencia de información sólo afectará al contenido de la celdilla de memoria mencionada por el entero de dos octetos.

Si la transacción es desde o hasta una pareja de registros, tanto el contenido de la celdilla de memoria mencionada por el entero de dos octetos y la siguiente celdilla de memoria se verán afectadas.

El *formato* de esta clase de instrucciones es:

octeto 1	código de operación
octeto 2	(posible subcódigo de operación)
octeto 3	parte inferior del valor de 16 bits.
octeto 4	parte superior del valor de 16 bits.



Esta es la forma en que un programa puede *leer* la memoria y colocar el valor en los registros del usuario. También, requiere una dirección absoluta; en otras palabras, el programa que resulta de utilizar esta clase de direccionamiento no puede ser reubicable, excepto cuando la dirección absoluta que se menciona en la instrucción es reubicable. (Y recuerda que reubicar es cambiar de *ubicación* o sitio).

eg.	FORMA DB n, n, n, ... ;	los datos de la tabla FORMA
	LD A, (FORMA) ;	cargar el primer octeto de la tabla en el acumulador.

Direccionamiento indiciado

LD r, (IX/IY) + d	6	LD (IX/IY + d), r
-------------------	---	-------------------

(o cualquier otra instrucción).

Esta clase de transacción *involucra* a un pie de la CPU, el registro indicial IX o IY.

La CPU *añade* al contenido del registro indicial la dirección especificada en la instrucción y así se obtiene la dirección efectiva del operando.

Es una de las instrucciones en la Z80 que tiene un código de operación de dos octetos. Otra *clase* de instrucciones bastante común con 16 bits en el código de operación, es la de las instrucciones que transfieren bloques de memoria, v.g.: (Cargar, incrementar y repetir).

La utilización típica de esta clase de direccionamiento es efectuar operaciones con *tablas*. Los registros indiciales pueden utilizarse como *punteros* al comienzo de la tabla. En la instrucción se da el valor del desplazamiento necesario para determinar la dirección del elemento de la tabla al que el programa desea referirse.

Eg.	LD IX, COMITABLA ;	inicializa el puntero al comienzo de la tabla
	LD A, (IX + 3) ;	corresponde al tercer octeto a partir del comienzo de la tabla.

El *formato* de las instrucciones de esta clase es:

octeto 1	código de operación	
octeto 2	código de operación	
octeto 3	d	; el valor del desplazamiento



El número "d" es un número de 8 bits que ha de especificarse juntamente con la instrucción y que no puede ser una variable. Esto es, el *ámbito* del direccionamiento está limitado desde -128 a +127 a partir de la dirección que indica el registro indicial.

El direccionamiento indiciado es más lento porque la CPU debe efectuar una suma con el fin de obtener la dirección efectiva.

Sin embargo, el direccionamiento indiciado es mucho más *flexible*, dado que la misma instrucción puede manejar todos los elementos de una serie o tabla.



RESUMEN

Hay varias formas para que la CPU pueda recabar de la memoria información en octetos (8 bits), o transferirla desde los registros a la memoria.

- **Direccionamiento inmediato**

Definiendo en el programa el número que ha de transferirse a cualquiera de los registros.

- **Direccionamiento de registros**

De cualquier registro a cualquier otro registro.

- **Direccionamiento indirecto de registros**

Bien usando BC ó DE para especificar la dirección, y el acumulador para guardar el número que va a transferirse.

Bien usando HL para especificar la dirección y definiendo el número dentro de la instrucción.

- **Direccionamiento extendido**

Especificando la dirección en el programa y empleando el acumulador para contener el número de 8 bits.

- **Direccionamiento indiciado**

Utilizando IX ó IY para especificar el comienzo de una tabla, y cualquier registro para guardar el número de 8 bits.

Debe especificarse en el programa, el desplazamiento a partir del comienzo de la tabla.

El número a transferir puede también especificarse en el programa, si se desea.

Estos modos de direccionamiento son los únicos modos de transferir información hasta y desde la memoria. No se permite ninguna otra combinación.



<u>Nemónimo</u>	<u>Octetos</u>	<u>Tiempo</u>	<u>Efecto sobre los banderines</u>					
			C	Z	PV	S	N	H
LD Registro, Registro	1	4	-	-	-	-	-	-
LD Registro, Número	2	7	-	-	-	-	-	-
LD A, (Dirección)	3	13	-	-	-	-	-	-
LD (Dirección), A	3	13	-	-	-	-	-	-
LD Registro, (HL)	1	7	-	-	-	-	-	-
LD A, (BC)	1	7	-	-	-	-	-	-
LD A, (DE)	1	7	-	-	-	-	-	-
LD (HL), Registro	1	7	-	-	-	-	-	-
LD (BC), A	1	7	-	-	-	-	-	-
LD (DE), A	1	7	-	-	-	-	-	-
LD Registro, (IX + d)	3	19	-	-	-	-	-	-
LD Registro, (IY + d)	3	19	-	-	-	-	-	-
LD (IX + d), Registro	3	19	-	-	-	-	-	-
LD (IY + d), Registro	3	19	-	-	-	-	-	-
LD (HL), Número	2	10	-	-	-	-	-	-
LD (IX + d), número	4	19	-	-	-	-	-	-
LD (IY + d), número	4	19	-	-	-	-	-	-

Notación para los banderines:

#	indica que el banderín se ve alterado por la operación.
0	indica que el banderín se baja (se pone a 0).
1	indica que se alza el banderín (se pone a 1).
-	indica que el banderín no se ve afectado.



MARCANDO NUMEROS A UNA MANO

Dado que *todo* en la CPU del Spectrum está diseñado sobre manos de 8 bits y celdillas de memoria de 8 bits, es indudablemente de primordial importancia, aprender cómo *anotar* números con una mano.

Hemos comentado en un capítulo anterior, alguna de las formas en que podemos transferir información de mano a mano. Ahora trataremos cada uno de esos métodos con más detalle. Puedes *recordar* que uno se denominaba direccionamiento de registros. Como hemos dicho, es un nombre petulante para decir transferencias de información de un registro a otro.

Son ejemplos:

LD A, B
LD H, E
e t c...

Recuerda la *forma* de hablar: LD significa cargar; ", " significa con; y se lee en el mismo orden que se pronuncia una frase.

Así en voz alta, para algo así como LD A, B *déjamos* "cargar A con B".

El otro ejemplo se *leería* como "cargar H con E".

Como hemos mencionado anteriormente, podemos *traspasar* de una mano a otra la información. Con sólo una *excepción* (el registro F, que no es como los otros registros), puedes *canjear* manos. Incluso la instrucción aparentemente estúpida "LD A, A" está permitida!

La forma breve de estas instrucciones es "LD r, r", en donde "r" representa cualquier registro de 8 bits excepto el F.

OK. Sabemos ahora que podemos canjear la información entre las manos, pero eso no nos hará falta si no tenemos alguna información *original* en esas manos.

La segunda forma en que podemos *marcar* números con nuestras manos, será especificando a la CPU lo que queremos marcar y en qué mano lo queremos.

Por ejemplo, *marcar* 215 en la mano "p". Estoy seguro de que ya sabes lo suficiente acerca de los *nemónimos* para que puedas escribir eso como:

LD D, D7

(ya que D7 es la representación hexadecimal de 215).

Quizá recuerdes que esto se llamaba direccionamiento *inmediato*. (Bastante obvio, ¿no es así?)



De nuevo insistimos en que puede hacerse con cualquiera de los registros y con cualquier número. La limitación está en el tamaño del número que puede especificarse con 8 bits: de 0 a 255.

La forma taquigráfica de estas instrucciones es LD r, n
siendo "r" cualquiera de los registros y "n" cualquier número.

El convenio de que una letra implica siempre 8 bits, todavía está en vigor.

Ahora comenzamos a *llegar* a algún sitio: ahora podemos especificar qué números queremos cargar y en qué registros; y además podemos *traspasarlos* de una mano a otra. Pero todavía no hemos aprendido cómo poner cualquiera de estos números en una celdilla de memoria, y *hay tantos muchos* registros!

Te hemos mostrado muy brevemente un ejemplo de "direccionamiento externo", cuando hicimos el ejercicio de la diferencia horaria:

LD A, (Caj#3)

El nemónimo general para estas instrucciones es:

LD A, (nn)

No olvides que en nuestra taquigrafía los paréntesis implican "el contenido de".

Observa dos cosas más:

1. Sólo puedes hacerlo con el registro A.
2. Tienes que dar el número del cajetín como un número a dos manos. (16 bits).

La instrucción contraria también existe. Es una cosa que observarás en el Z80. Hay *simetría* en el repertorio de instrucciones:

LD (nn), A

Tén en cuenta que estas instrucciones solamente se aplican al registro A. Hay desde luego, otras instrucciones para los otros registros, pero no son tan *explícitas* como ésta. Es de nuevo, el concepto de la mano *diestra*.

Descansemos durante un nanosegundo y consideremos lo que estas dos instrucciones significan realmente, y lo que hacen.

En primer lugar, la gama de números que puede definirse mediante un número a dos manos (nn) es de 0 - 65,535. Son los *famosos* 64K y significa que la máxima memoria que podemos manejar con esta instrucción es de sólo 64K! Esto significa que toda la memoria -ROM, programa, pantalla, memoria libre- tiene que *encajar* en las 64K. En un Spectrum con 16K hay *realmente* 16K ocupadas por la ROM y 16K de RAM que hacen un total de 32K. Al decir 16K, hacemos referencia únicamente a la parte de



RAM. En un Spectrum con 48K, las mismas 16K de ROM anteriores continúan presentes, y además hay 48K de RAM para hacer el total de 64K! No es posible, por tanto, que el Z80 acepte más memoria que la que está disponible en el Spectrum con 48K.

La instrucción "LD A, (nn)" que se leería "cargar A con el contenido de la celdilla nn", es una instrucción *pero que muy potente*. Nos permite *leer* el contenido de cualquier celdilla de memoria, ya sea en ROM o en RAM.

★ Puedes usar esta instrucción para *explorar* bajo los dictados de tu corazón, incluso donde no hay memoria -eg.: intenta ver qué hay más allá de las 32K de memoria incluso si no tienes memoria adicional. Te sorprenderás: no todos son zeros!

La instrucción *contraria* "LD (nn), A" -que se lee como "cargar el contenido de la celdilla nn con A"- intentará escribir en la celdilla mencionada, pero estará sujeta a las limitaciones físicas:

No puedes escribir en una celdilla que no puede almacenar información, como sucede al dirigirte a una celdilla que esté más allá del tamaño de tu sistema.

Una de las limitaciones de esta instrucción es que tendríamos que saber, en el momento de escribir el programa, cuál es la celdilla que deseamos *examinar* o en la que queremos *escribir*. La abreviatura 'nn' significa un número determinado. eg.: 17100 -y no una variable.

El principal uso de esta instrucción es para *apartar* determinadas celdillas de memoria para el almacenaje de variables.

Por ejemplo, en un programa de los de *alunizaje*, definiríamos:

32000 = velocidad
32001 = altura
32002 = combustible

Puedes plantearte un programa en el que sacas el valor del combustible, lo disminuyes según el recorrido, y almacenas la nueva cantidad de combustible en esa misma celdilla. En el *momento* de escribir tu programa has de saber la dirección de la celdilla de memoria que te sirve para actuar como almacén de esa información sobre combustible.

Seamos claros sobre esto. La celdilla 32002 no es una *variable*. Es solamente una celdilla de memoria que usamos para almacenar información. Cuando escribas tu programa en lenguaje ensamblador podrías escribir algo como LD A, (combustible) y cuando *tú*, o el *programa* ensamblador, tuviera que especificar el código máquina real para esta instrucción, tendría que sustituir (combustible) por la dirección en Hexadecimal de la celdilla de memoria que contuviera ese dato.

❖ Pero, ¿qué pasa si no sabemos la dirección exacta de la celdilla de memoria donde consultar una información? Supongamos que solamente podemos calcular dónde esa información va a estar. Dado que necesitamos 16 bits



para especificar la dirección de cualquier celdilla de memoria, necesitamos almacenar esa dirección en un registro de 16 bits; lo que significa uno de los pares de registros BC, DE o HL, o uno de los registros índices IX o IY.

Una forma en que podemos hacer esto, es tener en uno de los pares de registros la dirección de la celdilla de memoria. Como el registro contiene la dirección y nosotros no sabemos *directamente* esa dirección, a esta forma de indicar celdillas la llamamos direccionamiento indirecto de registros.

Las abreviaturas *nemotécnicas* para esto, es:

LD r, (HL)
LD A, (BC)
LD A, (DE)

Que en voz alta, leeríamos

"cargar el registro 'r' con el contenido de la celdilla a la que apunta HL"
"cargar A con el contenido de la celdilla a la que apunta BC"
"cargar A con el contenido de la celdilla a la que apunta DE"

Observa que, al utilizar HL como *puntero* en nuestras celdillas de memoria, podemos cargar el contenido en cualquier registro -incluso H ó L- por muy extraño que te pueda parecer; pero al usar BC ó DE, solamente podemos transferirlo al registro A.

Es consecuencia de que el par de registros HL es un par de registros *favorecido*, de la misma forma que el registro A era el *favorito* de los registros sencillos.

- De nuevo podemos comprobar que existen las *simétricas* a estas instrucciones, y que podemos almacenar información en celdillas de memoria de forma similar!

LD (HL), r
LD (BC), A
LD (DE), A

Y a esto le seguimos denominando direccionamiento indirecto de registros, porque no importa el sentido en que fluye la información.

Alternativamente, podríamos usar los registros índices IX e IY, para que *apunten* a la celdilla de memoria.

La forma taquigráfica de estas instrucciones es:

LD r, (IX + d)
LD r, (IY + d)



Siendo 'r' cualquier registro, y 'd' el desplazamiento desde la dirección a la que apunta IX ó IY. (Que usemos la letra 'd' no debe confundirte, porque no significa el registro "D", sino el desplazamiento 'd').

El número 'd' es un número a una mano (8 bits), que hay que especificar en el momento de programar y que no puede ser una variable. Esto constituye la parte *débil* de esta particular instrucción, por lo que habitualmente se usa sólo para leer y escribir tablas con datos.

La instrucción simétrica también está prevista:

LD (IX + d), r
LD (IY + d), r

Si este modo particular de direccionamiento te suena bastante complicado, no te preocupes: no es probable que lo necesites en tus primeros programas..

La pastilla Z80 usada en los computadores Sinclair, es *versátil* por antonomasia, y tú puedes combinar algunas de las formas de cargar números que hemos descrito. Por ejemplo: puedes combinar el direccionamiento inmediato (ie. dando el número que quieres cargar) con el direccionamiento externo (dar la dirección en que va a cargarse, utilizando un par de registros).

★ Se llamará *-oh, sorpresa-* direccionamiento externo inmediato.

Desafortunadamente, sólo puedes utilizar el par de registros HL y su expresión taquigráfica será:

LD (HL), n

Es útil porque puedes directamente *llenar* una celdilla de memoria sin tener que cargar ese valor previamente en un registro.

Con los registros indiciales es posible una combinación similar que se llama direccionamiento indicial inmediato.

Es de uso más limitado, y su forma abreviada es:

LD (IX + d), n
LD (IY + d), n



Pongamos algunas de estas instrucciones "LD" en práctica.

Sabemos de capítulos anteriores, que al volver de un programa en lenguaje-máquina, el valor de la función "USR" es el contenido de BC. Ejecutemos el siguiente programa:

Carga primero y ejecuta el monitor de rutinas en lenguaje-máquina, y fija la dirección de carga en 32000.

1	OE	00
2	C9	

Ahora usa la comando DUMP que vuelca estos códigos en la memoria.

A partir de ahora, ya no te daremos instrucciones tan explícitas sobre cargar y ejecutar programas en lenguaje-máquina, ya que es un método *lento* y no te aporta ninguna comprensión adicional al programa.

Supondremos que ya te has *familiarizado* lo suficiente con el monitor de rutinas en lenguaje-máquina (que está en BASIC), y con las tablas del final de este libro para que seas capaz de meter un programa. Te presentaremos por tanto, todos nuestros programas como sigue:

OE	00	LD, C, 0
C9		RET

Esta notación, te da el código máquina en la parte *izquierda*, y los nemónimos del ensamblador del Z80 en la parte *derecha*. Indica además muy claramente cuáles son las instrucciones que requieren sólo un octeto (tal como RETurn) y cuáles instrucciones requieren dos octetos, etc. (recordarás que algunas instrucciones del Z80 pueden ocupar hasta 4 octetos).

Otra observación. Intentaremos hacer todos nuestros programas independientes del origen (la celdilla de memoria en que empieza el programa) de manera que no importe la dirección de carga que especifiques.

- Sin embargo, recuerda que estos programas pueden meterse con el monitor de rutinas en lenguaje-máquina del final de este libro, o con cualquier otro programa de carga que te puedas *diseñar tú mismo*.

Antes de *pasar* este programa en lenguaje-máquina (debes volcar el código en la memoria y luego usar la comando RUN). *¿Cuál crees que será el resultado?* El programa *coloca* a 0 el registro C de la pareja de registros BC, y tú sabes que BC comienza con la dirección en que has cargado el programa, que es 32000.

Elige la respuesta:

A.	0000
B.	32000
C.	31896



Ahora pasa al programa. ¿Fue la respuesta la que tú esperabas?

Si no estás seguro de por qué la respuesta fue la que fue, vuelve a leer el capítulo sobre "La forma de contar de los computadores".

Ahora intenta *hacer* el siguiente programa:

06	00	LD B, 0
0E	00	LD C, 0
C9		RET

Te dará el resultado de 0, ya que BC es igual a 0 (tanto el registro B como el C han sido *puestos* a 0).

Ejercicio:

Puede que te *guste* intentar unos cuantos truquitos, tales como cargar A con un número, transferir a L, poner H a 0, y similares.

Ejercicio:

La zona de atributos comienza en la dirección 5800H. Podemos hacer que HL *apunte* a la zona de atributos con el siguiente programa:

26	58	LD H, 58H
2E	00	LD L, 0

Esto significa que puedes ahora cambiar los *colores* de la pantalla utilizando la instrucción LD (HL), n.

La estructura de la zona de atributos, está descrita en el *manual* del Spectrum. Pongamos el primer símbolo a papel rojo, tinta blanco, parpadeo sí. Eso es:

1	0	1	1	0	1	0	= BAH
---	---	---	---	---	---	---	-------

De forma que la siguiente instrucción de nuestro programa ha de ser:

36	BA	LD (HL), BAH
----	----	--------------

Ahora y como nunca debes olvidar *volver* del programa en lenguaje-máquina, nuestra última línea será:

C9	RET
----	-----

Ejecuta este programa en lenguaje-máquina ¿Funcionó?



BANDERAS Y SUS USOS

Los banderines son esas hermosas y ondulantes telas que flameamos en las ocasiones *patrióticas*... -no señor!

En lenguaje-máquina, la palabra *banderoleta* o banderín implica "indicador". Un banderín es algo que *alza* si deseas indicar a alguien que *existe* una determinada condición.

El *símil* obvio es el naval, donde levantas un banderín para indicar desgracia, país, pirata, o lo que sea.

La razón de que los diseñadores del Z80 (y los diseñadores de la mayoría de las CPU) usen banderines en su lenguaje-máquina, es dar al programador información acerca del número contenido en la mano *diestra* de la CPU (el registro A) o información sobre la *última* operación que se acaba de efectuar.

Recordarás que uno de los registros de la CPU está dedicado a ser el registro de banderines -registro F-. Puede que hayas observado además, al comienzo del último capítulo, una tabla resumiendo las diferentes instrucciones comentadas en ese capítulo, y que parte de esa tabla indicaba el efecto que cada instrucción tenía sobre los banderines.

(Afortunadamente ninguna de las instrucciones comentadas en el último capítulo afectaba a ninguno de los banderines).

El banderín cuyo funcionamiento es el más fácil de entender es el banderín de cero (Zero Flag).

Este banderín estará alzado si el contenido del registro A es cero. ¿Es difícil?

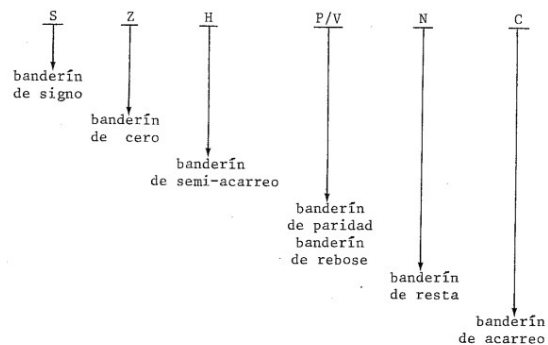
Hay muchas decisiones importantes que dependerán de si A es cero o no. Observa que el banderín de cero está alzado (on) o bajado (off). Tú no puedes tener un resultado intermedio (eso de un poquito preñada no vale aquí). De manera que sólo necesitas un bit para definir el estado del banderín de cero.

Eso mismo es verdad para todos los banderines. Están siempre alzados o bajados, y por ende requieren sólo un bit.

Las diferentes clases de banderines

El registro F es un registro de 8 bits, y puede por tanto, *acomodar* 8 diferentes banderines. En la práctica sin embargo, los diseñadores sólo pudieron pensar en 6 banderines.





Realmente, los diseñadores pensaron en 7 banderines, pero decidieron que un bit podía servir para dos funciones: el banderín de paridad / rebose.

Echémos una mirada minuciosa a cada uno de estos banderines:

Banderín de cero

Es el que ya hemos comentado anteriormente. Su aplicación es obvia, y el banderín se fija después de una operación aritmética, ya que sirve para indicar el contenido del registro A.

Sin embargo, ten en cuenta que es posible que el registro A contenga un 0 y que el banderín de cero no esté alzado. Puede suceder fácilmente si utilizamos la instrucción

`LD A, 0`

Ya hemos mencionado que ninguna de las instrucciones de carga a una mano (8 bits) tiene efecto sobre ningún banderín. El banderín de cero no estará alzado, a no ser que el acumulador ya contuviera cero.

El banderín de cero también se alza si como resultado de las instrucciones de rotar y desplazar obtenemos un 0.

También, el banderín de cero es el único resultado visible de algunas instrucciones de comprobación, tales como el grupo de instrucciones de comprobación de bits. En esos casos, el banderín de cero queda alzado si el bit que se comprobó estaba a cero.

Banderín de signo

El banderín de signo es muy similar al de cero y opera sobre aproximadamente el



mismo conjunto de instrucciones (con la diferencia mayor en el grupo de comprobación de bits donde el concepto de un bit *negativo* es algo que no tiene ningún significado).

Banderín de acarreo

Es uno de los banderines más importantes de los que se dispone en lenguaje ensamblador, ya que sin él, el resultado de la aritmética en lenguaje ensamblador sería totalmente inválido.

El *punto* que hay que recordar es que las instrucciones en lenguaje ensamblador siempre se refieren a números a una mano (8 bits) o a dos manos (16 bits). Esto significa que los números que estamos manejando pueden ser:

de 8 bits	→	0 - 255
de 16 bits	→	0 - 65535

Considerémos la situación en que efectuamos la siguiente resta

$$\begin{array}{r} 200 \\ - 201 \\ \hline \end{array}$$

Resultado = 255 !!!

Es una consecuencia directa de tener únicamente una gama limitada de números disponibles; y la misma cosa puede obviamente suceder con números de 16 bits.

Ya hemos comentado que solamente puedes contar hasta 255 con una mano. ¿Qué pasa si un registro ya *indica* 255 y se le *suma* 1?

- Puedes considerar que el registro funciona de la misma forma que el *medidor de kilómetros* de tu coche. Una vez que ha alcanzado el máximo, da la vuelta y comienza a contar de nuevo a partir de 0.

Por otro lado, si el registro, o el medidor del coche, indica todos ceros, y lo *giras* en sentido contrario, obtienes el valor máximo posible, que será 255 en un registro de 8 bits. Esa es la *causa* de que $200 - 201$ nos dé 255. Si fuéramos vendedores de coches, nos gustaría obviamente una indicación de que el medidor ha dado la vuelta, ya sea hacia adelante -en cuyo caso el coche ha viajado bastante más de lo que parece- o hacia atrás -en cuyo caso el medidor ha sido *amañado*.

Esta clase de *indicador* existe en la programación en lenguaje-máquina y se denomina *banderín de acarreo*. Afortunadamente no necesitamos preocuparnos de que puedan amañarse los registros.



Hemos visto que el banderín de acarreo se alza en aquellas restas en que *nos llevamos una* al restar los bits '7' de los operandos. También se alza en aquellas operaciones de suma en que *nos llevamos una* (positiva) al efectuar la operación con esos bits.

Por tanto es conveniente considerar el bit de acarreo como el noveno bit (bit 8) del registro A:

Número	Bit de acarreo	Número en formato binario
132	-	1 0 0 0 0 1 0 0
+ 135	-	1 0 0 0 0 1 1 1
267	1	0 0 0 0 1 0 1 1

Pero como no tenemos 9 bits, el registro A contendría el número OBH (Decimal 11) y el acarreo sería ON (ie. = 1).

Puedes comprobar en otros ejemplos de resta, que también se alza el bit de acarreo cuando *nos llevamos una*.

Usando banderines para el equivalente en lenguaje-máquina del BASIC: "Si ...PUES..."

En BASIC tenemos la capacidad de considerar situaciones "SI ...PUES..." tales como:

If A = 0 then...
donde lo que sigue puede ser LET
ó GOTO
ó GOSUB

Exactamente la misma clase de *decisión* puede programarse en lenguaje-máquina (excepto LET....). En lugar de decir "IF A = 0", miramos el estado del banderín de cero: si está alzado, pues ya sabemos que A = 0.

Los tres banderines que hemos considerado hasta ahora son, normalmente los únicos que nos permiten hacer una elección para la siguiente instrucción que vaya a ejecutarse.

Por ejemplo:

JP cc, fin

siendo 'JP' el nemónimo para un salto (JUMP) y 'fin' una etiqueta.

Esta instrucción se lee como "salte cuando se cumpla la condición cc hasta fin".



La condición "cc" puede ser cualquiera de:

→	Z	(cero)
→	NZ	(no cero)
→	P	(positivo)
→	M	(menos)
→	C	(ha habido acarreo)
→	NC	(no ha habido acarreo)

Los otros tres banderines no *suelen* ser de mucha utilidad en la programación normal.

Son:

Banderín de Paridad/Rebose

Este banderín actúa en algunas instrucciones como banderín de paridad y en otras como banderín de rebose! Pero no suele haber ninguna confusión ya que los dos tipos de operaciones no ocurren habitualmente juntos.

Como el de paridad tiene efecto durante operaciones *lógicas* y queda alzado si en el resultado hay un número par de bits en uno; trataremos de él en mayor profundidad en el capítulo sobre operaciones lógicas.

Como el de rebose* es un aviso que indica que la operación *aritmética* que se acaba de realizar puede que no *quepa* en los 8 bits; mejor que decirte realmente que el resultado necesita un *noveno* bit, te dice que el *octavo* cambió como un resultado de la operación!

En el ejemplo anterior, al sumar 132 y 135, el octavo bit era 1 *antes* de la suma y 0 *después*; por lo que el banderín de rebose quedará *alzado*. Pero el rebose también se pone a 1 al sumar:

64	0	1	0	0	0	0	0
+ 65	0	1	0	0	0	0	1
129	1	0	0	0	0	0	1

Banderín de resta

Este banderín queda alzado si la última operación fue una resta! Como restar es sumar el negativo, también se llama a veces de negación.

Banderín de semi-acarreo

Este banderín se alza de una manera similar al banderín de acarreo, pero sólo en el caso de que nos *llevemos* una al pasar al quinto bit (mientras que el de acarreo es el noveno bit).

Tanto el banderín de resta como el de semi-acarreo se utilizan sólo en la aritmética "decimal codificada en binario", y los *comentaremos* en el capítulo de aritmética BCD.

*En inglés "overflow". En español lo que tú digas cuando un líquido no cabe en su recipiente, ie.: rebase, rebose, desbordamiento, etc.



RESUMEN

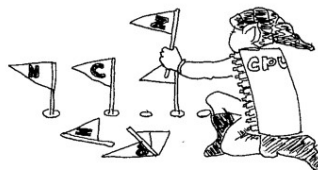
Los banderines se usan en la CPU para indicar que existen determinadas condiciones después de efectuar las instrucciones.

Hay seis de tales banderines, de cada uno de los cuales se dice que está *alzado* (on) o *bajado* (off). Están representados por 6 bits de los 8 que hay en el registro F. Los otros 2 bits no se utilizan.

Las condiciones indicadas por los diferentes banderines son: _____

- * Acarreo
- * Cero
- * Paridad/Rebose
- * Signo
- * Negación
- * Semi-acarreo

No todas las instrucciones afectan a cada banderín. Algunas afectan a todos, mientras otras sólo a banderines específicos y algunas no tienen ningún efecto sobre los banderines.



CONTANDO Y DESCONTANDO A UNA Y DOS MANOS

En el último capítulo, examinamos el concepto de banderines, y en el capítulo anterior, determinamos como la CPU es capaz de *cargar* cualquier número que se desee, sobre los dedos de sus manos y de sus pies.

Examinémos ahora, la forma más simple de manipular números con los propios dedos: podemos aumentar el número representado o podemos disminuirlo.

Es una aritmética bastante *rudimentaria*, pero va más allá que cargar números específicos en los dedos. La acción de contar es *esencial*: cualquiera que sea el número que hayas marcado en los dedos, lo aumenta en 1.

Esto se puede utilizar en situaciones tan ordinarias como al tomar el *censo* o al controlar el tráfico en un cruce determinado.

Contando:

En el Z80 es posible *aumentar* lo que marcan los dedos de cada mano de la CPU; esto es lo que queremos decir mediante el nemónimo general

INC r

INC se lee como *incrementar* y es una instrucción, por tanto, bastante *autoexplicativa*.

También es posible incrementar el número marcado en los dedos de los pies (incluyendo los pares de registros que como sabemos no son realmente pies).

Para incrementar la *cuenta* de los dedos de los pies, se escribe:

INC rr
INC IX
INC IY

en que "rr" denota cualquier par de registros tales como 'BC', 'DE' ó 'HL'.

De nuevo, debes observar la sencilla forma en que hemos indicado las operaciones que utilizan los números de 8 bits y las que emplean los de 16 bits: los números de 8 bits se indican mediante una única letra, mientras que los de 16 bits se indican con dos letras.

Pero la instrucción de *contar* es, de hecho, más potente de lo que pudiera parecer. Se puede *incrementar* el contenido de cualquier celdilla de memoria si especificamos su dirección con los registros indiciales o el par



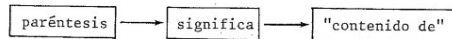
de registros preferido HL:

INC (IX + d)
INC (IY + d)
INC (HL)

siendo 'd' el desplazamiento y no el registro D.

Nota importante:

- Recuerda cuidadosamente nuestro convenio sobre la lectura de paréntesis.



Es muy importante porque hay un montón de *similitud* entre las instrucciones

INC HL
INC (HL)

pero otro montón de *diferencia* en su ejecución.

La *primera* se lee como incrementar HL; mientras que la *segunda* se lee incrementar el contenido de la celdilla cuya dirección es HL (se abrevia normalmente, diciendo incrementar *la que indica* HL).

En tanto y en cuanto, recuerdes las reglas de las abreviaturas nemotécnicas, te evitarás esta clase de confusiones. Examinémos como cada una opera y supongamos que HL = 5800H.

✱ INC HL : Observa HL. Incrementa en 1 lo marcado por sus dedos.

Resultado: HL = 5801H

✱ INC (HL): Mira en HL. Determina la celdilla de memoria a la que apunta HL. Aumenta lo que hay en esa celdilla por uno.

Resultado: HL = 5800H
(5800H) = (5800H) + 1

Son operaciones significativamente diferentes. (Puede que te guste jugar con ambas versiones ~5800H es el comienzo de la región de atributos). Observa, además, que mientras INC HL es una instrucción que actúa sobre un número de 16 bits; INC (HL) es una instrucción que actúa sobre un número de 8 bits; ie. el número guardado en la celdilla 5800H.

Descontando

La naturaleza simétrica del repertorio de instrucciones del Z80 nos permitiría garantizar que todo lo que tú aumentas, también lo puedes disminuir, y desde luego este es el caso.



DEC r
DEC rr
DEC IX
DEC IY
DEC (HL)
DEC (IX + d)
DEC (IY + d)

El nemónimo DEC es abreviatura de decrementar, y también debe aplicarse el mismo convenio para el uso de paréntesis.

Efecto sobre los banderines

Dado que las instrucciones de incrementar o decrementar que operan sobre números de 8 bits, afectan a todos los banderines excepto al de acarreo, puede ser bueno revisar el funcionamiento de los banderines.

NOTA IMPORTANTE:

Las instrucciones de incrementar y decrementar que operan sobre números de 16 bits, NO tienen efecto sobre ninguno de los banderines. Únicamente cuando se incrementa o decrementa un número de 8 bits, es cuando se ven afectados los banderines.

Signo → Este banderín se alzará (= 1) si el bit número 7 del resultado es 1. (negativo).

Esto significa que sólo si el *pulgar* está alzado -usando nuestra analogía anterior- se alzará este banderín. Y sucederá cualesquiera sea el convenio que estemos utilizando para el número.

Cero → Este banderín se alzará (= 1) si el resultado de la operación es cero.

Rebose → Este banderín se alzará (= 1) si el bit número 7 del operando cambia al efectuar la operación.

Semi acarreo → Este banderín se alzará (= 1) si hay un acarreo positivo o hasta negativo desde el bit número 4 del número.

Negación → Este banderín se alza cuando la última instrucción fue una resta. Por tanto, está bajado (= 0) para "INC" y alzado (= 1) para "DEC".

Ejercicios que se sugieren

Usa el grupo de instrucciones, "LD", "INC" y "DEC", para obtener los números que desees como resultado de la operación "USR". Esto te dará *familiaridad* con estas instrucciones.



RESUMEN

Podemos incrementar o decrementar el contenido de cualquiera de los registros de 8 bits, o de cualquiera de los pares de registros de 16 bits, o de cada registro indicial de 16 bits.

Además podemos incrementar o decrementar el contenido de las celdillas de memoria cuya dirección esté especificada por la pareja de registros HL, o por uno de los registros indiciales.

El incremento o decremento de números de 16 bits no afecta a *ninguno* de los banderines.

Al incrementar o decrementar números de 8 bits, ya sea en los registros o en la memoria, *todos* los banderines, excepto el de acarreo se ven afectados.



Instrucciones para operaciones aritméticas a una mano

Memónimo	Octetos	Tiempo	Efecto sobre los banderines					
			C	Z	PV	S	N	H
ADD A, registro	1	4	#	#	#	#	0	#
ADD A, número	2	7	#	#	#	#	0	#
ADD A, (HL)	1	7	#	#	#	#	0	#
ADD A, (IX + d)	3	19	#	#	#	#	0	#
ADD A, (IY + d)	3	19	#	#	#	#	0	#
ADC A, registro	1	4	#	#	#	#	0	#
ADC A, número	2	7	#	#	#	#	0	#
ADC A, (HL)	1	7	#	#	#	#	0	#
ADC A, (IX + d)	3	19	#	#	#	#	0	#
ADC A, (IY + d)	3	19	#	#	#	#	0	#
SUB registro	1	4	#	#	#	#	1	#
SUB número	2	7	#	#	#	#	1	#
SUB (HL)	1	7	#	#	#	#	1	#
SUB (IX + d)	3	19	#	#	#	#	1	#
SUB (IY + d)	3	19	#	#	#	#	1	#
SBC A, registro	1	4	#	#	#	#	1	#
SBC A, número	2	7	#	#	#	#	1	#
SBC A, (HL)	1	7	#	#	#	#	1	#
SBC A, (IX + d)	3	19	#	#	#	#	1	#
SBC A, (IY + d)	3	19	#	#	#	#	1	#
CP registro	1	4	#	#	#	#	1	#
CP número	2	7	#	#	#	#	1	#
CP (HL)	1	7	#	#	#	#	1	#
CP (IX + d)	3	19	#	#	#	#	1	#
CP (IY + d)	3	19	#	#	#	#	1	#

Notación sobre los banderines:

- # indica que el banderín se ve afectado por la operación
- 0 indica que el banderín se pone a cero
- 1 indica que el banderín se alza
- indica que el banderín no se ve afectado.



Aritmética a una mano, es nuestro *recordatorio* de que todas estas operaciones involucran sólo a 8 bits, y todas ellas deben llevarse a cabo mediante nuestra *diestra*: registro A.

Parece que sólo nuestra mano *dominante* sabe cómo sumar o restar!

Este hecho está tan *imbuido* en los nemónimos del Z80, que incluso se omite en algunos de ellos la abreviatura "A". Por ejemplo, para *substraer* B de A normalmente esperaríamos ver

SUB A, B
pero de hecho, el nemónimo es
SUB B

A pesar de esta limitación sobre las instrucciones aritméticas (están restringidas al registro A), el lenguaje del Z80 es *muy versátil* dado que realmente podemos añadir a cualquier número lo que tenemos en nuestra mano dominante:

* ADD A, r	Añade al A cualquier registro simple.
* ADD A, n	Añade al A cualquier número de 8 bits.
* ADD A, (HL)	Añade al registro A el número de 8 bits en el cajetín cuya dirección está dada por HL.
* ADD A, (IX + d)	Añade al registro A el número de 8 bits en el cajetín cuya dirección está dada por IX + d
* ADD A, (IY + d)	Añade al registro A el número de 8 bits en el cajetín cuya dirección está dada por IY + d

Podemos apreciar la gama extremadamente versátil de números posibles que podemos añadir a *cualquier* número que esté guardado en A; cualquier número, cualquier registro y virtualmente, cualquier forma de definir una celdilla de memoria.

- La que *falta* sería ADD A,(nn) en donde definimos la dirección en el transcurso del programa. Por lo que la única forma de hacer esa operación sería:

LD HL, nn
ADD A, (HL)

Observa otra vez el papel *favorito* del registro HL. No podemos especificar la celdilla de memoria usando los pares de registros BC ni DE.

La otra *limitación* implícita, es la limitación inherente de los números de 8 bits que como ya hemos visto sólo pueden tener valores de 0 a 255.

Por ejemplo:

LD A, 80H
ADD A, 81H



Sólo nos dará un uno en el registro A, pero el banderín de acarreo estará alzado para indicar que el resultado no *encaja*.

★ Si la aritmética decimal te *confunde*, es un buen ejercicio convertir los números a decimales y comprobar la suma.

La suma y resta hexadecimales son las mismas que en la aritmética ordinaria:

$$\begin{array}{l} 1 + 1 = 2 \\ 1 + 2 = 3 \\ \text{etc.,} \end{array}$$

y cuando llegas a 1 + 9 tienes

$$\begin{array}{l} 1 + 9 = A \\ 1 + A = B \\ \text{etc.,} \end{array}$$

y cuando alcanzas 1 + F entonces

$$1 + F = 1 \text{ y nos llevamos } 1$$

Así es que el acarreo en la siguiente columna tiene lugar cuando llegas a un número mayor que F, (en lugar de suceder al llegar al nuevo como en la aritmética decimal).

El resultado de nuestras instrucciones en lenguaje-máquina anteriores sería:

80 + 81 --- 101	ya que $8 + 8 = 16 \rightarrow 10H$
--------------------------	-------------------------------------

¿Qué puede hacerse con este error de acarreo?

Los diseñadores del Z80 nos han provisto otra instrucción similar a la de suma (ADD), pero que tiene en cuenta los posibles acarreos. Es una instrucción muy útil: ADC, que leemos como "sumar con acarreo".

Es exactamente la misma que la instrucción de sumar sin acarreo, con la misma gama de números, registros, etc., que puede añadirse al registro A, excepto que también se *añade* el acarreo si está alzado.

Así podemos operar con números mayores de 255, mediante un encadenamiento de operaciones:

eg.: para sumar 1000 (ie. 03E8H) a 2000 (ie. 07D0H) y guardar el resultado en BC:

★ LD A, E8H	; cargar la parte inferior del primer número.
★ ADD A, D0H	; añadirle la parte inferior del segundo número.
★ LD C, A	; guardar el resultado en C.
★ LD A, 03H	; cargar la parte superior del primer número.
★ ADC A, 07H	; añadirle la parte superior del segundo número.
★ LD B, A	; guardar el resultado en B.



Después de la primera suma ($E8 + D0$) tendremos el banderín de acarreo alzado (porque el resultado fue mayor que FF) y el registro A contiene B8 (compruébalo por tí mismo).

La segunda suma ($3 + 7$) no producirá 0AH (= 10 decimal) como pudiera parecer a primera vista sino 0BH (= 11 decimal) precisamente a causa del acarreo.

El resultado final es por tanto, 0BB8H = 3000! Con este encadenamiento, podemos manejar un número de cualquier tamaño, y el resultado almacenarlo en la memoria mejor que mantenerlo en una pareja de registros.

RESTA DE 8 BITS

Pasa exactamente lo mismo que con la suma de 8 bits. Existen dos grupos de comandos, una para la resta ordinaria, y otro para la resta, teniendo en cuenta el acarreo:

→ SUB s	restar (del acumulador) s
→ SBC s	restar (del acumulador) s con acarreo.

La notación "s" quiere indicar la misma gama de operandos posibles que en el caso de la instrucción ADD.



Salgámonos un momento del lenguaje-máquina y consideremos exactamente lo que queremos decir cuando hablamos de comparar dos números:

Sabemos lo que sucede cuando los dos números que se están comparando son el mismo -son "igual"- . Una manera de indicar ésto en un formato aritmético sería decir que la diferencia entre los dos números era cero.

Y si el comparando es mayor que el comparador (la comparación siempre implica relacionar dos números: comparamos el número a lo que ya tenemos en nuestros dedos). Entonces al restar el comparando del comparador, saldría negativo.

Similarmente, si el nuevo número es más pequeño, entonces la diferencia sería positiva.

Podemos usar estos conceptos para *ingeniar*nos un sistema de comparaciones en lenguaje-máquina. Todo lo que necesitamos son los banderines y la operación de resta. Supongamos que quieres comparar una serie de números con el 5;

decimos	LD A, 5	;	el número con el que comparamos.
	SUB N	;	el número que se está comparando.

Entonces, tendremos los siguientes resultados:

Si $N = 5$	El banderín de cero alzado, el banderín de acarreo bajado.
Si $N < 5$	El banderín de cero bajado, el banderín de acarreo bajado.
Si $N > 5$	El banderín de cero bajado, el banderín de acarreo alzado.

Así pues, es claro que la igualdad debe verse por el banderín de cero, y la de "mayor que" por el de acarreo. (La comprobación de "menor que" es ambos banderines bajados).

El único inconveniente de este método, es que habríamos alterado con la operación el contenido del registro A. Afortunadamente, tenemos las operaciones "CP s", que se leen como "comparar (a A) con s".

Observa que solamente podemos comparar a lo que ya tenemos en el registro A. Los números que se pueden comparar al del registro A son los mismos que para la suma o la resta.

"Comparar" es exactamente lo mismo que "restar" excepto en que no se ve cambiado el contenido del registro A. Todo el efecto es, pues, sobre los banderines.



RESUMEN

La aritmética de 8 bits en el Z80 está limitada a:

-suma o adición
-resta o sustracción
-comparación



y sólo puede efectuarse con el registro A.

Sin embargo, y con esta *limitación*, existe una amplia gama de modos de direccionamiento.

A causa de la naturaleza *intrínseca* de los números de 8 bits, debemos siempre ser muy cuidadosos sobre el *acarreo*. El banderín de acarreo (así como los otros banderines) se ve afectado por las operaciones aritméticas; y podemos usarlo como un *aviso* de haber excedido los 8 bits.

Hay instrucciones adicionales (sumar con acarreo y restar con acarreo) que nos permiten *concatenar* operaciones aritméticas para los casos donde se sobrepasen los 8 bits.



Instrucciones para los operadores lógicos

Nombre	Octetos	Tiempo	Efecto sobre los banderines					
			C	Z	PV	S	N	H
AND Registro	1	4	0	#	#	#	0	1
AND Número	2	7	0	#	#	#	0	1
AND (HL)	1	7	0	#	#	#	0	1
AND (IX + d)	3	19	0	#	#	#	0	1
AND (IY + d)	3	19	0	#	#	#	0	1
OR Registro	1	4	0	#	#	#	0	0
OR Número	2	7	0	#	#	#	0	0
OR (HL)	1	7	0	#	#	#	0	0
OR (IX + d)	3	19	0	#	#	#	0	0
OR (IY + d)	3	19	0	#	#	#	0	0
XOR Registro	1	4	0	#	#	#	0	0
XOR Número	2	7	0	#	#	#	0	0
XOR (HL)	1	7	0	#	#	#	0	0
XOR (IX + d)	3	19	0	#	#	#	0	0
XOR (IY + d)	3	19	0	#	#	#	0	0

Notación sobre los banderines:

indica que el banderín queda alterado por la operación
 0 indica que el banderín se pone a cero (bajado)
 1 indica que el banderín se pone a uno (alzado)
 - indica que el banderín no se ve afectado.



OPERADORES LOGICOS

Hay tres operaciones que son tan *valiosas* cuando programamos en lenguaje-máquina (o en ensamblador) como lo son la suma, resta, multiplicación o división en la aritmética ordinaria.

Se denominan generalmente operadores BOOLEANOS (de acuerdo con el nombre del hombre que formuló las reglas de estas operaciones).

Las operaciones son:

Y	(AND)
O	(OR)
O exclusivo	(XOR)

- Ya estamos familiarizados con operaciones que se aplican a un número de 8 bits, considerado como un todo (o sea, *la mano*) pero la razón de que estas operaciones sean tan valiosas es que operan sobre los bits individuales del número (o sea, *los dedos*).

Echémos un ojo a una de esas operaciones, la "Y":

Bit A	Bit B	Resultado de Bit A "Y" Bit B
0	0	0
1	0	0
0	1	0
1	1	1

Es obvio que el resultado de una operación "Y" es darnos un '1' sólo si A y B contienen ambos un '1'.

En lenguaje-máquina, si tu YLIAS dos números, el resultado es lo que obtendrías al YLIAR cada uno de los bits individuales que componen los dos números.

Te puedes estar preguntando a tí mismo *¿a qué viene esta operación tan rara?* La YLIACION es extremadamente útil ya que te permite discriminar un octeto de manera que se vea constreñido a mantener sólo ciertos bits con su valor primitivo.

Si, por ejemplo, deseamos limitar una determinada variable a valores entre 0 y 7; podemos claramente indicar que sólo deseamos que los bits 0, 1 y 2 contengan información (si el tercer bit contuviera información, el número sería como mínimo 8).

eg.

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

 = 5 Estos bits deben ser "0".

Tomamos un número cuyo valor no sabemos y lo YLIAMOS con 7, el resultado será un

"Y" Lógico Inclusivo



número dentro de la gama 0 - 7!!

eg.

0 1 1 0 1 0 0 1	= 105	105 \wedge 7 = 1
0 0 0 0 0 1 1 1	= 7	
0 0 0 0 0 0 0 1	= 1 (en el rango 0 - 7)	

De nuevo observa que la pastilla Z80, sólo permite la *Vliación* con el contenido del registro A. Podemos *Vliar* el acumulador con un número de 8 bits, con cualquiera de los otros registros de 8 bits, con (HL), con (IX + d) o con (IY + d).

eg.

AND 7
AND E
AND (HL)

Como solamente podemos actuar sobre el registro A, no necesitamos mencionarlo en las instrucciones.

La misma gama de posibilidades y de restricciones se produce para las otras operaciones Booleanas: *Oliación* y *oleación*.

La Oliación (operación OR) es muy similar en concepto a la Yliación (operación AND).

Bit A	Bit B	Bit A OR Bit B
0	0	0
0	1	1
1	0	1
1	1	1

Oliación
→ "0"
→ Lógico
→ Inclusivo

Es obvio, que el resultado de una OLIAACION es darnos un uno *sólo si* el registro A, o el registro B, o *ambos* contienen un "1".

También te podrás preguntar sobre el objetivo de una operación así.

La *oliación* también es extremadamente útil ya que nos permite *alzar* cualquiera de los bits de un número: Si queremos garantizar que un número sea impar, es obvio que tenemos que alzar el bit 0. (Ese mismo resultado puede obtenerse utilizando la instrucción "poner bit" que comentaremos posteriormente).

LD A, número
OR 1

; *hacé que el número sea impar.*

Las dos líneas anteriores podrían ser un listado típico en ensamblador.

- El concepto de *oleación* (llamado "o exclusivo" y "XOR") es también fácil de entender, pero su utilización habitual en programación está más limitada.

El resultado de OLEACION es darnos un '1' *sólo si* A ó B contienen un '1', pero



no si ambos contienen un '1'.

Bit A	Bit B	Bit A XOR Bit B
0	0	0
1	0	1
0	1	1
1	1	0

Operación
 "0"
 Lógico
 Exclusivo

La siguiente cuestión que debemos considerar es el efecto que estas operaciones tienen sobre los banderines:

- Banderín de cero* → Este banderín se alzarán (=1) si el resultado es cero.
- Banderín de signo* → Este banderín se alzarán (=1) si el bit 7 del resultado queda a uno.
- Banderín de acarreo* → Este banderín estará bajado (=0) después de AND, OR y XOR; es decir el banderín de acarreo queda siempre bajado.
- Banderín de paridad* → Este banderín estará alzado (=1) si hay un número par de bits en el resultado:
 0 1 1 0 1 1 1 0 entonces OFF.....BAJADO
 0 1 1 0 1 0 1 0 entonces ONALZADO
- Banderín de semi-acarreo* → Ambos banderines quedan siempre bajados (=0) después de AND, OR ó XOR.
- Banderín de resta* → (Estos banderines son útiles si se está utilizando aritmética "BCD"). (Sistema decimal pero codificado en binario).

Uso de operaciones Booleanas sobre los banderines

Hay un uso especial de las operaciones Booleanas que resulta muy práctico -el caso del registro A operando sobre sí mismo.

AND A	A no cambia, el banderín de acarreo se baja.
OR A	A no cambia, el banderín de acarreo se baja.
XOR A	A se pone a 0, el banderín de acarreo se baja.

- Estas instrucciones son muy populares, porque sólo requieren un octeto para hacer lo que de otra manera exigiría dos, tal como LD A, 0.

El banderín de acarreo necesita a menudo ser bajado -eg. como cuestión de rutina antes de usar cualquiera de las operaciones aritméticas tales como:

ADC	sumar con acarreo
SBC	restar con acarreo.

y puede fácilmente hacerse mediante la instrucción AND A, sin alterar el contenido de ninguno de los registros.



RESUMEN

Hay tres operaciones lógicas que son útiles en lenguaje-máquina:

→AND (Ylaci6n)
→OR (Oliaci6n)
→XOR (Oleaci6n)

Solamente operan con números de 8 bits, y uno de esos números debe estar guardado en el registro A. El resultado de la operación siempre se encuentra en el registro A.

Observa que el significado de las operaciones en lenguaje-máquina es diferente al significado que tienen como "conectores" en una instrucción BASIC.

Las operaciones lógicas, consideran separadamente a los bits de los dos números, y por tanto, son útiles para manipular individualmente los bits.



Hasta ahora, hemos tratado sólo con números a una mano (8 bits), pero hemos comentado de hecho que la CPU también puede manejar números a dos manos (16 bits) en algunos casos.

Uno de los casos que ya hemos mencionado son los registros índices. Estos *píes* tienen 16 *dédos* (16 bits) y sólo operan con números de 16 bits. Además, sabemos que utilizando conjuntamente dos manos, podemos algunas veces *retener* un número de 16 bits. A estas manos que pueden ir conjuntamente, las llamamos parejas de registros. Son BC, DE y HL.

La CPU maneja los números de 16 bits de una forma muy *pareja* a la que tú o yo usaríamos con objetos pesados: necesitamos las *dos manos* y no somos muy propensos a manipular tales objetos, y la forma en que lo hacemos es lenta y limitada.

Examinémos los diferentes modos de direccionamiento de los que disponemos para los números de 16 bits.

Direccionamiento extendido inmediato:

LD rr, nn (o cualquier otra instrucción)

Es el equivalente del direccionamiento inmediato de un octeto. Simplemente ahora sirve para transferir datos en grupos de 16 bits.

Como regla general, las instrucciones que operan con números de 16 bits son más lentas y más largas que las correspondientes a 8 bits. Por ejemplo, mientras las instrucciones con direccionamiento inmediato de 8 bits tienen una longitud de 2 octetos (uno para la instrucción y otro para el número), la versión ampliada -esto es, para 16 bits- requiere 3 octetos.

El formato para el direccionamiento extendido inmediato es el siguiente:

Octeto 1	Instrucción	
Octeto 2	n1	Octeto de orden inferior del número
Octeto 3	n2	Octeto de orden superior del número

* Usamos esta forma de direccionamiento para establecer el contenido de una pareja de registros, por ejemplo, el puntero a una celda de memoria.



Direccionamiento de registros:

Puedes *recordar* que el direccionamiento de registros es el nombre que damos a una instrucción, donde el valor sobre el que operamos está contenido en uno de los registros.

Eso es cierto para las instrucciones con 16 bits, excepto que en el repertorio, sólo hay unas pocas instrucciones de esta clase. Están relacionadas primordialmente con las operaciones aritméticas y extremadamente limitadas en las combinaciones de registros que permite.

eg.

ADD HL, BC

Mencionaremos de nuevo, la *preferencia* que la CPU tiene para su *pareja* de registros HL. Es donde tiene los *músculos*; y algunas instrucciones sólo pueden ser efectuadas con esta pareja de registros. Sigue siendo cierto con las instrucciones aritméticas y lo comentaremos con detalle en un capítulo posterior.

Direccionamiento indirecto de registros

Es el nombre que damos a aquellas instrucciones en que el valor que manejamos está en la memoria, y la dirección de esa celdilla de memoria está *contenida* en un par de registros.

En el Z80, esta clase de direccionamiento, es como siempre utilizada con los registros *emparejados* HL.

eg.

JP (HL)

Direccionamiento extendido

Es *similar* en concepto al direccionamiento extendido indirecto de registros, exceptuando que el valor que deseamos no está contenido en una pareja de registros, sino en una pareja de celdillas de memoria.

eg.

LD HL, (nn)

Estando "nn" especificado en el momento de *redactar* el programa.



Ejercicio:

Usando el *montador* de código en lenguaje-máquina, mete los siguientes programas:

1. *Direccionamiento extendido inmediato*

```
010F00    LD BC, 15    ; cargar BC con el valor 15
C9        RET          ; volver a BASIC
```

Cuando se *pase* este programa, verás que el valor de *USR* al *volver* del programa en lenguaje-máquina es 15, justamente como lo definimos. Tén en cuenta lo limitado de esta forma de direccionamiento: debes especificar el valor del número en el programa.

2. *Direccionamiento de registros*

Añadiremos ahora una línea al programa anterior:

```
210040    LD HL, 4000H ; cargar HL con 16384
010F00    LD BC, 15    ; cargar BC con 15
09        ADD HL, BC   ; sumar los dos números
C9        RET          ; retornar
```

Si *kulas* este programa, obtendrás la misma respuesta que anteriormente, a saber, 15! ¿por qué? ¿No le hemos sumado 16384?

La respuesta es, que lo hemos hecho, pero *todo* ha sido en el registro HL, por lo que no podemos ver nada de eso. Para ver lo que sucede tenemos que añadir unas cuantas líneas más, como las siguientes:

3. *Direccionamiento extendido*

```
210040    LD HL, 4000H
010F00    LD BC, 15
09        ADD HL, BC
22647D    LD (7D64H), HL ; poner lo de HL en las celdillas
                                32100 y 32101
ED4B647D  LD BC, (7D64H) ; colocar en BC el contenido de
                                32100 y 32101
C9        RET
```

Este método de *traspasar* información de HL a BC, no se utilizará realmente al programar, ya que las instrucciones PUSH y POP son más eficaces; pero ilustra lo que necesita hacerse a veces para superar los limitados modos de direccionamiento que la CPU Z80 tiene.



des examinar las celdillas de memoria 32100 y 32101, usando la
ando "mem" para comprobar este programa.



En capítulos anteriores hemos visto cuan *ágil* puede ser la CPU, manipulando números a una mano, y hemos comentado la forma en que puede tratar números a dos manos.

La *habilidad* matemática de la CPU es tal, que puede realizar cálculos muy complejos con números grandes y con una sola mano. ¿Por qué molestarnos entonces con números a dos manos?

Habría ocasiones en que encontrarías imposible especificar todo lo que deseas con sólo números de 8 bits. Si tuviéramos que constreñirnos a la gama de 0 a 255, nuestro computador sería, desde luego, una máquina muy limitada.

El ejemplo más *palmario* de necesitar números de 16 bits, es al especificar la dirección de una celdilla de memoria. Hemos implícitamente admitido, que eso podría ser posible cuando comentábamos instrucciones como LD A, HL.

La forma lenta de hacer estas cosas, sería cargar cada registro de una pareja de registros, como hicimos en los ejercicios anteriores.

Afortunadamente para nosotros, hay algunas (pero sólo unas pocas) instrucciones en la pastilla Z80 que nos permiten manejar números de 16 bits. En este capítulo trataremos con las transferencias de números de 16 bits, mientras que el siguiente capítulo versará sobre la aritmética con 16 bits.

Especificando direcciones con números a 16 bits

Observa, por favor, que todas las direcciones deben especificarse mediante un número de 16 bits. No puedes especificar una dirección con sólo 8 bits, *incluso* aunque sean direcciones entre 0 y 255. En la forma en que la CPU funciona, eso no es una dirección a no ser que emplees dos octetos de 8 bits cada uno.

Ya lo hemos, implícitamente, admitido cuando usamos la instrucción

```
LD A, (nn)
```

Recuerda, además, que los números de 16 bits se guardan en parejas de registros, con el octeto mayor al principio (comprueba de nuevo el capítulo "Una ojeada sobre la CPU").

Al decir "HL" corresponde a H = alto (HIGH); L = bajo (LOW)

Guardando números de 16 bits en la memoria

Hay una *faceta* en el diseño del Z80, que es muy difícil de explicar y justificar: cuando se cargan números de 16 bits en la memoria, se utiliza el *convenio inverso* al empleado en las parejas de registros.



Instrucciones para operaciones de carga a dos manos

Nemónimo	Octetos	Tiempo	Efecto sobre los banderines					
			C	Z	PV	S	N	H
LD Par de reg. Número	3/4	10	-	-	-	-	-	-
LD IX, Número	4	14	-	-	-	-	-	-
LD IY, Número	4	14	-	-	-	-	-	-
LD (Dirección), BC o DE	4	20	-	-	-	-	-	-
LD (Dirección), HL	3	16	-	-	-	-	-	-
LD (Dirección), IX	4	20	-	-	-	-	-	-
LD (Dirección), IY	4	20	-	-	-	-	-	-
LD BC o DE, (Dirección)	4	20	-	-	-	-	-	-
LD HL, (Dirección)	3	16	-	-	-	-	-	-
LD IX, (Dirección)	4	20	-	-	-	-	-	-
LD IY, (Dirección)	4	20	-	-	-	-	-	-

Notación sobre los banderines:

- # indica que el banderín se ve afectado por la operación
- 0 indica que el banderín se pone a cero
- 1 indica que el banderín se pone a uno
- indica que el banderín no se ve afectado



• El octeto inferior siempre se almacena el primero en la memoria!

Consideremos la situación cuando colocamos el contenido de HL en la memoria:

ANTES:		CELDILLA	CONTENIDO
H	L	32000	00
		32001	00
		32002	00
01	02		

HL contiene el número 258 decimal = 0102H. Las celdillas de memoria están vacías (Lo que, como puedes adivinar, significa que contienen "00").

DESPUES:		CELDILLA	CONTENIDO
H	L	32000	02
		32001	01
		32002	00
01	02		

El convenio con los números de 16 bits guardados en la memoria, (y en los listados de un programa), es que el octeto inferior siempre se almacena primero.

No hay ninguna justificación para esa *decisión*; excepto decir, que así fue como los diseñadores del Z80 lo pensaron, y ahora tenemos que vivir con ello. Asegúrate que lees esto cuidadosamente, y que te familiarizas con esta inversión del orden, ya que probablemente sea la causa más importante de errores al programar:

- En los registros el octeto superior se almacena primero
- En la memoria y en los programas, el octeto inferior se almacena primero.

No es algo que pueda ser comentado e ignorado; ya que cada vez que tratas con una instrucción de 16 bits en código máquina, necesitarás pensar cuidadosamente sobre el *orden* de los octetos superiores e inferiores. Sin embargo, no debes sentirte *anonadado* por esto -la vida en el Z80 sería virtualmente imposible sin las instrucciones de 16 bits- y es simplemente un precio que tenemos que pagar.

* Puedes comprobar por tí mismo todo esto, utilizando el Montador de código en lenguaje-máquina y examinando después el contenido de la memoria, utilizando la comando "mem".



Cargando números de 16 bits

El grupo de *instrucciones de carga* de 16 bits en su forma más simple, consiste en anotar un número de 16 bits en una pareja de registros. El nemónimo general es

LD rr, nn

*De nuevo, estamos usando la notación de dos letras para indicar un número de 16 bits. "rr" significa cualquier pareja de registros. "nn" cualquier número de 16 bits.

Para aquellos de vosotros sin programa ensamblador -vosotros que manualmente y usando las **tablas** al final del libro, convertís los nemónimos en código- los comentarios sobre el orden de los números de 16 bits en la memoria se convierte en crucial.

Incluso si tenéis ensamblador, debéis ser conscientes de estas inversiones de orden para que podáis leer el código cuando reviséis el contenido de la memoria.

Veamos un ejemplo específico:

cargar HL con el número 258
el nemónimo para esta instrucción es LD HL, 0102H

El código correspondiente a "LD HL, nn" es como puedes comprobar al final del libro:

21 XX XX

lo que significa que debemos inscribir el número 0102H en lugar de "XX XX"; pero a causa de la regla de inversión, no lo tecleamos como 0102H sino como 0201H.

La instrucción apropiada será:

21 02 01

Como ejemplos de lo mostraremos como

21 02 01 LD HL, 0102H (= 258)

Al meter nuestros programas, debes familiarizarte muy

Al meter nuestros programas, debes familiarizarte muy

Al meter nuestros programas, debes familiarizarte muy

Cargando números de 16 bits

Así como somos capaces de cargar números de 16 bits directamente en una pareja de registros, podemos cargar también números de 16 bits directamente en los registros índices (que ambos son *pés* con 16 dedos como recordarás).

LD IX, nn
LD IY, nn



Podemos manipular información entre una pareja de registros y dos celdillas sucesivas de la memoria (es el equivalente en 16 bits de cargar la información desde un registro sencillo en una sola celdilla de memoria).

Las instrucciones generales son:

LD (nn), rr
LD (nn), IX
LD (nn), IY

Recuerda que los paréntesis son la abreviatura de "contenido de"; así que la última instrucción se leería como: "cargar en la celdilla de memoria con dirección nn, el valor del registro IY".

Como estamos tratando con números de 16 bits, *realmente* cargamos la celdilla de memoria especificada y la siguiente celdilla de memoria. No es necesario especificar ambas direcciones (porque la CPU *sabe* determinar fácilmente la dirección de la segunda celdilla). Pero seremos cuidadosos para evitar la confusión entre operaciones de 8 bits y operaciones de 16 bits.

La estructura *recíproca* de muchas de las instrucciones, también aparece aquí; y podemos además, cargar una pareja de registros o un registro indicial con lo que haya en una pareja específica de celdillas de memoria:

LD rr, (nn)
LD IX, (nn)
LD IY, (nn)

Ejercicio:

Sabemos -por el manual del Spectrum- que el comienzo de la parte libre de memoria puede obtenerse mirando el contenido de las celdillas 23653 y 23654.

En BASIC usamos la instrucción:

PRINT PEEK 23653 + 256 * PEEK 23654

Efectuaremos ahora la *misma tarea* utilizando lenguaje-máquina: (23653 = 5X65H)

ED 4B 65 5C	LD BC, (23653)
C9	RET

OBSERVA CON ATENCION QUE LOS NUMEROS SE HAN METIDO CON EL OCTETO INFERIOR PRIMERO Y QUE OBTENDRIAS UNA RESPUESTA TOTALMENTE ERRONEA SI LOS METIERAS DE OTRA FORMA.



Utilizamos el registro BC para obtener esta información porque, como *recordarás* muy bien, el valor de USR, es el contenido que la pareja de registros BC tiene al finalizar el programa en lenguaje-máquina.

Ten en cuenta que "LD BC, (NN)" es una instrucción de 4 octetos!

- Podemos usar programas similares para determinar el valor de cualquiera de las variables de dos octetos, que aparecen en el manual del Spectrum.

En el Spectrum, sabemos que una vez que un programa ha *finalizado*, la posición en que comienza la *zona libre* de memoria queda *preñijada*, por lo que sólo necesitamos determinarla *una vez* por cada programa.



RESUMEN

Podemos *traer* números de 16 bits, a cualquiera de las *parejas* de registros o a los registros *indiciales*, bien especificando el *número* de 16 bits, o especificando la *celdilla* de memoria donde se encuentra el número de 16 bits.

De manera similar, podemos *llevar* a la memoria un número de 16 bits, desde cualquiera de las *parejas* de registros o desde los registros *indiciales*.

El único punto a tener muy en cuenta, es el *peculiar* orden en que los números de 16 bits se almacenan en la memoria y por tanto, en el orden en que aparecen en los listados los números de 16 bits:

- El octeto inferior siempre se almacena primero!!!



Instrucciones para operaciones con la "pila"

<i>Nemónimo</i>	<i>Octetos</i>	<i>Tiempo</i>	<i>Efecto sobre los banderines</i>					
			C	Z	PV	S	N	H
PUSH Par de reg.	1	11	-	-	-	-	-	-
PUSH IX o IY	2	15	-	-	-	-	-	-
POP Par de reg.	1	10	-	-	-	-	-	-
POP IX o IY	2	14	-	-	-	-	-	-
LD SP, Dirección	3	10	-	-	-	-	-	-
LD SP, (Dirección)	3	20	-	-	-	-	-	-
LD SP, HL	1	6	-	-	-	-	-	-
LD SP, IX o IY	2	10	-	-	-	-	-	-

Notación sobre los banderines:

- # indica que el banderín se ve afectado por la operación
- 0 indica que el banderín se pone a cero
- 1 indica que el banderín se pone a uno
- indica que el banderín no se ve afectado



MANIPULANDO LA PILA

Recuerda la *imagen* que establecimos al comienzo del libro sobre la pila: es donde la CPU era capaz de guardar información sin tener que recordar la dirección de esa información en particular.

Una de las ventajas, posiblemente *inadvertida*, de las operaciones con la pila es que sólo podemos meter (PUSH) y sacar (POP) información a dos manos 16 bits. Así es, por que la pila está *primordialmente* diseñada para recordar direcciones y las direcciones se especifican como números de 16 bits.

Las instrucciones generales para meter en la pila son:

PUSH rr
PUSH IX
PUSH IY

y las instrucciones generales para sacar de la pila son:

POP rr
POP IX
POP IY

Son instrucciones excepcionalmente simples, y observarás que no se necesita especificar ninguna dirección.

Para las parejas *ordinarias* de registros -ie. no para los registros *indiciales*- estas instrucciones sólo tienen un octeto de longitud, y por tanto, se ocupa muy poco espacio al emplearlas en un programa.

Las instrucciones PUSH, son además no destructivas. Es decir, el registro de 16 bits, todavía contiene la *misma* información después de haberla PUSHADO.

Ten en cuenta que podemos *Pushar* cualquier pareja de registros y *Popar* cualquier pareja de registros; la pareja que saquémos, no necesita ser la misma que la que metimos.

Por ejemplo

PUSH BC
POP HL

El efecto de estas dos instrucciones *seguidas* es dejar el contenido del registro BC sin cambio, pero coloca en el registro HL lo que hubiera en el registro BC en el momento de efectuar la instrucción PUSH.

Así equivale realmente a una instrucción de la clase:

LD rr, rr'

-que estaba *faltando* claramente en el grupo de carga en 16 bits que acabamos de ver.



Como cada instrucción PUSH y POP, para una pareja de registros sólo ocupa un octeto, el *coste*, en términos de ocupación de memoria, *no es mucho*.

- El otro *beneficio* extra es, que así somos capaces de PUSHAR o POPAR la pareja de registros AF! Es una de las pocas instrucciones en que AF es considerado una pareja de registros -pero es visiblemente *sensata* porque habrá muchas veces que queramos preservar el contenido de los banderines.

Por tanto, puedes mandar PUSH AF (en realidad *anotar* lo que A y F tienen), realizar cálculos que pueden afectar a los banderines con efectos *colaterales e indeseables*, y luego mandar POP AF dejando los banderines como estaban al principio.

Cambiando de sitio el stack

Como sabes, la *fuera* real de estas instrucciones, está en que no tenemos que pensar en qué direcciones estan los números que hemos PUSHADO o que POPAMOS.

Seguramente, estés de acuerdo en que no tiene *sentido* que la misma área de memoria sirva como pila cuando tienes 16K de memoria que cuando tienes 48K. La forma en que la CPU realmente *guarda nota* de la dirección de la pila es por medio del llamado "puntero de la pila" que se puede ver como un registro de 16 bits. Lo hemos mencionado brevemente en nuestros comentarios sobre los registros; pero no en las instrucciones LOAD, etc. porque es un registro *especial* -que no puede ser tratado de la misma forma que los otros registros.

- ★ Lo que *solemos* querer con el puntero de la pila es definir la posición dentro de la memoria en que queremos que comience; y eso, es exactamente la clase de instrucciones de las que disponemos:

LD SP, nn
LD SP, (nn)
LD SP, IX
LD SP, IY

Puedes examinar la pila del Spectrum usando la *comanda* "mem" del programa Montador de código en lenguaje-máquina EZ, y mirando los últimos 30 - 40 octetos antes de RAMTOP (el *tope* de memoria).

Pero *no cambies* el contenido de las celdillas de la pila!!

Casi con seguridad, cualquier cambio hará que tu Spectrum *se la pegue* -la pantalla se quedará en blanco y tendrás que volver a encender de nuevo. Eso ocurre, porque el sistema operativo coloca en la pila, mucha de la información que precisa y los cambios provocarán que se quede *colgado*.

Por la misma razón, no intentes manipular la posición del puntero de la pila, a menos que estés seguro de lo que estás haciendo.



AVISO

En un programa bien organizado, el número de POPS y PUSHES deberá siempre *balancearse*, sin importar el camino que el programa vaya a seguir. Cualquier *descompensación*, puede conducir a resultados extraños.

Ejercicio:

Podemos utilizar estas instrucciones para examinar la dirección en que la subrutina *USR reside*, si POPAMOS el valor de la pila en el registro BC. El siguiente programa nos muestra cómo:

C1	POP BC	;	<i>Obtiene la dirección en BC</i>
C5	PUSH BC	;	<i>La vuelve a poner en la pila</i>
C9	RET	;	<i>Para balancear antes de volver</i>



Instrucciones para aritmética a dos manos

Nemónimo	Octetos	Tiempo	Efecto sobre los banderines					
			C	Z	PV	S	N	H
ADD HL, Par de registros	1	11	#	-	-	-	0	?
ADD HL, SP	2	11	#	#	#	#	0	?
ADC HL, Par de registros	2	15	#	#	#	#	0	?
ADC IX, SP	2	15	#	-	-	-	0	?
ADD IX, BC o DE	2	15	#	-	-	-	0	?
ADD IX, IX	2	15	#	-	-	-	0	?
ADD IX, SP	2	15	#	-	-	-	0	?
ADD IY, BC o DE	2	15	#	-	-	-	0	?
ADD IY, IY	2	15	#	-	-	-	0	?
ADD IY, SP	2	15	#	-	-	-	0	?
SBC HL, Par de registros	2	15	#	#	#	#	1	?
SBC HL, SP	2	15	#	#	#	#	1	?

Notación sobre los banderines:

- # indica que el banderín se ve afectado por la operación
- 0 indica que el banderín se pone a cero.
- 1 indica que el banderín se pone a uno.
- indica que el banderín no se ve afectado.
- ? indica que el efecto sobre el banderín no es conocido



Uno de los beneficios de poder jugar con 16 bits en lo que realmente es un procesador de 8 bits, es que podemos utilizar los 16 bits para especificar direcciones o para realizar cálculos que involucran números enteros hasta 65.355 (o en la gama -32.768 a +32.767, si tratamos con números negativos).

Ahora se ve claramente por qué en algunos microcomputadores anteriores, como el primitivo Sinclair ZX80, toda la aritmética en BASIC estaba limitada a números enteros comprendidos entre -32.000 a +32.000.

Pero incluso, aunque podamos realizar algo de aritmética a dos manos, el título de este capítulo ya da una pista de lo que viene -la aritmética a dos manos es un poco chapucera comparada con la aritmética a una mano. La gama de opciones es bastante más cerrada!

Pareja de registros preferente

De la misma forma que el registro A es el favorito en la aritmética de 8 bits, hay una pareja de registros favorita en la aritmética de 16 bits y es la pareja HL.

Pero este favoritismo no es tan pronunciado como en el caso de los 8 bits, de manera que no omitiremos el nombre del par de registros.

Suma:

Las sumas son bastante directas:

```
ADD HL, BC
ADD HL, DE
ADD HL, HL
ADD HL, SP
```

Y no necesitan más comentarios que ...e o es todo!

Ves que no es posible añadir un número cualquiera a HL -eg. no está permitido "ADD HL, nn". Para realizar esta clase de cálculo necesitaríamos hacer:

```
LD DE, nn
ADD HL, DE
```

Pero si consideras que esto ata cuarlo de los registros de 8 bits (y sólo tienes siete), te darás cuenta que NO es algo que puedas hacer muy a menudo.

Tampoco hay suma entre HL y los registros índices. Además, recordará que no hay instrucción que cargue el contenido de IX ó IY en la pareja BC ó DE; de forma que la única manera de efectuar esa adición sería algo así como:

```
PUSH IX
POP DE
ADD HL, DE
```



El otro punto a observar es el registro "SP" (el puntero de la pila). Es una de las poquísimas operaciones en que es tratado como un auténtico registro, pero obviamente no puedes utilizarlo como una variable!

Piensa lo que pasaría a toda la información que has apilado, si varies a voluntad el contenido del puntero de la pila.

Efecto sobre los banderines

La aritmética de 16 bits es donde realmente el banderín de acarreo tiene toda su *sustancia*. Porque, como puedes ver en la tabla del comienzo del capítulo, solamente otro banderín se ve afectado por estas instrucciones, y es el de resta o negación (y todo lo que decimos con este banderín es que la instrucción ADD no es una resta).

El banderín de acarreo estará *alzado* si ha habido un *acarreo* desde el bit superior del *registro H* -cualquier acarreo desde el *registro L* queda *automáticamente* incluido en el cálculo.

Suma con acarreo:

Dado que los 16 bits también son de naturaleza limitada, somos capaces de *encadenar* sumas, igual que en el caso de 8 bits. La instrucción "suma con acarreo" (ADC) opera de manera similar a la de suma y con la misma gama de parejas de registros:

ADC HL, BC
ADC HL, DE
ADC HL, HL
ADC HL, SP

Resta con 16 bits:

La resta con 16 bits, también es una operación directa; pero *no* hay resta sin acarreo. Si no estás seguro del estado del banderín de acarreo, asegúrate que tu programa incluye una línea que lo baje antes de cualquier operación de resta:

SBC HL, BC
SBC HL, DE
SBC HL, HL
SBC HL, SP

(Esta última instrucción tiene aplicaciones *obvias*: coloca HL al final de la memoria ocupada por *todo* tu programa, resta SP, y el resultado (negativo) será la cantidad de memoria libre.



¿Puedes escribir un sencillo programa que lo haga?

Mira al final del capítulo para confirmar tu solución.

Efecto sobre los banderines de la aritmética con acarreo

Habrás notado que con 16 bits, hay otros tres banderines que se ven afectados por la suma y la resta con acarreo, y que no se veían afectados por la suma sin acarreo.

Son el banderín de **cero**, de **signo** y el de **rebose**. Cada uno de ellos se alza o baja según el resultado de la operación.

Aritmética con el registro indicial

Los registros indiciales están exclusivamente limitados a la adición sin acarreo!

Aún más, la *gama* de registros que pueden añadirse al registro indicial es extremadamente limitada:

- Añadir el par de registros BC ó DE.
- Añadir el registro indicial a él mismo
- Añadir el puntero de la pila.

Solución al ejercicio de memoria que deja libre un programa:

El final del espacio de memoria que el programa ocupó, está definido por el contenido de la *celdilla* llamada STKEND. En el manual del Spectrum te dicen que son la 23653 y la 23654!

Obviamente, si cargamos HL con el contenido de esa celdilla, ya estamos a medio camino:

```
LD HL, (STKEND)
```

luego le restamos el puntero de la pila (SBC HL, SP ?)

Por culpa del acarreo, necesitamos garantizar previamente que está en Off. La forma más fácil de lograrlo es con la instrucción "AND A", que ya hemos comentado anteriormente en este libro.

```
AND A  
SBC HL, SP
```

Apúntate un notable si ya sabías que tenías que tener en cuenta el acarreo, pero no sabías como hacerlo, y un *"insufi"* si olvidaste todo lo referente al acarreo.

Dado que el puntero de la pila está más *arriba* que la cima del programa que hay en memoria (o tú sabrás dónde, si estás haciendo *diabluras*) el resultado será negativo.

Prosigamos ahora, hasta obtener como una *cantidad* positiva el número de octetos que quedan, utilizando el registro BC (lo mismo nos valdría con el registro DE). Primero queremos traspasar HL a BC, pero como no hay LOAD que haga eso,



necesitaremos meterlo y sacarlo de la pila:

```
PUSH HL
POP BC
```

HL todavía tiene la misma información que antes, así que HL es igual a BC.

Para hacer que HL sea igual a *menos* BC, resta BC de HL dos veces! (pero no olvides que el acarreo se acaba de alzar a causa de la resta anterior, así que *bájalo* otra vez):

```
AND A
SBC HL, BC
SBC HL, BC
```

HL ahora contiene el valor *negativo* de lo que contenía antes -ie. el número positivo de octetos que quedan libres.

Necesitamos ahora devolver ese número al par de registros BC para que nos lo de como resultado de la función USR. Para *devolver* HL a BC:

```
PUSH HL
POP BC
```

y finalmente a *regresar* de la función USR.

```
RET
```

¡Agarraste todo esto correctamente?

No lo sueltes (que no da calambre) y con una pila tan manejable.



LAZOS Y SALTOS

Los *lazos* y los *saltos* son los que dan a un computador, toda su potencia. Una vez que domines cómo hacer decisiones y cómo ejecutar diferentes trozos de un programa según el resultado de los cálculos previos, es que ...vaquero: estás ganando este rodeo!

Esta *libertad* puede causarte problemas: crear programas que son difíciles de seguir, y casi imposibles de depurar.

Te sugerimos muy mucho que diseñes tus programas cuidadosamente antes de escribir nada en código máquina; y por eso es por lo que hemos incluido el capítulo "Planteando un programa en lenguaje-máquina". Te lo resaltamos ahora porque los lazos y los saltos son los que te apartan siempre de un buen programa.

Equivalente en lenguaje-máquina a GOTO

En BASIC, estás familiarizado con la instrucción "GOTO", por la que haces que el programa vaya a la instrucción con el número de línea que mencionas después de GOTO.

Nada hay más simple que implementar en lenguaje-máquina: simplemente especificas la celdilla de memoria en donde te gustaría que la CPU recabara la siguiente instrucción y ya tienes medio camino recorrido. El resto es pan comido.

La más simple instrucción es "Saltar a" ...(en inglés "JUMP").

JP XX XX
JP (HL)
JP (IX)
JP (IY)

Cada una de estas instrucciones también puede hacerse que dependa del estado de alguno de los *banderines*, por ejemplo el de acarreo. Esta instrucción de salto condicional es:

JP cc, nn

siendo 'cc' la condición que debe cumplirse. Si tenemos

JP Z, 0000

por ejemplo, se leería "saltar -SI el banderín de cero está alzado- la dirección "0000".

(Esta es la dirección a la que el Spectrum salta nada más encenderle; y como tal, un salto a cero puede utilizarse en un programa en lenguaje-máquina, si quieres *limpiar* toda la memoria y empezar de nuevo con "K").

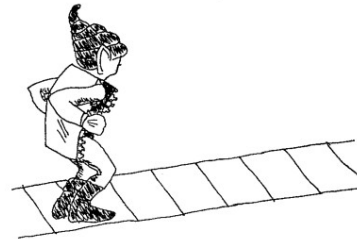
Observa ahora que la CPU no permite ninguna equivocación. Si tú le dices "jump" saltará. Dado que cualquier código puede ser equivalente a una instrucción, la CPU no le importa si al saltar caes en medio de los datos, o en el *segundo* octeto de una instrucción de dos octetos. Leerá el octeto en la dirección que le hayas dicho, *supondrá* que es el comienzo de una instrucción, y seguirá obedeciéndola.



La forma en que la CPU maneja las instrucciones de salto es *bastante* simple: tiene un pequeño *contador*, llamado el *contador de programa*, que le dice dónde tiene que buscar la siguiente instrucción a cumplimentar. En el transcurso normal de un programa (es decir, sin saltos) la CPU mira la instrucción a obedecer, y *añade* tantos octetos como haya en esa instrucción al *contador de programa*.

Así *pués*, si encuentra una instrucción de dos octetos, suma dos; y una de cuatro octetos le hará sumar cuatro al contador de programas, y demás.

Cuando *tropieza* con una instrucción de salto, simplemente *sustituye* el contenido del contador del programa con el valor que tú hayas especificado. Por eso es, por lo que no puedes permitirte que se te *cuelé* ningún error.



Salto largo y salto corto

Podemos describir las instrucciones anteriores como de un "salto largo", porque una dirección de 16 bits nos permite saltar a donde quiera que la pastilla Z80 sea capaz de ir; y puede ir muy lejos al usar dos octetos para la dirección.

La desventaja de un salto largo, es que:

- A. *A menudo no queremos saltar tan lejos, pero si quisiéramos usar una instrucción de menos octetos.*
- B. *No podemos reubicar fácilmente el programa en otra parte de la memoria porque estamos empleando direcciones absolutas (ie. no relativas).*

Fue primordialmente, para superar estas dos desventajas que se introdujo el "salto corto". Nos referimos a él como un salto relativo, y nos permite saltar hasta 127 octetos *delante* de nuestra posición actual o -128 octetos *detrás* de esa posición. ie. la distancia que se salta puede especificarse en un octeto!

Instrucción de salto relativo

JR d

siendo "d" el desplazamiento relativo, ie. que toma donde estamos, como *punto de partida*.

Podemos hacer también que el salto relativo *dependa* de alguna condición, tal como acarreo alzado, o banderín de cero bajado, etc.

Estos saltos condicionales se escriben:

JR cc, d

siendo cc la condición que ha de *cumplirse* para que *salte*.

El valor del desplazamiento "d" se suma al contador de programa.

Esto significa que se toma el valor *presente* del contador de programa y se le *añade* el valor relativo que has especificado. El valor que especificas puede ser positivo -salto hacia delante-, o negativo -salto hacia atrás. Si vuelves a estudiar nuestro capítulo sobre números negativos, te darás cuenta que los saltos relativos están limitados a la gama -128 a +127.

Ten en cuenta que cuando la CPU está ejecutando una instrucción de salto relativo, el contador de programa ya está *apuntando* a la siguiente instrucción. La que se ejecutaría si no se cumpliera la condición.

Nos explicamos, cuando la CPU tropieza con "JR" sabe que es una instrucción de dos octetos, y añade inmediatamente dos al contador de programa -que por tanto, está apuntando a la instrucción *que sigue a la de* salto relativo-!!!



Eg. En un programita como

Celdilla	Código
32000	ADD A, B
32001	JR Z, 02H
32003	LD B, 0
32005	OTRA LD HL, 4000H

A continuación, te describimos lo que la CPU haría con el programita si *ignorase* la instrucción de salto que hay en la celdilla 32001 (ie. si estuviera bajado el banderín de cero:

- ★ Cargar el octeto de la 32000, ya que el contador de programa tiene 32000.
Dado que es una instrucción de un solo octeto, pondrá el contador de programa a 32001 y luego ejecutará la instrucción.
- ★ Cargar el octeto especificado por el contador de programa (32001).
Este octeto es parte de una instrucción de dos octetos, así que suma dos al contador de programa con lo que éste tendrá 32003. Recabar el siguiente octeto para completar la instrucción vigente y luego ejecutar esa instrucción.
- ★ Cargar el octeto especificado por el contador de programa (32003).
Este octeto es parte de una instrucción de dos octetos, así que añade dos al contador de programa (ahora igual a 32005) y recaba el siguiente octeto para completar la instrucción. Luego ejecuta la instrucción.
- ★ Seguir.

En la celdilla 32001 el programa encuentra una instrucción condicional de salto relativo. Si el banderín de cero no está alzado como en el ejemplo anterior, la CPU no hace nada más que sumar dos.

En general, la CPU ejecuta esa instrucción de salto como sigue:

- Si el banderín de cero está alzado, suma otras dos más al contador de programa -esto lo haría = 32005- y se va a guipar lo que allí haya.
- Si el banderín de cero no está alzado, no sumaría nada extra -el contador de programa permanecería = 32003-, y allí que se iría.

En otras palabras, el salto relativo nos permite *olvidarnos* de la instrucción "LD B, 0" en ciertos casos.

Esto explica además, por qué en estas instrucciones hay indicados dos tiempos en la columna de "tiempo empleado". Se tarda menos tiempo en no hacer nada, que en calcular el nuevo valor del contador de programa.

- La CPU, por tanto, ejecutará la instrucción en 32003, o la instrucción 32005, dependiendo del estado del banderín de cero.

Además te *reiteramos* que es posible hacer saltos relativos negativos, ie. *retrocesos*.



Ejercicio:

Como el salto relativo es una instrucción de 1 octeto, y el contador de programa está apuntando a la instrucción que viene *detrás* del salto relativo, ¿cuál sería el efecto de una instrucción que mandara:

JR -2

Lenguaje-máquina para lazos "For ...Next":

Estoy seguro que ya has dominado los lazos "for ...next..." del lenguaje BASIC:

```
FOR I = 1 to 6
LET C = C + 1
NEXT I
```

Hay un lazo similar en lenguaje-máquina, pero adopta una forma diferente. Consideremos cómo podemos implementar un lazo de 6 vueltas en lenguaje máquina, utilizando las funciones aritméticas y el salto relativo:

LAZO	{	LD B, 1	; Poner el contador a 1
		LD A, 7	; Máximo de cuenta + 1
		INC C	; C = C + 1
		INC B	; Incrementar el contador
		CP B	; ¿Es B = A?
		JR NZ, LAZO	; Si no lo es, de nuevo a LAZO.

Esto funcionará, pero observa lo siguiente:

Tenemos *trabajados* dos registros, uno para incrementar, y otro para fijar el máximo de la cuenta; y la instrucción que incrementa el contador no afecta a ninguno de los banderines al concluir. Sería mucho mejor si hubiéramos contado hacia abajo.

○ Sabemos que lo que queremos hacer es repetir 6 veces, así que ¿por qué no poner B a 6 y contar hacia abajo?

Eso nos daría:

LAZO	{	LD B, 6	; coloca en el contador el máximo
			de cuenta.
		INC C	; C = C + 1
		DEC B	; decrementa el contador
		JR NZ, LAZO	; si no ha acabado vuelve a LAZO.

Como puedes ver, es una forma mucho más eficaz de hacer las cosas.

La pastilla Z80 tiene una instrucción especial que combina las dos últimas instrucciones anteriores. Esa instrucción se escribe como:

DJNZ d



La anterior instrucción se lee como "decrementar (B) y saltar si no es cero". (La "d" es el desplazamiento relativo). Esta instrucción sólo tiene dos octetos y nos *ahorra*, por tanto, un octeto en la codificación anterior.

A causa de la existencia de esta instrucción especial, el registro "B" se utiliza habitualmente como contador.

Las limitaciones de la instrucción DJNZ, es que sólo puede contar hasta 256. Sin embargo, podemos *anidar* instrucciones DJNZ, si se precisa un lazo con más vueltas:

	LD B, 10H	; B = 16
LMAYOR	PUSH BC	; guardar el valor de B
	LD B, 0	; hacer B = 256
LMENOR	; cálculo cualquiera
	
	DJNZ LMENOR	; ¿hecho 256 veces?
	POP BC	; recoger de nuevo el valor de B
	DJNZ LMAYOR	; hecho 16 veces el lazo mayor.

Ejercicio:

Intenta escribir en un trozo de papel lo que aparecería en cada registro después de cada instrucción del programa anterior, pero habrás de coger papel en *cantidad*.

Lazos de espera:

Hay veces, en lenguaje-máquina, que las cosas *suceden* tan rápidamente que es necesario *esperar* durante un momento. Los ejemplos que nos vienen a la mente son al enviar información a una cassette (los *pitidos* tienen que espaciarse suficientemente para que puedan posteriormente ser leídos), o enviar información a una máquina de escribir (habría que imprimir a *miles* de caracteres por segundo).

Es, por tanto, útil establecer lazos de espera, utilizando la instrucción DJNZ:

	LD B, cuenta
ESPERA	DJNZ ESPERA

La instrucción "DJNZ ESPERA" hará que la CPU salte *hacia atrás*, -hasta la propia instrucción DJNZ-, tantas veces como sean precisas para que el registro B llegue a *cero*, antes de proseguir con nuevas instrucciones.

Esto te da además, la respuesta al ejercicio en que preguntábamos lo que pasa cuando escribes JR -2. El tiempo de espera es función del valor que cargues en B.



Instrucciones para el grupo de llamada y vuelta (CALL y RETURN)

<i>Nemónimo</i>	<i>Octetos</i>	<i>Tiempo</i>	<i>Efecto sobre los banderines</i>					
			C	Z	PV	S	N	H
CALL dirección	3	17	-	-	-	-	-	-
CALL cc, dirección	3	10/17	-	-	-	-	-	-
RET	1	10	-	-	-	-	-	-
RET cc	1	5/11	-	-	-	-	-	-

Mediante "cc", se indica la condición que ha de cumplirse para que se ejecute la instrucción. Las condiciones que pueden usarse son:

<i>Banderín</i>	<i>Abreviatura</i>	<i>Significado</i>
Acarreo	C	Acarreo alzado (= 1)
	NC	Acarreo bajado (= 0)
Cero	Z	Cero alzado (= 1)
	NZ	Cero bajado (= 0)
Paridad	PE	Paridad par (= 1)
	PO	Paridad impar (= 0)
Signo	M	Signo menos (= 1)
	P	Signo positivo (= 0)

Banderines afectados:

Observa que ninguno de los banderines se ven afectados por las instrucciones CALL y RETURN.

Tiempos:

Cuando se muestran dos tiempos, el más corto indica el caso en que no se cumple la condición.



USO DE SUBROUTINAS EN TUS PROGRAMAS EN LENGUAJE-MAQUINA

El uso de subrutinas es tan fácil en la programación en lenguaje-máquina como lo es en los programas BASIC, si no más. De hecho, recuerda que usar la función "USR" en BASIC, es realmente *llamar* a una subrutina: incluso recordarás que necesitas tener una instrucción RETURN para acabar. Por tanto, es muy fácil que *compruebes* ciertas rutinas independientemente de tu programa principal.

La diferencia mayor con la que te encaras al implementar subrutinas en lenguaje-máquina, es que necesitas saber la dirección donde comienza la subrutina. Esto puede causarte problemas si guardas las rutinas en lenguaje-máquina en una tabla de variables porque la dirección de esas variables no es necesariamente constante. Significa también que los programas en lenguaje-máquina que usan subrutinas, no pueden fácilmente ser trasladados a distintas posiciones en la memoria, ie. reubicados.

Las subrutinas pueden llamarse también condicionalmente, lo que es el equivalente en lenguaje-máquina de la sentencia BASIC:

IF (condición) THEN GOSUB (número de línea)

Se debe tener *cuidado* en las subrutinas para no afectar ninguno de los banderines ni registros que se necesitan en comparaciones subsiguientes. Tenlo presente si no quieres *desvirtuar* las sentencias CALL que haya después de volver al punto donde *citaste* a la subrutina. (Y *llamar, citar, mencionar...* todo significa lo mismo).

Las únicas condiciones permitidas para las llamadas condicionales son:

- -Banderín de acarreo
- -Banderín de cero
- -Banderín de paridad (también de rebose)
- -Banderín de signo.

Recuerda que todos estos banderines se alzan o bajan según la última instrucción que afecte a ese banderín en particular.

Es por tanto, muy buena práctica colocar las instrucciones CALL o RETURN inmediatamente después de la instrucción que fija los banderines.

eg.

LD	A, (Número)
CP	1
CALL	Z, UNO
CP	2
CALL	Z, DOS
CP	3
CALL	Z, TRES

La rutina anterior te permite *saltar* a diferentes puntos *según* el valor almacenado en la celdilla llamada NUMERO; pero observa que presupone que las subrutinas no cambian el valor que hay en el registro A!!!
¿Por qué?



Si sabes que hay *sólo tres posibilidades* para el valor guardado en la celdilla número, es posible redactar una rutina más corta:

LD	A, (NUMERO)	
CP	2	
CALL	Z, DOS	; A = 2
CALL	C, UNO	; A < 2 → A = 1
CALL	TRES	; A > 2 → A = 3

Lo podemos hacer porque la instrucción CP 2 afecta tanto al banderín de acarreo como al de **cero**, y la instrucción de llamada no afecta a *ninguno* de los banderines.

Similarmente, el uso de un **regreso condicional** desde una subrutina es muy útil.
(*Pero no está considerado por muchos, como una buena práctica de programación*).



Instrucciones de comparación y traslado de bloques

Nemónimo	Octetos	Tiempo	Efecto sobre los banderines					
			C	Z	PV	S	N	H
LDI	2	16	-	-	#	-	0	0
LDD	2	16	-	-	#	-	0	0
LDIR	2	21/16	-	-	0	-	0	0
LDDR	2	21/16	-	-	0	-	0	0
CPI	2	16	-	#	#	#	1	#
CPD	2	16	-	#	#	#	1	#
CPIR	2	21/16	-	#	#	#	1	#
CPDR	2	21/16	-	#	#	#	1	#

Notación sobre los banderines:

- # indica que el banderín se ve alterado por la operación
- 0 indica que el banderín se pone a cero
- 1 indica que el banderín se pone a uno
- indica que el banderín no se ve afectado.

Tiempos:

En las instrucciones *repetitivas*, los tiempos que se indican son para cada ciclo. El tiempo más corto que se indica es para el caso de la instrucción que *termina* la repetición -eg. para CPIR, bien cuando BC = 0, o bien cuando A = (HL).



OPERACIONES CON BLOQUES

A estas alturas, deberías estar muy familiarizado con el lenguaje que tu computador comprende -es muy parecido a cuando se aprende un idioma extranjero. Cuando puedes pensar en ese lenguaje es cuando lo has dominado.

Este capítulo cubre el último grupo de instrucciones utilísimas -los siguientes capítulos tratan con instrucciones que son muy convenientes y que en algunas circunstancias resultan valiosas; pero en términos generales ya deberías estar capacitado para escribir programas en lenguaje-máquina con las que ya conoces. Asegúrate, sin embargo, que comprendes el capítulo sobre cómo plantear un programa en lenguaje-máquina.

Las instrucciones que comentamos en este capítulo, son por su propia naturaleza potentes como para saltar rascacielos en un brinco de gigante, y más rápidas que una ráfaga de ametralladora -en otras palabras, instrucciones que manejan un bloque de memoria, en lugar de simples octetos.

Comencemos por la más simple de ellas:

CPI

Con lo que ya sabes del lenguaje del Z80, seguro que has reconocido inmediatamente que es un miembro de la familia de las "comparaciones"; y de hecho, es una comparación ampliada.

Se lee como COMPARAR e INCREMENTAR. (Recordarás que sólo podemos comparar algo con el contenido del registro A, y por tanto, no lo mencionamos en la instrucción).

CPI compara "A" con (HL) e incrementa HL. Esto significa que después de esta operación, HL ya está apuntando a la siguiente celdilla y presto para repetir la operación.

Con esta instrucción somos capaces de escribir una rutina que escrute toda la memoria en busca de un determinado octeto, como sigue:

ESCRUTA	CPI
	JR NZ, ESCRUTA

* De esta forma, a menos que encontremos una concordancia (con lo que el banderín de cero quedará alzado como en todas las instrucciones de comparación), el procesador continuará buscando.

Desafortunadamente, no ha sido una buena idea porque a no ser que encontremos una concordancia el programa nunca acabará!

Afortunadamente, también los diseñadores del Z80 pensaron en ello y la instrucción CPU también decreuenta la pareja BC!

Podemos, por tanto, elegir voluntariamente la longitud del bloque que deseamos sea escrutado, permitiéndonos así que siempre acabe el escrutinio.



Supongamos que la longitud del bloque en que vamos a buscar es menor de 255, de manera que la cuenta pueda guardarse sólo en el registro C.

Escribiríamos:

ESCRUTA	CPI
	JR Z, SIENCONTRO
	INC C
	DEC C
	JR NZ, ESCRUTA
NOENCONTRO

SIENCONTRO

Obviamente, realizaremos una rutina diferente si la longitud del bloque tiene más de 255 octetos.

Observa cómo usamos las instrucciones INC y DEC para comprobar si $C = 0$ (bloque completo). Estas dos instrucciones sólo precisan de un octeto cada una, y como ambas afectan al banderín de cero, el efecto neto de las dos es alzar el banderín sólo cuando C era cero antes de ellas (y desde luego, esta codificación no altera ninguno de los otros registros).

- Además, si deseáramos *escrutar* un bloque de memoria comenzando desde la parte superior, usaríamos la instrucción CPD que se lee como COMPARAR y DECREMENTAR. El decremento se refiere obviamente a HL, y el efecto sobre BC sigue siendo el mismo!

Todavía más potentes que éstas dos instrucciones, son las verdaderamente *supermanes*:

CPIR
CPDR

que se leen como "comparar, incrementar y repetir"
y "comparar, decrementar y repetir"

Estas instrucciones de sólo dos octetos son increíblemente potentes:

Permiten que la CPU continúe automáticamente escrutando un bloque de memoria hasta que se encuentra una concordancia, o se alcanza el final del bloque. (Naturalmente tendremos que especificar A, HL y BC antes de dictar la instrucción, pero incluso así es una codificación maravillosamente económica).

Dado que las instrucciones pararán con dos posibilidades (ie. porque han encontrado una concordancia dentro del bloque, o ninguna concordancia en todo el bloque), tendremos que colocar algunas instrucciones al final para diferenciar entre estas dos posibilidades.

Has de ser consciente sin embargo,
que no importa la velocidad del lenguaje-máquina
para que CPIR y otras instrucciones similares
puedan ser tremendamente lentas!!!



CPIR, por ejemplo, requiere 21 ciclos por cada octeto que se compara. Sabemos que hay 3.500.000 ciclos en cada segundo, pero incluso así significa que escribir 3.500 octetos requiere 1/50 de segundo. Lo que puede no parecer un tiempo muy largo, pero si consideras que la imagen de pantalla se *proyecta* cada 1/50 de segundo, aproximadamente, te darás cuenta que puede ser significativo.

Las otras operaciones con bloques siguen líneas ajedrecistas de *mueve, mate*:

Son:

LDI LDIR
LDD LDDR

Son parte, obviamente, de la familia de "carga" y se leen:

- * - Cargar e incrementar
- * - Cargar, incrementar y repetir
- * - Cargar y decrementar
- * - Cargar, decrementar y repetir

Empezando por la más simple, LDI es realmente una combinación de las siguientes acciones:

Cargar (DE) con (HL)
Incrementar DE, HL
Decrementar BC

Observa que ésta es la única instrucción que trasvasa de una celdilla de memoria (HL) a otra (DE), sin tener que cargar previamente el contenido con un registro.

El uso del registro "DE" como dirección de destino es muy *inteligente*, así nunca olvidarás cuál es el registro que contiene la dirección DEstino. Obviamente, HL contiene la dirección origen del trasvase.

La instrucción simétrica LDD es exactamente la misma, excepto en que HL y DE se *decrementan* a medida que prosigue la carga. La diferencia entre LDI y LDD es más *importante* cuando los dos bloques (donde está la información y donde va a estar la información) se *solapan*. Supongamos que utilizamos esta instrucción en una aplicación de *tratamiento de textos*, y queremos eliminar una palabra de una frase:

EL GRAN PERRO CASTAÑO SE LANZO HACIA LA ZORRA.
1 3 5 7 9 1 3 5 7 9 1 3 5 7 9 1 3 5 7 9 1 3 5

- Si queremos eliminar la palabra "castaño", todo lo que necesitamos hacer es *mudar* de sitio el resto de la frase que hay a la derecha.

DE = destino → lugar 15
HL = origen → lugar 23
BC = cuenta → 24 letras.

Comencemos con LDI: después de efectuada una vez la instrucción, tenemos:

original = ➡ EL GRAN PERRO CASTAÑO SE LANZO HACIA LA ZORRA.
mudamos una letra
y el nuevo es: ➡ EL GRAN PERRO SASTAÑO SE LANZO HACIA LA ZORRA.

y HL = 24, DE = 16, BC = 23.

Después de dos instrucciones más, habrá:

➡ EL GRAN PERRO SE TAÑO SE LANZO HACIA LA ZORRA.

y después de que las instrucciones se hayan completado, tendremos:

➡ EL GRAN PERRO SE LANZO HACIA LA ZORRA.A ZORRA.

(Si hubieras querido que el trozo después del punto quedara en blanco, se podría haber logrado, añadiendo espacios al final de la frase original e incrementando BC a 32).

- Si ahora queremos invertir el proceso y devolver la palabra "castaño" a la frase, no podemos usar simplemente "LDI" de nuevo, porque escribiremos encima de la información que queremos desplazar.

eg.

DE = destino = lugar 23
HL = origen = lugar 15
BC = cuenta = 24 letras.

Después de la instrucción, tendríamos:

➡ EL GRAN PERRO SE LANZOSHACIA LA ZORRA.A ZORRA.

Después de 6 instrucciones tendríamos:

➡ EL GRAN PERRO SE LANZOSE LAN LA ZORRA.A ZORRA.

y así sucesivamente ¿perfectamente?. Mira que con otras tres ya tendríamos:

➡ EL GRAN PERRO SE LANZOSE LANZOS ZORRA.A ZORRA.

y comprueba que acabaríamos con:

➡ EL GRAN PERRO SE LANZOSE LANZOSE LANZOSE LANZ

El problema es que ya habíamos escrito sobre la información que íbamos a transferir antes de sacar la que allí había. Puedes verificar eso moviendo una letra cada vez, por tí mismo y a mano.



Es mejor, por tanto, usar la instrucción LDD, con el registro DE apuntando al *extremo* de la frase, lo que nos garantiza que la información *no se montará encima* al mudar de sitio el bloque, ya que vamos decrementando.

Las instrucciones "LDIR" y "LDDR" son aún más potentes, capaces de *trasvasar* millares de octetos de un lado a otro y muy rápidamente.

Ejercicio:

Escribe una rutina breve para transferir 32 octetos desde la zona ROM de la memoria hasta la zona de pantalla.

Observa cómo se disponen en la pantalla los 32 primeros octetos.

Ahora inténtalo con 256 octetos; luego con 2048. Y así hasta que lo domines.



INSTRUCCIONES DEL Z80 QUE SE USAN MENOS FRECUENTEMENTE



CANJE DE REGISTROS

Comentamos brevemente en los primeros capítulos, la idea de que la CPU llevaba guantes acartonados que se podía poner y quitar; y mantener así temporalmente información en otros lugares más accesibles que las celdillas de memoria.

Debes recordar que no puedes maniobrar con estos registros alternativos y que la analogía con los guantes es muy valiosa. Aunque ellos *retienen* su forma primitiva, no hay manera que por sí mismos puedan hacer ninguna operación aritmética.

La primera instrucción es:

EX AF, AF'

que hace exactamente lo que su nombre sugiere: *canjea* (EXCHANGE) los pares de registros AF y AF'. En la analogía guanteril, diríamos cambiar los guantes del par de manos AF. En otras palabras, *se pone* el conjunto de guantes de reserva AF -recordarás que el conjunto de guantes de reserva, siempre se ha simbolizado como AF'.

La siguiente instrucción general para el canje de guantes es:

EXX

Esta instrucción, intercambia los guantes de todas las otras manos de 8 bits, es decir:

B	C	→	B'	C'
D	E	→	D'	E'
H	L	→	H'	L'

Es una instrucción muy potente; pero precisamente por ser potente, está limitada en su utilización. Actúa sobre todos los registros de una vez y no es posible *retener* ningún valor de registro (exceptuando el registro A que no se ve afectado por "EXX").

La única forma de *abordar* este problema es escribir una breve rutina de la forma:

PUSH HL
EXX
POP HL

Esto significa que has guardado los valores de BC, DE y HL, en el conjunto alternativo de registros, pero todavía *retienes* el valor de HL para trabajar con él.

La última instrucción de este grupo, no pertenece estrictamente a la clase *canjeadora de guantes*:

EX DE, HL



En la anterior instrucción, DE queda con el contenido de HL, y HL con el contenido de DE.

Y desde luego, es una instrucción muy útil, porque como ya vimos "HL" es una pareja de registros preferente en muchas instrucciones, y hay veces que el valor que queremos manejar está en "DE".



BITS: MIRAR, PONER Y REPONER

Hasta ahora, todas las instrucciones con las que hemos tratado, involucraban el manejo de números de 8 bits o de 16 bits.

El grupo de instrucciones MIRAR, PONER y REPONER, nos permite manipular individualmente los *dedos* de una mano de la CPU (los bits de los registros) y del contenido de las celdillas de memoria. Dado que es muy *farrañoso* el jugar con los bits uno a uno, no es un grupo de instrucciones que se utilice habitualmente. Además se tarda incluso más en manejar un *simple bit* de un registro o de una celdilla de memoria, que en cambiar o examinar *globalmente* los 8 bits de esa celdilla, o de ese registro.

- Sin embargo, hay veces que necesitas *mirar* si un determinado bit está *puesto o repuesto*; o incluso fijar un bit a uno o a cero. Observa, sin embargo, que muchas de las operaciones para prefijar bits pueden realizarse utilizando las operaciones lógicas.

Este grupo de instrucciones nos permite poner cualquier bit a uno (SET), o reponerlo (RESET) a su valor cero, o incluso simplemente mirar (BIT) cuál es el estado de un bit determinado.

Vamos con el primer grupo de instrucciones:

SET n, r
SET n, (HL)
SET n, (IX + d)
SET n, (IY + d)

La instrucción SET, pone a uno (activa), el bit numerado con "n" del registro r, o de la celdilla de memoria que se menciona en la instrucción.

** No se cambia ninguno de los banderines.*

El grupo RESET opera sobre, exactamente, la *misma gama* de registros y de celdillas de memoria, pero en lugar de poner un bit, lo REPONE (lo devuelve al estado "0" que habitualmente es el que consideramos de reposo o inactivo).

La instrucción BIT pudiera escribirse correctamente como *¡BIT?*, ya que la función de esta instrucción es *observar* el estado del bit que se especifica en la instrucción. No se hace ningún cambio al registro o a la celdilla, pero el banderín de cero nos indica el estado del bit que hemos *mirado*.

Si bit = 0 entonces el banderín de cero se alza (en activo \leftrightarrow 1 \leftrightarrow ON)
Si bit = 1 entonces el banderín de cero se baja (en reposo \leftrightarrow 0 \leftrightarrow OFF)

A primera vista puede parecer confuso, pero piénsalo de esta forma:

- Si el bit es 0 entonces el banderín de cero se pone a 1 como *testigo* de SI.
- Si el bit es 1, es *natural* que el banderín de 0 no se alze, sino que se baje.





ROTACIONES Y DESPLAZAMIENTOS

Puedes *girarlo* a la izquierda, puedes *rotarlo* hacia la derecha, puedes *desplazar* el contenido de los registros *casi* de cualquier forma que te guste.

El *truco* para diferenciar entre los desplazamientos y las rotaciones con el fin de saber cuál usar, es recordar que el bit de acarreo puede considerarse como el *noveno* bit de los registros. (ie. el bit de acarreo es el bit *numerado* con 8 si los bits del registro se numeran de 0 a 7).

Algunas instrucciones de rotación pasan por el acarreo (como noveno bit) de manera que se da una vuelta completa, formando un ciclo de 9 bits.

Por ejemplo, observemos la instrucción "RLA" (su significado será más claro al final de este capítulo):



Otras rotaciones, sólo forman un ciclo de 8 bits, aunque el banderín de acarreo cambia de acuerdo con el bit que ha dado *la vuelta más larga*. Un ejemplo de esto es la instrucción RLCA:



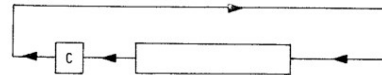
Esto significa que en una rotación a izquierdas como la anterior, el contenido del bit *cero* se transfiere al bit *uno*, el del bit *uno* al *dos*, etc.; pero en contenido del bit *siete* es transferido al bit de *acarreo* y al bit *cero*. Compáralo con la instrucción RLA de arriba en que el bit *siete* se pasa al bit de *acarreo*, mientras que el de *acarreo* se transfiere al bit *cero*.

Rotaciones a izquierdas:

Básicamente, hay dos clases de rotaciones a izquierdas:

- ★ ROTAR A LA IZQUIERDA UN REGISTRO -es una rotación con un ciclo de nueve bits como la mencionada anteriormente en la primera figura.

RLA "Rotar izquierda (left) acumulador"
RL r "Rotar izquierda (left) registro r"



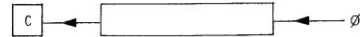
★ ROTAR A LA IZQUIERDA CIRCULARMENTE -circularmente, significa que es un ciclo de ocho bits solamente, como en la instrucción mencionada anteriormente en segundo lugar.

RLCA	rotar izquierda circular A
RLC r	rotar izquierda circular r
RLC (HL)	rotar izquierda circular (HL)
RLC (IX + d)	rotar izquierda circular (IX + d)
RLC (IY + d)	rotar izquierda circular (IY + d)



Además de estas dos instrucciones de rotación a la izquierda, hay un desplazamiento a la izquierda, pero que solamente opera con el registro A.

SLA Shift (desplazar) Left (izquierda) acumulador.



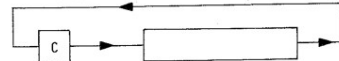
Es diferente, ya que el contenido del bit de acarreo se pierde, mientras que el Bit cero se rellena con el valor cero. En realidad es multiplicar A por 2 ya que nada se mete en el acumulador.

✦ Piensa un poco sobre SLA con A = 80H.

Rotaciones a la derecha:

De nuevo, tenemos los dos modos fundamentales de rotaciones pero esta vez a la derecha. Exactamente la misma gama de posibles celdillas de memorias y de rotaciones pueden efectuarse hacia la derecha como se podían a la izquierda.

RRA	rotar derecha (right) acumulador
RR r	rotar derecha (right) registro



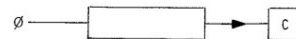
RRCA	rotar derecha circular A
RRC r	rotar derecha circular r
RRC (HL)	rotar derecha circular (HL)
RRC (IX + d)	rotar derecha circular (IX + d)
RRC (IY + d)	rotar derecha circular (IY + d)





Igual para el desplazamiento a izquierdas hay un desplazamiento a derechas *similar*, que se llama:

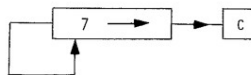
SRL r - Desplazamiento derecha lógico registro r



En este caso es una mera división por 2 siempre que usemos números sin signo (ie. la gama de números entre 0 y 255).

Pero a causa de que en algunas aplicaciones utilizamos el convenio de marcar los números negativos poniendo el bit 7 a uno (ie. la gama entre -128 y +127, hay una *instrucción adicional* de desplazamiento a derechas que se llama

SR A r - Desplazamiento derechas aritmético r



Como puedes ver, también es una división por 2, pero se mantiene el bit de signo.



DENTRO Y FUERA

Dentro (IN) y fuera (OUT) son tan *simples* en concepto como podría esperarse de la programación en lenguaje-máquina.

Hay veces que la CPU necesita obtener información del mundo exterior (ninguna CPU es una *isla*) tal como del teclado o del cassette.

En lo que se refiere a la CPU eso es territorio *extranjero* y como todas las buenas CPU nunca dejan su *patria*. A lo más que llegan es a *señalar* un PORTAL para dejar en él las mercancías. La CPU no sabe ni le importa saber cómo funciona un cassette. (O un *puerto* si prefieres. Port en inglés).

Toda la información que le interesa saber es *en qué portal*, el hombrechito del cassette va a dejar su mercancía (puede elegir entre 256 portales al usar la pastilla 280; pero el número real disponible en un determinado computador es el resultado de las decisiones que han hecho los que fabricaron los circuitos). En cuanto *conciérne* al Sinclair sólo están el portal del teclado, el de la impresora, y el del cassette.

- Otra cosa que la CPU no quiere saber, es *cómo se transmiten* los datos. Por lo que a ella le atañe, lo que hay en el portal es simplemente un octeto de 8 bits, que se van a *llevar a*, o que han *traído de*, algún sitio.

Tanto el teclado como el cassette, están *al otro lado* del portal numerado FEH (254 en decimal), de manera que para *meter* los datos que se tecleen, se *usa* la instrucción

IN A, (FE)

Ahora bien, te preguntarás *cómo están dispuestas las 40 teclas del teclado y cómo están representadas por octetos de 8 bits*.

La respuesta, no es lo que podía esperarse. El teclado sólo *entrega* de cada vez, información de *cinco* teclas. Es el valor que hay en A cuando la CPU *acude al* portal, lo que determina el grupo de cinco teclas que va a ser *investigado*.

El teclado está dividido en 4 filas, cada una con 2 grupos de 5 teclas:

3	→	1	2	3	4	5	6	7	8	9	0	←	4
2	→	Q	W	E	R	T	Q	U	I	O	P	←	5
1	→	A	S	D	F	G	H	J	K	L	N/L	←	6
0	→	SFT	Z	X	C	V	B	N	M	.	SPC	←	7

Por tanto, hay "8" grupos de 5 símbolos y es posible correlacionarlos con los "8" bits de A. De hecho, así es como se procede.



Todos los bits de A se ponen a uno, excepto el que especifica el grupo a investigar.

Puedes verlo como una *contraseña* -cuando la CPU va al portal a recoger la información, la *contraseña* determina a qué grupo de teclas pertenece la información que recibe.

Así, para leer las teclas del grupo "1 2 3 4 5", coloca sólo el bit 3 del acumulador al valor cero.

A = 1 1 1 1 0 1 1 1 = F7

y da la instrucción: IN A, (FE)

El símbolo teclado se *recoge* en A, con la información en los bits inferiores:

ie.

tecla "1"	→	bit 0 de A con valor cero
tecla "2"	→	bit 1 de A con valor cero

Si se hubiera escogido el grupo *número cuatro* (ie. A = EFH), la información sería:

tecla "0"	→	bit 0 de A con valor cero
tecla "9"	→	bit 1 de A con valor cero

Puedes considerar que la información llega a A primeramente, desde los *bordes exteriores* del teclado, de forma que tanto la tecla "0" como la tecla "1" corresponden al bit 0 del registro A.

En algunas aplicaciones, puedes querer que se lea *toda la fila superior* de teclas, y se puede leer con una sola instrucción (mejor con las dos instrucciones que se precisarían si leyéramos los grupos de teclas que la componen). Se logra *engañando al portero* para que te dé dos lotes de información de una sola pasada:

eg. A = 1 1 1 0 0 1 1 1 = E7

Observa que tanto el bit 3 como el 4 están a cero, como corresponde a los dos grupos.

- Esta *contraseña* le dice al portero que la CPU desea información de los bloques 3 y 4, y eso es *lo que recibe* a cambio. Desde luego, los dos lotes de información se *mezclan* y no puedes precisar por ejemplo, si se pulsó la tecla "0" o la tecla "1", (ya que *ambas* corresponden al bit 0 de A).

ie.

teclas 1 ó 0	→	bit 0 de A con valor cero
teclas 2 ó 9	→	bit 1 de A con valor cero



(Es útil en los juegos de movimiento, porque permite que se usen las teclas "5" y "8" para dirección derecha e izquierda, incluso aunque pertenezcan a diferentes grupos de teclas).

Observa que si usas la instrucción

IN r, (C)

siendo el registro C el que especifica *el portal* que deseas, es el contenido del registro "B" el que define el grupo de teclas que estás investigando. (Lo que indica un buen sentido del humor...

Los *otros portales* que pueden interesarte, son obviamente los de entrada y salida hacia el *cassette*.

Como ya hemos dicho, *sigue siendo* el portal "FE". El problema mayor ahora es la *temporización* de los datos que *entran y salen*; esta clase de problema exige un *montón de experiencia* en lenguaje-máquina y *cálculos complejos* del tiempo empleado en ejecutar cada instrucción.

La instrucción OUT también se utiliza para generar sonido en el Spectrum y para establecer el color del borde de la imagen.

El manual del Spectrum, comenta la instrucción "OUT" de lenguaje BASIC, y en código máquina el comando OUT es exactamente lo mismo. En otras palabras: los bits 0, 1 y 2, definen el color del *borde*; el bit 3 envía un *impulso* hasta los enchufes MIC y EAR; mientras el bit 4, lo envía hacia el *altavoz* interno.

Para cambiar el color del borde, se ha de cargar A con el *valor* apropiado de color y luego ejecutar la instrucción OUT (FE), A. Observa que esto es sólo un cambio TEMPORAL en el color del borde. Para cambiarlo permanentemente, debes efectuar la instrucción anterior OUT y *además* cambiar el valor de la celdilla de memoria 23624, que es la *variable* del sistema operativo BORDER (véase el manual del Spectrum).

La razón está en que los circuitos del Spectrum (la pastilla ULA del Spectrum) controlan el color del borde, y que obtienen su información *guiando* el contenido de esa celdilla de memoria. Tú puedes *impedir* que los circuitos *te incordien* con el color del borde, sólo si desactivas todas las interrupciones (instrucción DI). Pero ten en cuenta que algunas de las subrutinas de la ROM, reactivan las interrupciones (instrucción EI).

Creando tu propio sonido:

Puedes crear tu propio sonido, pero hay algunas limitaciones debidas a los circuitos, cuando el usuario sólo tiene 16K de RAM.

- Para que la pantalla se esté *actualizando* constantemente, los circuitos interrumpen regularmente al Z80 para que *dejando sus tareas* aparte, muestre lo que hay en la región de pantalla. Lo logra, colocando la línea de espera (WAIT) del procesador a valor bajo.



Como consecuencia de esto, cualquier programa que exija una temporización exacta o regular es imposible, ya que no se puede predecir los tiempos de estas interrupciones. El diseño del Spectrum es tal, que el Z80 sólo es interrumpido cuando está intentando procesar información guardada en las primeras 16K de RAM. No se producen esas interrupciones si el programa o los datos que el Z80 está accediendo corresponden a la ROM, o a la parte superior de los 32K de memoria.

Para resumir esto en los términos de un profano: Puedes producir sonidos y ruidos utilizando el comando OUT si tienes una máquina con 16K, pero no notas puras. (Se puede soslayar esto llamando a la rutina de pitido (BEEP) de la ROM -véase el capítulo sobre las peculiaridades del Spectrum).

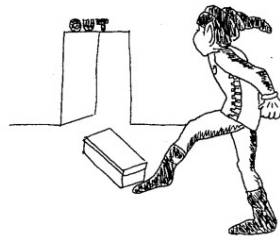
★ Para crear sonido, necesitas enviar un impulso que conecte el altavoz (y/o el enchufe MIC si ha de ser amplificado). Luego y un poco más tarde, necesitas enviar otro impulso para desenchufarlo. Luego y un poco más tarde, volver a enchufar, y así sucesivamente.

El sonido se crea de esta forma. El tiempo transcurrido entre conectar el altavoz y la siguiente vez en que lo vuelves a conectar, es el que determina la frecuencia del sonido. La longitud del tiempo que dejas el impulso puesto, en oposición al tiempo total entre impulsos, puede darte un pequeño grado de control sobre el volumen.

Ten en cuenta que siempre debes usar un valor de A para conectar y desconectar tal que el color del borde permanezca constante. En caso contrario, obtendrás un tramado de bandas similar a cuando estás cargando un programa.

Ejercicio:

Escribe una rutina que simule una sirena de ambulancia (incrementando la frecuencia y luego decrementándola). Ten en cuenta que debes mantener cada frecuencia durante un período corto antes de pasar al siguiente valor de frecuencia.



REPRESENTACION BCD

BCD corresponde a Binario Codificado Decimal. Es una forma de representar información numérica en base decimal.

Para codificar cada una de las *cifras* desde el 0 hasta el 9, *sólo* necesitamos 4 bits, y las seis restantes combinaciones *no* corresponden a ninguna *cifra*.

Dado que se necesitan *cuatro bits* para codificar una *cifra decimal*, se pueden codificar en cada octeto, dos cifras decimales.

★ Eso se denomina representación BCD.

eg.

00000000	→ es la representación BCD del decimal 00.
10011001	→ es la representación BCD del decimal 99.

¿Cuál es la representación BCD de 58? ¿Y de 10?

¿Es 10100000 una representación BCD válida?

ARITMETICA BCD

- Este extraño convenio de representar números, puede darnos *problemas* con la suma y la resta.

Intenta sumar lo siguiente en decimal y en BCD.

	08	0000	1000
+	03	0000	0011
	11	0000	1011

Observa que el resultado de la segunda operación, es erróneo y es un número BCD que no es válido. Para compensar, se debe utilizar una * instrucción especial, DAA -"decimal ajuste aritmético"-, para corregir el resultado de la suma. (ie. Sumar 6 si el resultado es mayor que 9).

Otro problema se *ilustra* con el mismo ejemplo. Se generará un acarreo desde el *dígito* inferior BCD (el de la parte *derecha*) hacia el de la parte *izquierda*). Este acarreo *interno* también debe tomarse en cuenta para *añadirlo* al segundo *dígito* BCD.

El banderín de semi-acarreo H, se usa para *detectar* este acarreo.

LD A, 12H	; cargar el número BCD "12"
ADD A, 24H	; sumar el número BCD "24"
DAA	; ajustar el resultado a valor decimal
LD (direc), A	; guardar el resultado en la memoria.



No es probable que uses en tus programas, representación BCD. Pero es bueno que sepas que el Z80 también permite esta representación y que la instrucción DAA hace que la vida de un pequeño grupo de usuarios de BCD sea más simple.

INTERRUPCIONES

Una interrupción es una señal enviada al microprocesador, que puede producirse en cualquier momento y que habitualmente detendrá la ejecución del programa en curso (sin que el programa lo sepa).

En el Z80 hay provistos tres mecanismos de interrupción:

- La *recuesta* (requerimiento) de los haces de datos (BUSRQ).
- La interrupción no cancelable (non-maskable),
- y la *interrupción* normal (INT).

Desde el punto de vista de programación, sólo consideramos la interrupción normal (INT) que puede ser cancelable a voluntad. (Y los anglófilos la llaman enmascarable).

La instrucción DI (cancelar o deshabilitar las interrupciones) se utiliza para anular la posibilidad de ser interrumpido (MASK) mientras que la EI (permitir o habilitar las interrupciones), vuelve a admitir que el procesador sea interrumpido.

Habitualmente, una interrupción ordinaria, hará que el contador del programa sea *apilado* sobre la pila, y luego varíe su valor al correspondiente en la página cero de la ROM mediante la instrucción RST (reanudar). Para regresar de la interrupción al punto anterior se precisa una instrucción RETI (retornar de la interrupción).

- Durante la operación normal, el Spectrum tiene las interrupciones *habilitadas* (EI), y de hecho, el programa se interrumpe 50 veces por segundo. Esta interrupción permite *escrutar* el teclado mediante la rutina de la ROM para ver si hay alguna tecla pulsada.

Puede que quieras *cancelar las interrupciones* en tus programas, ya que eso *aumentará* la velocidad de ejecución. Y todavía podrás leer el teclado si utilizas tu propia rutina para hacerlo. Pero *asegúrate* que cuando acabas con tu programa dejas las interrupciones permitidas, ya que en caso contrario el sistema no podrá leer el teclado para ver lo que deseas *mandarle*.



INSTRUCCIONES DE REANUDACION (RST)

Esta instrucción está en realidad *dejada* en el Z80 para compatibilidad con el procesador 8080. Por ello es *muy poco probable* que utilices instrucciones RST en tu programa.

- La instrucción RST, realiza las mismas acciones que una *llamada*, pero sólo permite *saltar* a una de las 8 direcciones siguientes: 00H, 08H, 10H, 18H, 20H o 38H. (Todas están dentro de las primeras 256 celdillas de la memoria).

La ventaja de la instrucción RST, es que podemos *acudir* a las subrutinas que son más frecuentes, mediante un *solo octeto*. Además la instrucción RST, tarda menos tiempo que la instrucción CALL.

La desventaja de la instrucción RST, es que sólo puede utilizarse para acudir a una de las ocho celdillas mencionadas anteriormente.

Como todas estas celdillas pertenecen a la ROM, tú no puedes beneficiarte de esta ventaja en tus propios programas. Sin embargo, *es posible* que utilices las subrutinas de la ROM que están situadas en esa dirección *si sabes lo que hacen*, y para eso has de usar las instrucciones RST.

Podrás saber más acerca de las instrucciones RST con nuestro libro "Entendiendo tu Spectrum" del Dr. Ian Logan.



PROGRAMANDO TU SPECTRUM



PLANTEANDO TU PROGRAMA EN LENGUAJE-MAQUINA

La programación en lenguaje-máquina es extremadamente flexible, ya que te permite hacer *todo de todo*.

Dado que todos los lenguajes de alto nivel, al fin y al cabo, se *traducen* al lenguaje-máquina, deduciremos que *todo* lo que puedas programar en FORTRAN, COBOL o en cualquier otro lenguaje, *puede* realizarse en lenguaje-máquina. Con el beneficio adicional que el programa en lenguaje-máquina será el más rápido de todos.

○ Esta flexibilidad total puede también ser el *cepo* donde quede atrapado el programador *inconsciente*. Con tanta libertad, es posible que no se logre nada. A diferencia del sistema operativo del Spectrum, por ejemplo, no hay comprobación de si las instrucciones son legales o no.

○ Y dado que todos los números que tú puedas teclear corresponden a una instrucción de una clase o de otra, la pastilla Z80 procesará todo y resultará lo que resulte.

Pero incluso y más allá de los problemas de comprobación de sintáxis, la programación en lenguaje-máquina no tiene restricciones sobre tu razonamiento -tú puedes realizar funciones, saltos, etc. que serían totalmente ilegales en cualquier lenguaje de mayor nivel.

Es, por tanto, de importancia primordial que guardes una cierta disciplina sobre tí mismo, al programar en lenguaje-máquina. No podemos *encomiar* todo lo útil que el concepto de programación "de arriba hacia abajo" es, en general, sobre todo y muy *especialmente* al usar lenguaje-máquina.

La aproximación "de arriba hacia abajo" te fuerza a desglosar el problema en unidades más pequeñas y te capacita para comprobar la lógica de tu diseño sin tener que codificar hasta que lo tienes completamente diseñado.

Suponte que quieres escribir un programa para un alunizaje:

La *primerísima* aproximación puede ser de la forma:

NORMAS	Presentar las normas a seguir. Volver a NORMAS hasta que pulsen ENTER.
DIBUJO	Dibujar el paisaje; módulo lunar arriba
BAJADA	Mover el módulo alunizador <i>Si combustible acabado ir a TORTAZO</i> <i>Regresar a BAJADA si no alunizaje</i>
LLEGADA	Dar enhorabuena Regresar a NORMAS para otra jugada
TORTAZO	Acompañarle en el sentimiento Regresar a NORMAS para otra jugada



Observa cómo este programa está escrito totalmente en "puro idioma hispano". En este momento no se ha hecho ninguna decisión de si el programa va a ser codificado en BASIC o en lenguaje-máquina. Ni es necesario hacer esa decisión. El concepto en que se basa un programa de alunizaje no depende de la codificación.

Ahora viene la parte de la comprobación de la *lógica*. Tú haces de computador y ves si todas las posibilidades que deseas están incluidas en el programa. ¿Hay algunos datos o cosas que querías incluir pero olvidaste? ¿Todo está ahí? ¿Son algunas rutinas redundantes? ¿Algunas de las cosas deberían ponerse como subrutinas?

Ojémos de nuevo el programa. *Leñe*, olvidamos decir en qué forma acaba!

La lógica anterior puede estar bien para algunas aplicaciones, tales como *comecocos*; pero en tu programa puedes decidir que te gustaría poder terminar el juego!

Cambiamos ahora, la última parte del programa como sigue:

LLEGADA	Dar enhorabuena <i>Saltar a Final</i>
TORTAZO	Acompañarle en el sentimiento
FINAL	Preguntar si quiere acabar <i>Si "no", regresar a NORMAS</i> <i>Si "sí", STOP.</i>

Verás que hemos usado *rótulos* para indicar ciertas líneas del programa. Son muy valiosos. Tanto más, si eliges *rótulos cortos* que describan el *significado* de las acciones que se efectúan en el *módulo encabezado* por esa línea.

Una vez que hemos terminado con este nivel, *descendemos* un nivel más, haciendo lo mismo con cada una de las líneas o *módulos* anteriores. Por eso es por lo que esta forma de trabajar se denomina *aproximación de arriba hacia abajo*.

Por ejemplo, ampliémos el módulo FINAL anterior:

FINAL	Limpiar pantalla Preguntar: ¿quieres parar ahora? Explorar el teclado para aceptar respuesta Si respuesta = "sí" pues STOP Saltar a NORMAS
-------	--

Otro beneficio de esta aproximación de arriba hacia abajo, es que puedes comprobar un módulo en particular, sin que esté preparado el programa completo.

Sigamos bajando un nivel más y analicémos la línea

Limpiar pantalla



En este momento, ya tenemos que decidir en qué lenguaje vamos a escribir el programa y *escojamos* el lenguaje-máquina del Sinclair.

Si lo escribiéramos en BASIC, todo lo que tendríamos que decir es:

900 CLS

pero en lenguaje-máquina, esa simple instrucción, limpiar la pantalla, puede ser *engañosa*.

Podríamos hacer algo como:

LIMPIAR

Buscar en la memoria el comienzo de la región de pantalla
Rellenar las siguientes 6144 posiciones con blancos

Todavía no hemos hecho ninguna codificación, pero obviamente nos estamos basando en lenguaje-máquina. Mirémos más cuidadosamente qué es lo que exactamente significa limpiar pantalla y cómo realmente se hace.

* Recordarás del manual del Spectrum, que la pantalla está compuesta de 6144 posiciones y que hay 768 celdillas más, que describen los atributos de la pantalla -color del papel, color de la tinta, etc.

Las dos frases anteriores, desde luego que limpiarán la región de pantalla, pero no tendrá ningún efecto sobre la región de atributos. Puede que no toda la pantalla tuviera el mismo color de papel o que algunas posiciones tuvieran puesta la condición de parpadeo o de brillante, y entonces la rutina anterior de limpiar pantalla es claramente *incompleta*.

Necesitamos también, manejar la región de atributos (date cuenta como ciertas tareas se complican en lenguaje-máquina con relación a BASIC).

Necesitamos por tanto, ampliar el programa para que diga:

*Encontrar el comienzo de la región de pantalla
Rellenar los siguientes 6144 octetos con blancos
Encontrar el comienzo de la región de atributos
Rellenar los siguientes 768 octetos con el papel/tinta deseados.*

Si bajamos al siguiente nivel, es en el que finalmente hacemos la codificación, así que, veámos cómo rellenamos la pantalla con blancos:

LIMPIAR

LD HL, PANTALLA ; Comienzo de pantalla
LD EC, 6144 ; Octetos a blanquear
LD D, 0 ; D = blancos
LD (HL), D ; Llenar con blanco
INC HL ; Siguiente celdilla
DEC EC ; Disminuir cuenta
LD A, B
OR C ; Comprobar si BC = 0
JR NZ, LAZO ; Repetir si no terminado.

LAZO



Un programa de esta longitud, puede manejarse bastante *fácilmente* y de esta forma *construir* programas realmente muy *complejos*.

A propósito, no creo que te quede duda ahora, en entender por qué los programas en lenguaje-máquina tienden a ser tan largos y por qué la gente inventó los lenguajes de alto nivel.

EJERCICIOS

Hay más de una forma de escribir una determinada rutina, así es que revisémos la que hemos escrito para limpiar la pantalla. Se puede abordar desde varias aproximaciones diferentes.

Ejercicio 1:

¿Puedes pensar una forma que haga que el lazo *blanquee* 6144 posiciones sin usar el registro BC, sino usando *sólo* el registro B de manera que nos sirva la instrucción DJNZ?

Ejercicio 2:

Piensa un poco sobre lo que LDIR hace: ¡no siempre es necesario tener en otro lugar 6144 posiciones en blanco!

Respuestas:

Más de una posible respuesta puede ser correcta -la única comprobación es que *funcione*. En otras palabras, que hace lo que tú quieres.

Usando DJNZ:

LIMPIAR	LD HL, PANTALLA	
	LD A, 0	
	LD B, 24	; Poner B = 24
LAZOMAYOR	PUSH BC	; Guardar el valor
	LD B, A	; Poner B = 256
LAZOMENOR	LD (HL), A	
	INC HL	; Rellenar 256 posiciones
	DJNZ LAZOMENOR	
	POP BC	; Recuperar el valor de B
	DJNZ LAZOMAYOR	; Repetir hasta terminar

Hemos repetido 24 veces 256 (= 6144) para limpiar la pantalla.

Los puntos a tener en cuenta son:

Podemos hacer B = 0 para recorrer el lazo DJNZ 256 veces. ¿Por qué? No usaríamos normalmente este procedimiento en un programa; a no ser que estuviéramos empleando el registro C con otros fines.



Usando LDIR:

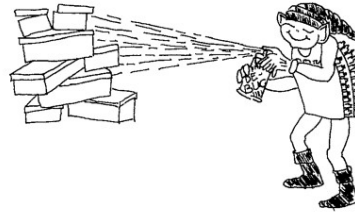
LIMPIAR

```
LD HL, PANTALLA ; origen
PUSH HL
POP DE
INC DE           ; DEST = HL + 1
LD BC, 6144      ; Cuántos
LD (HL), 0       ; 1 posición = 0
LDIR             ; Mudarlo.
```

Observa que hemos hallado $DE = HL + 1$ haciendo $DE = HL$ e incrementando DE . Puede lograrse más fácilmente, cargando en DE directamente el valor de pantalla + 1, pero ¡eso requiere un octeto más!

La razón por la que ésta LDIR funciona, es porque, de hecho, los datos están sobrescribiendo el bloque, a medida que va avanzando. Es usar de manera positiva el problema que comentábamos en el capítulo de movimiento de bloques.

- Si calculáras la memoria que requieren; el primer método daría 14 octetos, el segundo 16 octetos y el último 13 octetos.



Ya es hora que echémos una ojeada sobre aquellas peculiaridades de tu ZX Spectrum, que son útiles cuando se desarrollan programas en lenguaje-máquina.

Entrada-Teclado

En lo que concierne al Spectrum como entradas, despreciaremos la entrada del cassette y nos concentraremos en la del teclado.

El teclado es la única entrada que *provee* comunicación en tiempo real. Puede afectar dinámicamente el procesado de cualquier programa, ya sea del sistema operativo en la ROM o del programa de usuario en la RAM.

Podemos *ver* el teclado como una matriz bidimensional con 8 filas y 5 columnas (como en el apéndice A).

Cada uno de los 40 *cruces* representa una tecla del teclado. En su estado normal (cuando no están presionadas), están siempre con un *humor* alto, ie. la intersección se pone a 1.

○ Cuando se pulsa una tecla en particular, la intersección correspondiente a esa tecla se coloca en un *humor* bajo, es decir, se pone a cero. Sabiendo la relación entre el teclado y esta representación matricial interna, podemos deducir una forma racional de comprobar la tecla que está pulsada, usando un programa en lenguaje-máquina.

○ En BASIC, cuando exploramos el teclado, necesitamos suministrar una dirección para esa determinada semifila del teclado en que la tecla buscada *reside*, antes de usar la función IN como se describe en el manual del Spectrum. Igualmente, en un programa en lenguaje-máquina, necesitamos cargar en el acumulador un valor que corresponda con la dirección de la semifila de teclas que queremos examinar. El valor *requerido* para cada semifila está numerado en la columna *izquierda* de la tabla del apéndice A.

Por ejemplo, para la semifila "H - ENTER" cargamos A con el valor de BFH

```
LD A, BFH
```

El valor en A, se usará luego para *recabar* el octeto que contiene el estado de esa particular semifila de teclas, y entregarlo al acumulador cuando se dicta la instrucción INput.

eg. el portal usado es el portal FEH

```
IN A, (FEH)
```

Dado que por cada semifila hay cinco teclas, sólo nos incumben los cinco bits de orden inferior del octeto que hay en el acumulador.



Si en esa semifila no hay ninguna tecla pulsada, el valor de los 5 bits de menor orden será $(2^{*4} + 2^{*3} + 2^{*2} + 2^{*1} + 2^{*0})$ ie. $16 + 8 + 4 + 2 + 1 = 31$).

registro A = xxxlllll

cuando no se pulsa ninguna tecla.

Si queremos comprobar si el bit extremo derecho se ha pulsado, miraremos a ver si ese bit está bajo.

Hay dos formas de examinar eso:

- i. utilizar las instrucciones de comprobación de bits, eg BIT 0, A si el bit está bajo (off) entonces el banderín de cero estará alzado.
- ii. utilizar la instrucción lógica de iliación (en este ejemplo: AND 1) si el bit está bajo (off) el resultado será cero y el banderín de cero estará alzado.

El primer método es más fácil porque especificamos directamente en la instrucción de comprobación de bits el bit particular que queremos. Pero tiene una desventaja si queremos comprobar dos teclas de esa semifila, necesitaremos usar dos instrucciones y posiblemente dos saltos relativos.

eg. para comprobar el bit cero y el bit uno con el primer método.

```
BIT 0, A      ; comprobar el bit 0 de A
JR Z, NOTECLA ; saltar si no se ha pulsado
BIT 1, A      ; comprobar el bit 1 de A
JR Z, NOTECLA ; saltar si no se ha pulsado
```

.

hacer lo que corresponda cuando ambas están pulsadas

.

NOPULSO

hacer lo que corresponda cuando no están pulsadas ambas

.

El segundo método de comprobación utilizando la iliación requiere un poco más de lógica. Para comprobar el bit 0 usamos AND 1; para comprobar el bit 1 usamos AND 2; para comprobar el bit 2 usamos AND 4 y así sucesivamente.

Para comprobar dos teclas simultáneamente, usaremos AND x, siendo "x" la suma de los valores que usaríamos al comprobar cada tecla individualmente.

eg. Para comprobar ambos, el bit cero y el bit uno de A.

```
AND 3      ; elegir el bit 0 y el bit 1
CP 3       ; comprobar si ambos están alzados
JR NZ, NAMBOS ; saltar si ambos no están pulsados
```

.

.



Para comprobar si *bien* el bit cero, o *bien* el bit uno están puestos:

```
AND 3           ; elige el bit 0 y el bit 1
JR Z, NINGUNO   ; salta si ninguno está pulsado
.
```

Ejercicio:

Para *resumir* lo que hemos aprendido en relación con el teclado, puedes codificar una rutina en lenguaje-máquina que *atrapé* la pulsación en tu Spectrum de la tecla ENTER.

Necesitarás hacer lo siguiente:

- a. *mirar la dirección de la semifila que necesita cargarse en A.*
- b. *enviarla al portal de entrada FEH.*
- c. *comprobar que es el bit que queda alzado al pulsar la tecla ENTER.*



La pantalla de video es la principal *fente* de salida que tiene el computador para comunicarse con el usuario.

El siguiente programa en lenguaje-máquina, muestra la forma en que está organizada la zona de memoria que guarda relación con la pantalla:

210040	LD HL, 4000H	; cargar HL con el comienzo de la memoria de pantalla
36FF	LD (HL), FFH	; llenar de unos esa celdilla
110140	LD DE, 4001H	; cargar DE con el siguiente octeto de la memoria de pantalla
010100	LD BC, 1	; BC contiene el número de octetos a transferir
EDB0	LDIR	; mover un bloque de longitud igual a BC desde (HL) hasta (DE)
C9	RET	; regresar a BASIC

Carga este programa en tu Spectrum y *rúlalo*. La forma en que lo hemos escrito hace que *sólo* se transfiera un octeto desde (HL) a (DE).

Ahora *cambia* la cuarta línea para que sea LD BC, 31 (011F00). Puede que te sorprendas al ver los primeros 32 octetos de la pantalla. Verás como se ha dibujado una línea *muy delgada* a lo largo del borde superior de la pantalla. Los primeros 32 octetos de la zona de memoria correspondiente a la pantalla guardan relación con el primer octeto de cada uno de los primeros 32 caracteres.

- Ahora *cambia* esa línea para que sea LD BC, 255 (01FF00). Puede que te sorprendas de nuevo. El siguiente octeto después del 32-ésimo no está en la segunda fila de *motas* de la pantalla; es el primer octeto del 32-ésimo carácter! Y así sigue hasta el 256-ésimo carácter.

*¿Estás preparado para predecir dónde irá el siguiente octeto?
Cambia esa línea a LD BC, 2047 (01FF07) y rula el programa.
Verás que el tercio superior de la pantalla es el único que se ha llenado.*

Experimenta con eso, utilizando diferentes valores para BC hasta que sea LD LD BC, 6143 (01FF17). De esa forma, verás la manera en que el Spectrum organiza la pantalla.

- | | |
|----------------------------|----------------------|
| i. Memoria 4000H - 47FFH | primeras ocho líneas |
| ii. Memoria 4800H - 4FFFFH | segundas ocho líneas |
| iii. Memoria 5000H - 57FFH | terceras ocho líneas |

No sólo eso, sino recordarás que cada carácter en un Spectrum se compone de 8 octetos lo que hace 64 motas.



eg. Para el signo "!", su representación es

0	00000000	0H
16	00010000	10H
16	00010000	10H
16	00010000	10H
16	00010000	10H
16	00010000	10H
0	00000000	0H
16	00010000	10H
0	00000000	0H

La organización de la memoria de pantalla del Spectrum es tal, que los primeros 256 octetos, (desde 4000H hasta 40FFH) corresponden al primer octeto de cada uno de los 256 símbolos de 8 octetos, que hay en las primeras ocho líneas. Luego los siguientes 256 octetos (desde la celdilla 4100H hasta 41FFH) corresponden al segundo octeto de cada uno de los 256 símbolos de 8 octetos de las primeras ocho líneas; y así sigue.

Así, las ocho celdillas de memoria que corresponden al primer símbolo de la primera línea de la pantalla son

Primer octeto	4000H
Segundo octeto	4100H
Tercer octeto	4200H
Cuarto octeto	4300H
Quinto octeto	4400H
Sexto octeto	4500H
Séptimo octeto	4600H
Octavo octeto	4700H

Extraño, ¿verdad?. Pero tenemos que admirar del Spectrum la forma en que lo han construido.

¿Podrías escribir los ocho octetos que corresponden al 31-ésimo carácter de la tercera línea de la pantalla? Puedes acudir al apéndice B, que es un mapa de la memoria de pantalla.

Solución: 405EH, 415EH, 425EH, ..., 475EH.

Para seguir con el concepto que hemos desarrollado sobre la memoria de pantalla, la celdilla de memoria que corresponde al primer carácter del segundo trozo de 8 líneas es:

4800H, 4900H, 4A00H, 4B00H, 4C00H, 4D00H, 4E00H, 4F00H.

De igual manera, el primer carácter del tercer lote de 8 líneas tiene sus 8 octetos en las celdillas de memoria:

5000H, 5100H, 5200H, 5300H, 5400H, 5500H, 5600H, 5700H.

Sin embargo, hay algunas ventajas en usar lenguaje-máquina. Y las complejidades aparentes merecen ser superadas. Como ejemplo trivial, en BASIC, si intentas exponer (PRINT) en la sección de entrada de la pantalla -las dos últimas líneas-



el sistema operativo *protestará* de lo más violentamente. Pero en lenguaje-máquina tienes *accesibilidad total* a toda la pantalla, incluyendo esas dos líneas.

Si observas minuciosamente la organización de la memoria de pantalla, verás que el octeto de mayor orden de la dirección de una celdilla, determina en cual de los tres trozos de pantalla saldrá el contenido de esa celdilla.

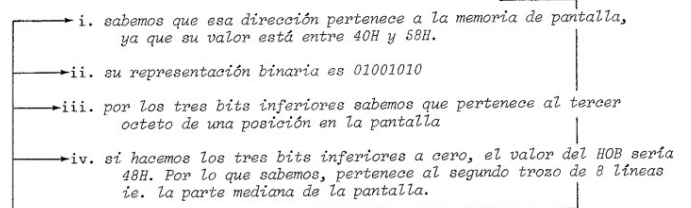
Por ejemplo, si está entre 40H y 41H saldrá en el primer trozo de ocho líneas
si está entre 48H y 49H saldrá en el segundo trozo de ocho líneas
si está entre 50H y 51H saldrá en el tercer trozo de ocho líneas

- No sólo eso, los tres bits de menor orden del octeto de mayor orden (HOB) determina a qué octeto pertenece, de los ocho que todo carácter tiene en forma gráfica.

○ ¿Se está volviendo todo un completo galimatías? Vete al apéndice B e intenta observar la relación entre las direcciones de la memoria y las posiciones en la pantalla (¿ves alguna?).

Intentemos el siguiente ejemplo. Supongamos que nos dan la dirección 4A36H.

El HOB (alto-orden-octeto) de esa dirección es 4AH. Así que:

- 
- i. sabemos que esa dirección pertenece a la memoria de pantalla, ya que su valor está entre 40H y 58H.
 - ii. su representación binaria es 01001010
 - iii. por los tres bits inferiores sabemos que pertenece al tercer octeto de una posición en la pantalla
 - iv. si hacemos los tres bits inferiores a cero, el valor del HOB sería 48H. Por lo que sabemos, pertenece al segundo trozo de 8 líneas ie. la parte mediana de la pantalla.

La conclusión a la que llegamos es que el octeto dado se refiere al tercero de un carácter en la parte mediana de la memoria de pantalla.

○ ¿A qué carácter de la parte mediana pertenece ese octeto? Para contestar esta cuestión, necesitamos analizar el valor del octeto de orden bajo (LOB) de la dirección.

Sabemos que el LOB de la dirección es 36H. Así que esa dirección se refiere al carácter 36H ($3 \times 16 + 6$) que es la posición 54-ésima a partir del primer carácter del trozo del medio.

- Dado que cada línea tiene 32 caracteres, la posición mencionada está en la segunda línea y es el $(54 - 32 + 1)$ -ésimo carácter de esa línea.

Concluyendo.

El octeto dado es el tercer octeto del 23-ésimo carácter de la décima línea a partir del comienzo de la pantalla.



Ejercicio:

¿Cuál es el octeto y a qué carácter corresponde, si la dirección de la celda es 564FH?

Ejercicio:

¿Puedes escribir una subrutina que exponga el signo de exclamación en la pantalla?

* (Los octetos que forman este carácter, ya han sido dados anteriormente).



La memoria de atributos es más fácil de comprender que la memoria de pantalla porque guarda una relación uno a uno con posibles posiciones en la pantalla.

La zona de atributos está situada en la memoria desde la dirección 5800H hasta la 5AFFH. Son 768 octetos que corresponden a las 24 líneas de 32 caracteres. En otras palabras, hay un octeto de atributos por cada posición de carácter en la pantalla.

● Así, 5800H corresponde a los atributos del primer carácter de la primera línea, 5801H al segundo carácter, 5802H al tercero, ..., 581FH al 32-ésimo carácter de la primera línea.

○ De igual manera, 5820H contiene el atributo del primer carácter de la segunda línea, 5840H el de la tercera línea, ... y 5AE0H el de la última línea de la pantalla.

Sabemos que por cada posición de carácter en la pantalla, hay un octeto de atributos en la memoria de atributos, formado de la manera siguiente:

octeto de atributos	b b b b b b b b
bit 0 - 2	→ representa el color de la tinta del carácter (del 0 al 7).
bit 3 - 5	→ representa el color del papel del carácter (del 0 al 7).
bit 6	→ brillante si es 1, normal si es 0.
bit 7	→ parpadeando si es 1, normal si es 0.

Ejercicio:

¿Cuál es la dirección del octeto de atributos que corresponde al primer octeto de la parte mediana de la pantalla? ¿Cuál es la dirección del primer octeto de la tercera parte?

Veremos la respuesta en la siguiente página, pero intenta sacarlas por tí mismo.

Ejercicio:

¿Puedes escribir una subrutina que convierta una determinada dirección de pantalla a su correspondiente dirección de atributo?

eg. 4529H

De hecho, debes determinar a qué carácter de la pantalla pertenece y luego sumarlo a 5800H.



El siguiente programa te muestra un método de lograrlo:

```
LD HL, 4529H ; cargar la dirección dada en HL
LD A, H      ; cargar el octeto de mayor orden en A.
AND 18H      ; elegir sólo los bits 3 y 4; para hallar a
              ; qué porción de la pantalla pertenece
SRA A        ; desplazar a la derecha el acumulador
SRA A        ; 3 veces - ie. divide por 8. El resultado
SRA A        ; puede ser 0, 1 ó 2, dependiendo de si H
              ; fue 40H, 48H ó 58H.
ADD A, 58H   ; pasar a memoria de atributos
LD H, A      ; H contiene la dirección del atributo.
              ; ie. H = 58H, 59H ó 60H
              ; L sigue siendo el mismo
```

Puede que necesites pensar un *ratillo* sobre esto!

La forma en que el programa trabaja está relacionada con la respuesta al primer ejercicio:

1er. carácter de la primera sección de pantalla = 4000H Dirección atributo = 5000H
1er. carácter de la segunda sección de pantalla = 4800H Dirección atributo = 5900H
1er. carácter de la tercera sección de pantalla = 5000H Dirección atributo = 5A00H

2º carácter de la primera sección de pantalla = 4801H Dirección atributo = 5801H

etc...

etc...

Esto debiera hacer que las cosas estuvieran un poco más claras!



Otra de las comunicaciones en tiempo real que tu Spectrum ofrece, es el sonido.

Sería una pena si no hiciéramos un uso completo de esa facilidad.

En lenguaje-máquina hay dos formas principales de generar sonido:

- i. enviando señales al portal de salida hacia la cassette (254), durante un cierto tiempo, usando la instrucción OUT.
eg. OUT (FEH), A
- ii. poniendo HL y DE a determinados valores, y llamando a la rutina de la ROM que genera sonido.

Los parámetros de entrada son:

DE - duración en segundos * frecuencia
HL - (437.500 / frecuencia) - 30,125.

Luego

CALL 03B5H

La primera forma de generación de sonido tiene la ventaja de que es independiente de las llamadas a la ROM. Es *más corta* en términos de tiempo de ejecutar, pero... siempre hay un PERO!

- Dado que la ULA está "accesando" constantemente los primeros 16K de RAM para sacar la imagen en la pantalla, tu programa -si reside dentro de los primeros 16K-, se verá a menudo, temporalmente interrumpido.

Si el programa está generando sonido, el sonido será en ráfagas de duración impredecibles. Una *solución* es mover la parte del programa que genera sonido, a una *región* superior de la memoria (si tienes una máquina con 48K). Si no la tienes, aún puedes generar sonido con este método pero no será un *sonido puro*. Tendrás que utilizar el segundo método (llamar a la rutina en la ROM).

Como enviamos valores al *portal* 254 de salida, eso afectará al color del borde, y activará el micrófono, así como el altavoz, en función del valor que enviemos. Consulta el capítulo 23 de tu *manual de usuario* del Spectrum. Por otro lado, la rutina en ROM que genera sonido te permite utilizar el comando BEEP desde tu programa en lenguaje-máquina. Puedes pensar que la pareja de registros DE contiene el valor de la duración del sonido y HL el valor de la frecuencia. Haz *ensayos* con diferentes valores de HL y DE hasta que obtengas el sonido que desees.

La limitación de este método, estriba en que estás restringido a los sonidos que puedas crear con el comando BEEP.



Programa Monitor

Es un programa que *ayuda* a la realización de rutinas en lenguaje-máquina permitiéndote:

- i. teclear tu rutina en lenguaje-máquina, bien completamente ensamblada, bien *semi-ensamblada* (con todos los *saltos* relativos y absolutos expresados como números de línea del programa).
- ii. listar la rutina fuente que hayas escrito.
- iii. *volcar* la rutina en lenguaje-máquina en la memoria y a partir de una determinada dirección.
- iv. examinar una serie de celdillas de memoria, para comprobar su contenido.
- v. guardar en cassette la rutina fuente, tal como lo has tecleado o bien el programa ya en formato de código máquina.
- vi. cargar la rutina fuente guardada en el cassette.
- vii. ejecutar la rutina en código máquina.

PREREQUISITOS del programa

Antes de usar este programa monitor para preparar rutinas en lenguaje-máquina, debes ensamblar *a mano* tu rutina fuente. No necesitas calcular los *destinos* de los saltos, sean relativos o absolutos!!!

Tu rutina fuente no debe ocupar más de 800 octetos ni tener más de 200 instrucciones.

No puedes cargar tu rutina por debajo de la dirección 31499, con el fin de no borrar parte del programa monitor.

CONCEPTO del programa

El concepto que subyace en este programa, es permitirte tratar las instrucciones en lenguaje-máquina como líneas numeradas, de manera muy similar a un programa en BASIC.

Cada línea de tu rutina fuente (es el nombre que damos a las *líneas* de tu programa en código máquina) tiene un número de línea y hasta cuatro octetos de código máquina.



Un beneficio importante es la capacidad para *revisar* cualquier línea. La rutina fuente puede *guardarse* separadamente en cinta, permitiéndote que lo hagas a medida que progresas en el desarrollo del programa. Otra innovación de éste es la capacidad para insertar *saltos* relativos o absolutos, sin tener que calcular la dirección destino del salto, sino simplemente citando el número de línea al que deseas saltar.

Esto significa que pueden hacerse cambios sin problemas, incluso dentro del ámbito de un salto relativo.

El código máquina de tu rutina fuente, se *travasa* a la memoria mediante el comando de vuelco (DUMP); el código máquina resultante también puede guardarse (SAVE) en la memoria.

Resumen de instrucciones

Observa que la primera pregunta que el programa te hace es

"Loading address" (Dirección de carga).

Es la dirección donde quieres comenzar a cargar tu rutina en código máquina. Y no puede ser inferior a 31500.

**** Tecleando el programa ****

i. Para teclear las líneas de la rutina fuente:

(número de línea) (blanco) (hasta cuatro octetos en hexadecimal)
y luego ENTER.

eg. 1 210040 meterá la instrucción LD HL, 4000H como línea con número 1.

ii. Para revisar una línea:

(número de línea) (blanco) (tecleas los nuevos octetos) (ENTER)

eg. 1 230140 cambiará la línea de número 1 para que sea
LD HL, 4001H

iii. para eliminar una línea:

(número de línea) (ENTER)

eg. 1 (ENTER) elimina la línea con número 1.



- iv. para escribir un salto relativo o absoluto:

(número de línea) (blanco) (instrucción de salto) ("I") (número de línea) (ENTER)

eg. 1 C3I2 representa la instrucción JP a la línea 2
2 I8I1 representa la instrucción JR a la línea 1

**** COMANDOS ****

- i. dump (VOLCAR)

- ⊗ *vuelca en la memoria, el código máquina de tu listado, a partir de la dirección de carga que hayas especificado.*
- ⊗ *debe hacerse antes de nular tu programa.*

abreviatura: du

- ii. exit (SALIR)

- ⊗ *sales del monitor de rutinas y vuelves al BASIC*

abreviatura: ex

- iii. list (LISTAR)

- ⊗ *lista las primeras 22 instrucciones de tu rutina fuente.*
- ⊗ *pulsa cualquier tecla excepto "m" y "break" para continuar listando la rutina.*

abreviatura: li

list# (LISTAR)

- ⊗ *lista 22 líneas de tu rutina fuente a partir de la línea con número # (que debe estar entre 1 y 200 inclusive).*

abreviatura: NO TIENE ABREVIATURA

- iv. load (CARGAR)

- ⊗ *carga tu rutina fuente desde el cassette, sustituyendo a la rutina que hubiera en memoria.*

abreviatura: lo

- v. mem (MEMORIA)

pregunta: "Starting address" (Dirección de comienzo)

- ⊗ *teclea la dirección de memoria a partir de la que deseas comenzar a examinar la memoria.*
- ⊗ *puede ser desde 0 a 32767 para un Spectrum con 16K, o desde 0 a 65535 para uno con 48K.*
- ⊗ *pulsa "m" para dejar de examinar la memoria.*



abreviatura: me

vi. new (NUEVO)

- ⊗ limpia la rutina que haya en memoria y sigue con el monitor
- ⊗ es útil cuando deseas comenzar a codificar otro nuevo programa.

abreviatura: ne

vii. run (RULAR)

- ⊗ rula tu rutina en lenguaje-máquina, a partir de la dirección de carga que especificaste cuando empezaste con el monitor de rutinas, o cuando cargaste una nueva rutina fuente.

abreviatura: ru

viii. save (GUARDAR)

- ⊗ guarda en el cassette tu rutina fuente, o ya traducida a código máquina.

Pregunta: "enter name" (meter nombre)
y le contestas con el nombre que vas a usar para el programa.

Pregunta: "source or Machine code" (s or m)
pulsas "s" para guardar la rutina fuente (source);
pulsas "m" para guardar el código máquina (machine).
Preparas el cassette y pulsas cualquier tecla,
asegurándote que los terminales del cassette están
adecuadamente conectados.

abreviatura: sa

OBSERVACIONES

1. Si no quieres que después de *rular* el programa te entregue el resultado del registro BC, cambia la línea 3090 a:

```
3090 IF k$ = "ru" THEN LET L = USR R
```
2. Para *reiniciar* el monitor de rutinas:
bien usar RUN con lo que todas las variables volverán a inicializarse
o bien usa GOTO 2020 que te mostrará la pregunta COMMAND OR LINE (###):"
3. Todos los datos numéricos que teclees excepto las instrucciones en código máquina tienen que estar en formato decimal.
4. Para que puedas *insertar* líneas adicionales en un programa, es bueno que las vayas numerando a saltos.
ie. en lugar de dar los números de líneas como 1, 2, 3, etc. dalos como 1, 5, 10, etc. Eso te hará más flexible el tecleo de la rutina.



Mete los siguientes códigos.

210040	LD HL, 4000H	; llena la pantalla
110140	LD DE, 4001H	
01FF17	LD BC, 6143	
3EFF	LD A, 0FFH	
77	LD (HL), A	
EDB0	LDIR	
3E7F	LAZO: LD A, 7FH	; cebo para la tecla BREAK
DBFE	IN A, (OFEH)	
E601	AND 1	
20F8	JR NZ, LAZO	
C9	RET	

para meter este programa, teclea usando el monitor:

(RUN)

Dirección de carga: 31500 (ENTER)
 Comando o Línea (###): 1 210040 (ENTER)
 Comando o Línea (###): 5 110140 (ENTER)
 Comando o Línea (###): 10 01ff17 (ENTER)
 Comando o Línea (###): 15 3eff (ENTER)
 Comando o Línea (###): 20 77 (ENTER)
 Comando o Línea (###): 25 edb0 (ENTER)
 Comando o Línea (###): 30 3e7f (ENTER)
 Comando o Línea (###): 35 dbfe (ENTER)
 Comando o Línea (###): 40 e601 (ENTER)
 Comando o Línea (###): 45 20130 (ENTER)

(Esto es "20" luego "f", luego "30". En otras palabras JR NZ, línea 30).

Comando o Línea (###): 50 c9 (ENTER)
 Comando o Línea (###): list (ENTER)
 Comando o Línea (###): dump (ENTER)
 Comando o Línea (###): mem (ENTER)

Dirección de comienzo: 31500 (ENTER)

m (es la tecla para dejar de examinar la memoria).

Comando o Línea (###): run (ENTER)
 (BREAK)

* Observa que sí debe haber un blanco después del número de línea.




```

100 REM máquina
110 REM monitor_de_código
120 GO TO 9000
130 DEF FN d(s$) = (s$ > "9") * (CODE s$-55)
    +(s$ <= "9") * (CODE s$-48) - (s$ > "L") * 32
140 DEF FN o(o$) = ((O$ = "ca")+(O$ = "da")
    +(O$ = "ea")+(O$ = "fa")+(O$ = "c2")
    +(O$ = "d2")+(O$ = "e2")+(O$ = "f2")
    +(O$ = "c3"))-(O$ = "38")+(O$ = "30")
    +(O$ = "28")+(O$ = "20")+(O$ = "18")
    +(O$ = "10"))
1000 REM
1010 REM INW rutina_de_impresora IRU
1020 CLS : PRINT AT ze,24; INVERSE on; FLASH on; "listando"
1030 LET F = ze : PRINT AT ze, ze;
1040 FOR J = p11 TO p12
1050 IF C$(J, on) = "_" THEN GO TO 1110
1060 PRINT TAB tr-LEN STR$ J; J; TAB fr; "_";
1070 IF C$(J, tw, on TO on) = "1"
    THEN PRINT C$(J, on)+"-"+C$(J, tw)+C$(J, tr)
    : GO TO 1090
1080 PRINT C$(J, on);"-";C$(J, tw);"-";
    :C$(J, tr);"-";C$(J, fr)
1090 LET F = F+on
1100 IF F = 22 THEN GO TO 1120
1110 NEXT J
1120 PRINT AT ze, 24; "-----"
1130 RETURN
2000 REM
2010 REM INW rutina_principal IRU
2020 INPUT "Comando_o_Linea(###) : "; A$
2030 IF A$( TO fr) = "-----" THEN GO TO mr
2040 IF A$(on) > "9" THEN GO TO 3000
2050 LET k$ = "" : FOR K = on TO fr
2060 IF A$(K TO K) = "_" THEN GO TO 2090
2070 LET k$ = k$+A$(K TO K)
2080 NEXT K
2090 IF K = 5 OR VAL k$ = ze OR VAL k$ > ln
    THEN GO TO mr
2100 LET J = VAL k$ : LET n = J
    : REM número_de_linea_es_3_octetos
2110 LET A$ = A$(K+on TO)
2120 LET k$ = ""
2130 FOR K = on TO LEN A$
2140 IF A$(K TO K) <> "-"
    THEN LET k$ = k$+A$(K TO K)
2150 NEXT K
2160 LET A$ = k$
2162 IF A$(on) = "1" THEN GO TO mr

```



```

2170 CLS : FOR I = on TO 7 STEP tw
2180 LET K = INT (I/tw+on)
2190 LET C$(J, K) = A$(I TO I+on)
2200 NEXT I
2210 IF C$(n, on) = "___" THEN GO TO 2250
2220 IF n < TP THEN LET TP = n
2230 IF n > BP THEN LET BP = n
2240 GO TO 2320
2250 IF n <> BP THEN GO TO 2280
2260 IF BP = on OR C$(BP, on) <> "___"
      THEN GO TO 2320
2270 LET BP = BP-on : GO TO 2260
2280 IF n <> TP THEN GO TO 2320
2290 IF C$(TP, on) <> "___" THEN GO TO 2320
2300 IF TP <> BP AND TP <> 1n THEN LET TP = TP+on
      : GO TO 2290
2310 LET TP = on
2320 LET pp = n
2330 IF n < TP THEN LET pp = TP : GO TO 2380
2340 LET numlp = ze
2350 IF pp = TP OR numlp = 11 THEN GO TO 2380
2360 IF C$(pp, on) <> "___"
      THEN LET numlp = numlp+on
2370 LET pp = pp-on : GO TO 2350
2380 LET p11 = pp : LET p12 = BP
2390 GO SUB 1000
      : REM saca_un_bloque_de_lifneas
2400 GO TO mr
3000 REM
3010 REM INW Comandos***** IRU
3020 LET k$ = A$( TO tw)
3030 IF k$ = "du" THEN GO TO 5000
3040 IF k$ = "ex" THEN STOP
3050 IF k$ = "li" THEN GO TO 4000
3060 IF k$ = "lo" THEN GO TO 7000
3070 IF k$ = "me" THEN GO TO 6000
3080 IF k$ = "ne" THEN RUN
3090 IF k$ = "ru" THEN PRINT USR R
3100 IF k$ = "sa" THEN GO TO 8000
3110 GO TO mr
4000 REM
4010 REM INW Rutina_de_listado***** IRU
4020 LET p11 = TP : LET p12 = BP
4030 LET n1 = CODE A$(6 TO 6)
4040 IF LEN A$ > fr AND n1 > 47 AND n1 < 58
      THEN LET p11 = VAL A$(5 TO 8)
4050 GO SUB 1000
4060 GO TO mr
5000 REM
5010 REM INW rutina_de_volcado***** IRU
5020 CLS : PRINT AT ze, 24; INK on; INVERSE on
      : FLASH on; "VOLCANO" : LET G = R

```

```

5030 PRINT AT on, ze;
5040 FOR J = TP TO BP
5050 IF C$(J, on) = "___" THEN GO TO 5470
5060 IF C$(J, tw, on TO on) <> "1" THEN GO TO 5380
5070 POKE G, ze : POKE G+on, ze : POKE G+tw, ze
      : POKE G+tr, ze
5080 LET j1 = VAL (C$(J, tw, tw TO tw)+C$(J, tr))
5090 PRINT TAB tr-LEN STR$ J; INVERSE on; J
      : TAB fr; INVERSE ze; "___"
      : C$(J, on)+"___"+C$(J, tw)+C$(J, tr)
      : "___"
5100 IF j1 < ze OR j1 > 1n THEN GO TO 5460
5110 LET CJ = FN O(C$(J, on))
5120 PRINT TAB 17-LEN STR$ j1; INVERSE on; j1
      : TAB 18; INVERSE ze; "___"; C$(j1, on)
      : "___"; C$(j1, tw); "___"; C$(j1, tr); "___"
      : C$(j1, fr);
5130 IF ABS CJ <> on THEN GO TO 5460
5140 LET dd = (j1 > J)-(j1 < J)
5150 LET ja = G : LET dp = ze
5160 IF j1 = J THEN GO TO 5270
5170 LET cl = J+dd
5180 LET n1 = ze : IF C$(cl, on) = "___"
      THEN GO TO 5220
5190 IF C$(cl, tw, on TO on) <> "1"
      THEN LET n1 = on+(C$(cl, tw) <> "___")
      + (C$(cl, tr) <> "___")
      + (C$(cl, fr) <> "___")
      : GO TO 5220
5200 LET TJ = FN o(C$(cl, on))
5210 LET n1 = (TJ = on)*tr+(TJ = -on)*tw
5220 IF cl = j1 AND dd > ze THEN GO TO 5270
5230 LET dp = dp+n1
5240 IF cl = j1 THEN GO TO 5270
5250 LET cl = cl+dd
5260 GO TO 5180
5270 IF CJ = on THEN LET ja = ja+dd*dp+(dd > ze)*tr
      : GO TO 5310
5280 IF dd > ze THEN LET dp = dp+2
5290 IF dp > 126 AND dd < ze THEN GO TO 5460
5300 IF dp > 129 AND dd > ze THEN GO TO 5460
5310 LET V = 16*FN d(C$(J, on, on TO on))
      + FN d(C$(J, on, tw TO tw))
5320 POKE G, V: LET G = G+on
5330 IF CJ = on THEN POKE G, ja-INT (ja/qk)*qk
      : LET G = G+on : POKE G, INT (ja/qk)
      : LET G = G+on : GO TO 5360
5340 IF dd < ze THEN LET dp = -dp
5350 LET dp = dp-tw : POKE G, dp : LET G = G+on
5360 PRINT "ok"
5370 GO TO 5470
5380 FOR I = on TO 7 STEP tw

```



```

5390 LET K = INT (I/tw+on)
5400 LET V = 16* FN d(C$(J, K, on TO on))
      + FN d(C$(J, K, tw TO tw))
5410 IF V < ze THEN GO TO 5440
5420 POKE G, V
5430 LET G = G+on
5440 NEXT I
5450 GO TO 5470
5460 PRINT "***"
5470 NEXT J
5480 PRINT AT ze, 24; "_____"
      : GO TO mr
6000 REM
6010 REM INW Mostrando_memoria***** IRU
6020 INPUT "Direccion_de_comienzo :_"; dm
6030 CLS : PRINT AT ze, ze;
6040 LET G = dm : LET F = ze
6050 LET F = F+on
      : PRINT TAB 5- LEN STR$ G; G ; TAB 6;
6060 FOR I = on TO fr
6070 LET V = PEEK G
6080 LET H = INT (V/16)
6090 LET L = V-16*H
6100 PRINT D$(H+on); D$(L+on); "_";
6110 LET G = G+on
6120 NEXT I
6130 PRINT "_"
6140 IF F <> 22 THEN GO TO 6050
6150 LET K$ = INKEY$ : IF K$ = "" THEN GO TO 6150
6160 IF K$ <> "m" AND K$ <> "M" THEN LET F = ze
      : POKE 23692, qk-on : GO TO 6050
6200 POKE 23692, on : PAUSE 20 : GO TO mr
7000 REM
7010 REM INW Carga_en_memoria***** IRU
7020 CLS
7030 INPUT
      "Carga_tabla_:Pulsa_una_tecla____cuando_preparado._"
      : k$
7040 PRINT AT ze, 25; INVERSE on; FLASH on; "CARGANDO"
7050 LOAD "source" DATA C$( )
7060 FOR I = on TO ln
7070 LET TP = I
7080 IF C$(I, on) <> "_" THEN GO TO 7100
7090 NEXT I
7100 FOR I = ln TO on STEP -1
7110 LET BP = I
7120 IF C$(I, on) <> "_" THEN GO TO 7140
7130 NEXT I
7140 PRINT AT ze, 24; "_____"
7150 GO TO 9150
8000 REM
8010 REM INW Guarda_en_cassette***** IRU

```



```

8020 INPUT "Dime_nombre :_"; n$
8030 IF n$ = "" THEN GO TO 8020
8040 INPUT
      "Fuente_o_Código_máquina_(s_o_m)"
      : k$
8050 IF k$ <> "s" AND k$ <> "m" THEN GO TO 8040
8060 IF k$ = "s" THEN SAVE n$ DATA C$( ) : GO TO mr
8070 INPUT "Dirección_de_comienzo :_"; ss
8080 INPUT "Dirección_de_final... :_"; sf
8090 LET sb = sf-ss+on
8100 SAVE n$ CODE ss, sb
8110 GO TO mr
9000 REM
9010 REM inicialización
9020 LET ze = PI - PI : LET on = PI / PI
      : LET tw = on+on : LET tr = on+tw
      : LET fr = tw+tw : LET qk = 256
      : LET mr = 2020 : LET ln = 200
9025 BORDER 7 : PAPER 7 : INK on : INVERSE ze
      : OVER ze : FLASH ze : BRIGHT ze
      : BEEP .25, 24 : BEEP .25, 12
9030 DIM A$(15) : DIM O$(tw)
9040 LET TP = ln : LET BP = on
      : REM las_200_líneas_posibles
9050 DIM C$(ln, fr, tw) : REM códigos
9060 PRINT AT ze, 19; INVERSE on; FLASH on
      : "INICIALIZANDO"
9070 FOR I = on TO ln
9080 FOR J = on TO fr
9090 LET C$(I,J) = "_"
9100 NEXT J
9110 BEEP .01, 20
9120 NEXT I
9130 PRINT AT ze, 19; "-----"
9140 LET D$ = "0123456789ABCDEF"
9150 CLS : PRINT "Dirección_más_baja :_"; 31500
9160 INPUT "Dirección_de_carga :_"; R : PAUSE 20
9170 IF R < 31500 THEN GO TO 9160
9180 CLS : GO TO mr

```



CARGADOR HEXADECIMAL

Este programa BASIC puede por sí mismo, ser un monitor, ya que permite escribir los códigos hexadecimales en la memoria, listar la memoria, mover bloques de memoria, guardar bloques en el cassette y cargar desde el cassette a la memoria.

Por otro lado, lo podemos usar como cargador y *montador* de las rutinas que hemos creado con el monitor de rutinas anterior, ya que el monitor solamente puede usarse para meter módulos *pequeños* de menos de 800 octetos y con menos de 200 instrucciones.

Así, para programas *mayores*, usamos el monitor de rutinas para desarrollar los módulos y guardamos cada módulo en cassette como bloques de código máquina. Luego usamos el cargador hexadecimal, que es un programa BASIC mucho más pequeño, para cargar esos módulos en la memoria y *montarlos* juntos, moviendo los módulos a sus correspondientes celdillas de memoria.

Aplicaremos realmente esta técnica a medida que desarrollemos el programa LA RANA URBANA.

Conceptos del cargador hexadecimal

El concepto de este programa es extremadamente simple. El programa cargador coloca la *cima* o *tope* de la RAM (RAMTOP) del intérprete BASIC en la dirección 26999.

Eso significa que puedes meter tus programas en lenguaje-máquina en cualquiera de las direcciones entre 27000 hasta 32578 para un Spectrum con 16K, y entre 27000 y 65346 para uno con 48K.

- El cargador es una forma directa de observación del código-máquina: A partir del mensaje "Start of machine code area" Comienzo de la zona de código máquina)

Ofrece las funciones básicas de observación como:

→ ESCRIBIR en memoria en formato hexadecimal
→ GUARDAR de la memoria a la cassette
→ CARGAR de la cassette a la memoria
→ LISTAR la memoria a partir de una dirección de comienzo
→ MOVER contenidos de memoria desde una celdilla a otra.

Resumen de instrucciones

1. WRITE (escribir)

escribe en la memoria código con formato HEX.

Procedimiento:

- a. como respuesta a la pregunta, teclea en formato decimal la dirección de memoria a partir de la que quieres comenzar a escribir. La dirección está limitada a 27000 - 32578 para 16K.
27000 - 65346 para 48K.



- b. Mete los códigos en formato hexadecimal.
- c. Pulsa "m" para volver al *menú* principal.

2. SAVE (guardar)

guarda en el cassette un bloque de memoria.

Procedimiento:

- a. La dirección de memoria a partir de la que comienza a guardar el contenido, puede ser cualquier dirección
0 - 32767 para 16K.
0 - 65535 para 48K.
- b. Teclea el número de octetos a guardar.
- c. Teclea el nombre del módulo a guardar.
- d. Pulsa cualquier tecla cuando esté preparado el cassette.
- e. Tienes la opción de *verificar* si se ha guardado correctamente el módulo en el cassette. Es bueno verificarlo para garantizar que durante el trasvase al cassette no ha habido errores.

3. LOAD (cargar)

carga en la memoria el módulo guardado en el cassette.

Procedimiento:

- a. Teclea la dirección a partir de la cual quieres cargar el módulo. La dirección está limitada a los mismos valores en el caso de escritura.
- b. Teclea el nombre que usaste cuando guardaste el módulo. Si no estás seguro del nombre, simplemente pulsa ENTER.

(4. LIST (listar))

muestra el contenido de la memoria a partir de una dirección.

Procedimiento:

- a. Teclea la dirección de comienzo del listado.
Pueden ser las mismas direcciones que en el caso de guardar.
- b. Pulsa cualquier tecla para seguir listando.
- c. Teclea "m" para volver al menú principal.

5. MOVE (mover)

mueve el contenido de la memoria desde la dirección inicial hasta la dirección final del bloque, a unas nuevas direcciones de memoria.

Procedimiento:

- a. Con FROM, teclea cualquier dirección. Es el inicio del bloque a mover.
- b. Con UNTIL, teclea cualquier dirección. Es el final del bloque a mover.
- c. Con TO, teclea la dirección donde va a *quedar* el bloque. Los valores permitidos son los mismos que en escritura.
- d. Utilizando este comando puedes incluso copiar la ROM en la RAM.

eg. Move from memory: 0 (ENTER)
Move until memory: 1000 (ENTER)
Move to memory: 32000 (ENTER)

esto moverá el bloque entre 0 y 1000 (en la ROM) a la dirección 32000 y siguientes de la RAM.

OBSERVACIONES: Cualquiera de los tecleos en los comandos anteriores que traspase la gama de direcciones permitida, dará como resultado que se vuelva a hacer la pregunta.

Ejercicio:

Intenta usar este cargador hexadecimal para meter el módulo que hemos desarrollado anteriormente con el monitor de programas.




```

100 REM
110 REM programa monitor
120 CLEAR 26999 : LET ze = PI - PI
   : LET on = PI / PI : LET tw = on+on
   : LET qk = 256 : LET lm = 27000
   : LET mr = 140 : LET wl = 340
130 GO SUB 2000
140 CLS
   : PRINT "Comienzo área código máquina = "
   : lm
150 PRINT "menu" : PRINT
   : PRINT
      "Escribe código máquina.....1"
160 PRINT
   : PRINT
      "Guarda código máquina.....2"
170 PRINT
   : PRINT
      "Carga código máquina.....3"
180 PRINT
   : PRINT
      "Lista código máquina.....4"
190 PRINT
   : PRINT
      "Ubica código máquina.....5"
200 PRINT
   : PRINT
      "Por favor pulsa la tecla correcta."
210 LET g$ = INKEY$
220 IF g$ = "m" OR g$ = "M" THEN STOP
230 IF g$ = "" OR g$ < "1" OR g$ > "5"
      THEN GO TO 210
240 CLS
   : PRINT "Comienzo área código máquina = "
   : lm
250 GO TO 300* VAL g$
300 REM INV Escribe***** IBU
310 INPUT "Escribe a partir de dirección :"; d
320 IF d > mr OR d < lm THEN GO TO 310
330 PRINT : PRINT "Escribe Dirección :_"; d
   : PRINT "Para volver al menú pulsa "m""
340 LET a$ = ""
350 IF a$ = "" THEN INPUT "Teclea código hex. : "
   : a$
360 IF a$(on) = "m" OR a$(on) = "M"
      THEN GO TO mr
370 IF LEN a$/tw <> INT ( LEN a$/tw)
      THEN PRINT "Dato incorrecto ";
   : GO TO wl

```



```

380 LET c = ze
390 FOR f = 16 TO on STEP -15
400 LET a = CODE a$(f=16)+tw*(f=on)
410 IF a < 48 OR a > 102 OR (a > 57 AND a < 65)
    OR (a > 70 AND a < 97)
    THEN PRINT "Dato incorrecto ";
    : GO TO w1
420 LET c = c+f*((a < 58)*(a-48)
    +(a > 64 AND a < 71)*(a-55)+(a > 96)*(a-87))
430 NEXT f : POKE d, c : LET d = d+on
440 PRINT a$(TO tw); "___";
450 LET a$ = a$(3 TO )
460 IF d = UDG
    THEN PRINT
    "Aviso :ésto es área de gráficos
    de usuario!";
    : GO TO w1
470 IF d = UDG-20
    THEN PRINT
    "Aviso :ésto es área de rutinas!";
    : GO TO w1
480 GO TO w1+on
600 REM INW Guarda***** IRU
610 INPUT "Guarda C.M. /desde/ dirección :"; a
620 INPUT "Número de octetos a guardar :"; n
630 INPUT "Nombre de la rutina :"; a$
640 SAVE a$ CODE a, n
650 PRINT "Deseas verificarlo?"
660 INPUT v$
670 IF v$ <> "y" THEN GO TO mr
680 PRINT "Rebobina y pulsa "PLAY"."
690 VERIFY a$ CODE a, n
700 PRINT "O.K." : PAUSE 50
710 GO TO mr
900 REM INW Carga***** IRU
910 INPUT
    "Carga C.M. a partir dirección : "
    ; a
920 IF a > mm OR a < 1m THEN GO TO 910
930 INPUT "Nombre del programa :"; a$
940 PRINT "Pulsa "PLAY" en cassette"
950 LOAD a$ CODE a : GO TO mr
1200 REM INW Lista***** IRU
1210 LET a$ = "0123456789ABCDEF"
1220 INPUT "Lista Dirección :"; d
1230 PRINT "Pulsa "M" para volver a Menu."
1240 LET a = INT ( PEEK d/16)
    : LET b = PEEK d-16* INT ( PEEK d/16)
1250 PRINT d; TAB 7; a$(a+on); a$(b+on)
1260 LET d = d+on
1270 IF INKEY$ = "m" OR INKEY$ = "M" THEN GO TO mr
1280 GO TO 1240
1500 REM INW Mudar bloques***** IRU

```



```

1510 INPUT "Bloque desde memoria : "; fm
1520 INPUT "..... hasta memoria : "; um
1530 INPUT "Ubicar en memoria : "; tm
1540 IF tm > fm THEN GO TO 1610
1550 LET mp = tm
1560 FOR I = fm TO um
1570 POKE mp, PEEK I
1580 LET mp = mp+on
1590 NEXT I
1600 GO TO mr
1610 LET mp = um+tm-fm
1620 FOR I = um TO fm STEP -on
1630 POKE mp, PEEK I
1640 LET mp = mp-on
1650 NEXT I
1660 GO TO mr
2000 LET RT = PEEK 23732+qk* PEEK 23733
2010 IF RT = 65535 THEN LET mm = 65347
: LET UDG = 65367
2020 IF RT = 32767 THEN LET mm = 32579
: LET UDG = 32599
2030 LET n1 = INT (UDG/qk)
2040 POKE 23675, UDG-n1*qk : POKE 23676, n1
2050 RETURN

```



Pasando el programa

Te presentamos el programa para ese popular juego en que unas ranas -en su camino a casa- deben cruzar de un lado a otro de una autopista, brincando cuando el camino está libre de obstáculos. Hay camiones, coches y motocicletas; además de coches policiales que patrullan inesperadamente por la autopista.

La puntuación lograda es función del número de brinco que la rana dé.

Debes comprender el juego muy claramente, porque tú vas a ser el programador.

Esto es simplemente la etapa de definición del problema. A menos que podamos definir y comprender claramente el problema, será muy difícil que sepamos hacia dónde nos encaminamos cuando estemos en las etapas posteriores del desarrollo del programa.

Estructurando el programa

- Podemos aplicar lo que hemos aprendido sobre programación modular de arriba hacia abajo: Empezamos a partir del nivel más alto y desglosamos el programa en módulos racionales bien definidos.

Son los que siguen:

1. INICIO

Efectuará todas las tareas de preparación y apresto.

2. TRAFICO

Controlará la circulación sobre la autopista.

Puede desglosarse a su vez en:

- i. Flujo regular de camiones, coches y motos.
- ii. Flujo irregular de los coches policiales.

3. RANITAS

Seguirá el movimiento de la rana, comprobando si choca o llega a su casita.

4. GENERAL

Calculará y expondrá la puntuación y vigilará la terminación.

5. TERMINO

Efectuará las tareas domésticas necesarias para regresar a BASIC.



Desarrollando el programa

El desarrollo del programa lo comentaremos en seis etapas; lo que sigue muy estrechamente a los cinco módulos mencionados!

En cada etapa probaremos -para garantizar que funciona apropiadamente; antes de pasar a desarrollar la siguiente.

Las seis etapas serán:

1. Diseño de las bases de datos:

Lo que implica dibujar las diversas figuras, crear las tablas con los datos de cada figura y decidir las variables sobre las que el programa operará.

2. Inicialización:

Consiste en la preparación de la imagen y en la asignación de valores iniciales a las diferentes variables.

3. Gobierno del tráfico:

Aquí sólo desarrollaremos el flujo regular de vehículos.
(La aparición irregular del coche policial exige diferente lógica).

4. Inclusión de policías:

Analizamos y probamos lo referente a coches policiales.

5. Cruce de autopista:

Movemos a la rana -borrando la figura vieja y dibujando la nueva-, comprobando si hay o no, estacazo; calculamos la puntuación, etc.

6. Supervisión general

Consta de la actualización de la puntuación máxima, la reanudación o terminación del juego, y de la vuelta al sistema operativo.

Antes de proceder con la primera etapa del desarrollo, te proponemos un programa en BASIC que suma los contenidos de las celdillas de un bloque de memoria determinado, como una forma simple y eficaz de cotejar los datos que tecleas.

```
9000      REM comprobando bloques
9010      REM mediante suma
9020      INPUT "Desde dirección: "; d
9030      INPUT "Hasta dirección: "; h
9040      LET s = 0
9050      FOR I = d TO h
```



9060	LET s = s + PEEK I
9070	NEXT I
9080	PRINT "cotejo: "; s
9090	GOTO 9020

Teclea la dirección donde empieza, y luego la de donde termina el bloque que deseas comprobar. El programa hará la suma -en decimal-, del contenido de las celdillas y la presentará como el valor de "cotejo". Deberá coincidir con el que figura en las tablas.



ETAPA PRIMERA: BASES DE DATOS

****Dibujo de las figuras****

Como hay tráfico en los dos sentidos, dibujaremos dos figuras de camión: uno hacia la izquierda y otro hacia la derecha; etc.

Para la rana: hay cuatro posibles direcciones y por ende una figura distinta para cada dirección.

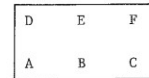
Adoptaremos el siguiente convenio para la posición de un objeto y para su figura:

Si la forma de un objeto ocupa cuatro pictos, los dispondremos así:



C	D
A	B

Si la forma del objeto ocupa seis pictos, los colocaremos así:



D	E	F
A	B	C

y el puntero de posición señalará siempre al picto marcado con "A".

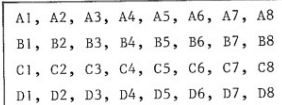
REMemora que:

En una pantalla con 24 filas y 32 columnas hay $24 \times 32 = 768$ pictos.

Como los pictos de tu Spectrum son tramas de 8 hilos horizontales cruzados por otras ocho líneas horizontales, para dibujar algo en ellos, debemos usar ocho octetos, con los "1" y "0" adecuados.

Y si acordamos empezar a pintar en cada picto de arriba hacia abajo y de izquierda a derecha la tabla que define una figura será:

FIGDE 4



A1, A2, A3, A4, A5, A6, A7, A8
B1, B2, B3, B4, B5, B6, B7, B8
C1, C2, C3, C4, C5, C6, C7, C8
D1, D2, D3, D4, D5, D6, D7, D8

REMemora que:

A cada picto le corresponde un octeto de atributos con los valores del color de la tinta, del papel, del brillo y del parpadeo.

Por tanto, para definir los atributos de una figura que ocupe cuatro pictos, almacenamos después de los 32 octetos que fijan su forma, los 4 octetos que determinan sus atributos.



**** Cargando las tablas de las figuras ****

<i>Etiqueta</i>	<i>Línea#</i>	<i>Desde(H)</i>	<i>Hasta(H)</i>	<i>Desde(D)</i>	<i>Hasta(D)</i>	<i>Cotejo</i>
FRGSHP	120	69AFH	6A36H	27055	27190	18085
LBIKE	340	6A37H	6A76H	27191	27254	3647
LBATT	430	6A77H	6A7EH	27255	27262	28
RBIKE	460	6A7FH	6ABEH	27263	27326	3355
RBATT	560	6ABFH	6AC6H	27327	27334	28
LCAR	600	6AC7H	6B26H	27335	27430	5073
LCATT	730	6B27H	6B32H	27431	27442	36
RCAR	770	6B33H	6B92H	27443	27538	4902
RCATT	900	6B93H	6B9EH	27539	27550	12
LTRUCK	940	6B9FH	6C76H	27551	27766	22023
LTATT	1230	6C77H	6C91H	27767	27793	87
RTRUCK	1280	6C92H	6D69H	27794	28009	21834
RTATT	1570	6D6AH	6D84H	28010	28036	87
BLANK	1620	6D85H	6D88H	28037	28040	0

Este módulo ocupa desde 27055 hasta 28040 y tiene, por tanto, 986 octetos.
El valor de cotejo es de 79197. El nombre asignado es SHAPDB.

Todos los objetos anteriores, excepto la rana, constan de los octetos con los valores de la forma del objeto seguidos por dos octetos con los valores de los atributos.

La rana no corresponde a ese formato porque hemos decidido que la rana tenga sólo un color en cada momento: verde cuando esté viva, roja cuando está muriendo y amarilla cuando alcance su hogar.

En este juego, usamos negro (0) para color del papel excepto para los setos de la autopista y la línea informativa de la parte superior donde usamos blanco (7) como color del papel.

Para objetos que sólo se mueven en la autopista, el atributo del papel será 0 y el color de la tinta será el dado en su base de datos. Antes de meter en la memoria la base de datos de las figuras y guardarla sobre cassette, se supone que has entendido la representación de los caracteres gráficos en la memoria.

Ahora, explicaremos el listado en ensamblador utilizando el ejemplo de la rana FROGI que comienza en la línea 160.

En la línea 160 verás:

69B7 6F 160 FROGI DB 111, 15, 31, 159, 220, 216, 120, 48
 OF 1F 9F DC D8 78 30

69B7 es la dirección de la celdilla en formato hexadecimal.

6F es el comienzo de los 8 octetos de la presente instrucción DB en valor hexadecimal.



★ El valor hexadecimal de los siguientes 7 octetos, está en la siguiente línea -entre la línea 160 y línea 170.
ie. 0FH, 1FH, 9FH, DCH, D8H, 78H, 30H.

160	es el número de línea en el listado ensamblador.
FROG1	es la etiqueta. Es sólo para tener más ventajas.
DB	es un nemónimo. Significa que lo que sigue es una secuencia de octetos (similar a DATA en lenguaje BASIC).

111, 15, 31, 159, 220, 216, 120, 48
son los octetos a cargar en la memoria.

Construyamos ahora la figura FROG1.

00	00000000	00000000	00
01	00000001	10000000	80
23	00100011	11000100	C4
25	00100101	10100100	A4
6F	01101111	11110110	F6
4F	01001111	11110010	F2
DF	11011111	11111011	FB
FF	11111111	11111111	FF
6F	01101111	11110110	F6
0F	00001111	11110000	F0
1F	00011111	11111000	F8
9F	10011111	11111001	F9
DC	11011100	00111011	3B
D8	11011000	00011011	1B
78	01111000	00011110	1E
30	00110000	00001100	0C

Recuerda:

- i. que dibujamos primero la parte inferior de izquierda a derecha.
- ii. Luego dibujamos la siguiente parte superior.
- iii. Para cada carácter, dibujamos los 8 octetos de arriba hacia abajo.
- iv. Luego y como final, rellenamos los atributos.

En la línea 120, la etiqueta FRGSHP, define uno de los cuatro punteros correspondientes a las 4 formas de la rana. En el programa, podremos encontrar la forma adecuada, dando la dirección de la rana.



DEFW

es un nemónimo que significa que vamos a definir 2 octetos "nn".
Primer el octeto menos significativo y luego el octeto más significativo.

Usa el programa CARGAHEX para meter desde la línea 120 hasta la 1590 del listado provisto. Sólo tienes que teclear los octetos hexadecimales que se muestran en la columna dos.

Recuerda que antes de continuar con la siguiente etapa, has de guardar y verificar el código que has metido.

Tabla de Tráfico

Hemos decidido que habrá un flujo regular de 6 vehículos por los dos carriles de la autopista, que estará distribuido aleatoriamente.

La tabla de tráfico guardará información sobre el estado actual del tráfico; tal y como se está observando en la pantalla.

Por ejemplo, por cada vehículo, necesitamos saber:

- -Existencia,
- -ciclo de movimiento,
- -dirección de movimiento (si está parcialmente en la pantalla o no),
- -puntero de la figura,
- -puntero de los atributos,
- -número de filas que el vehículo ocupa, y
- -número de columnas ocupadas.

Las seis primeras partes de la tabla -desde la línea 1710 del programa hasta la 2040- corresponden a los 6 vehículos que van a estar en la autopista en un momento dado. Cuando cualquiera de los vehículos se sale de la autopista, se generará otro vehículo aleatoriamente.

Una forma simple, es preparar la información inicial de cada uno de los posibles vehículos y almacenarla en memoria. Cuando se genera un nuevo vehículo, vamos a las celdillas de memoria correspondientes y cambiamos los valores de la tabla de tráfico.

Aplicaremos el mismo principio al coche policial y a la rana.

Por tanto, cuando construimos la tabla de tráfico, no necesitamos asignar valores a las variables que la componen, ya que será inicializada por el programa.

Formato: para los 6 vehículos presentes, la rana y el coche de policía:

Existencia	DEFB	1 octeto
Cuenta del ciclo	DEFB	1 octeto
Dirección	DEFB	1 octeto
Real/abstracto	DEFB	1 octeto
Posición		
en pantalla	DEFW	2 octetos

Puntero de forma	DEFW	2 octetos
Puntero a atributo	DEFW	2 octetos
Filas	DEFB	1 octeto
Columnas	DEFB	1 octeto
TOTAL		12 octetos

<u>Etiqueta</u>	<u>Línea#</u>	<u>Desde(H)</u>	<u>Hasta(H)</u>	<u>Desde(D)</u>	<u>Hasta(D)</u>
OB1EXT	1710	6E25H	6E30H	28197	28208
OB2EXT	1800	6E31H	6E3CH	28209	28220
OB3EXT	1850	6E3DH	6E48H	28221	28232
OB4EXT	1900	6E49H	6E54H	28233	28244
OB5EXT	1950	6E55H	6E60H	28245	28256
OB6EXT	2000	6E61H	6E6CH	28257	28268
PCAREXT	2070	6E6DH	6E78H	28269	28280
FRGEXT	2180	6E79H	6E80H	28281	28288

Como hemos mencionado, éstos son sólo almacenamientos temporales de trabajo, la información que contienen cambia a medida que transcurre el juego.

Hay otras dos áreas importantes de almacenamiento. Se utilizan para guardar lo que está "detrás" de la rana y del coche de la policía, respectivamente.

<u>Etiqueta</u>	<u>Línea#</u>	<u>Desde(H)</u>	<u>Hasta(H)</u>	<u>Desde(D)</u>	<u>Hasta(D)</u>
FRGSTR	1650	6D89H	6DACH	28041	28076
PCSTR	1660	6DADH	6F24H	28077	28196

No necesitamos definir ninguna de estas celdillas -únicamente reservarlas-. Sólo necesitamos construir la siguiente base de datos.

La base de datos de los objetos, está organizada de la siguiente forma:

FRGDB	base de datos de la rana
DBINDEX	indicador de la base de datos de otros objetos
RBDB	base de datos de motocicleta a derechas
LBDB	base de datos de motocicleta a izquierdas
RCDB	base de datos de coche a derechas
LCDB	base de datos de coche a izquierdas
RTDB	base de datos de camión a derechas
LTDB	base de datos de camión a izquierdas
LPCDB	base de datos de coche de policía a izquierdas
LPCATT	base de datos de atributos de policía a izquierdas
RPCDB	base de datos de coche de policía a derechas
RPCATT	base de datos de atributos de policía a derechas.

Etiqueta	Línea#	Desde(H)	Hasta(H)	Desde(D)	Hasta(H)	Cotejo
PRCDB	2260	6E81H	6E88H	28289	28296	561
DBINDEX	2320	6E89H	6E94H	28297	28308	1734
RBDB	2400	6E95H	6EA0H	28309	28320	640
LBDB	2470	6EA1H	6EACH	28321	28332	692
RCDB	2540	6EADH	6EB8H	28333	28344	523
LCDB	2610	6EB9H	6EC4H	28345	28356	760
RTDB	2680	6EC5H	6ED0H	28357	28368	584
LTDB	2750	6ED1H	6EDCH	28369	28380	809
LPCDB	2820	6EDDH	6EE8H	28381	28392	955
LPCATT	2890	6EE9H	6EF4H	28393	28404	30
RPCDB	2930	6EF5H	6F00H	28405	28416	379
RPCATT	3000	6F01H	6F0CH	28417	28428	30

Módulo desde 28289 hasta 28428, 140 octetos, cotejo 7697.

El nombre que se sugiere es "objdb" (base de datos de los objetos). Sabemos que todos los objetos excepto la rana, tienen una base de datos con 12 octetos.

El significado y contenido de cada uno de estos octetos es:

Existencia (1 octeto)

- a cero cuando el objeto es no-existente.
- al valor n, siendo n - 1, el número de ciclos que el objeto debe esperar antes de que se le permita mover:

el valor de n para	moto izquierda y derecha	es 2
	coche izquierda y derecha	es 3
	camión izquierda y derecha	es 6
	coche de policía	es 1
	rana	es 3

en otras palabras, el coche de policía se mueve en cada ciclo, la moto cada dos ciclos, etc.

Cuenta del ciclo (1 octeto)

- inicialmente a 1, para que esté preparado para moverse inmediatamente, y se decrementa en cada ciclo.
- cuando llega a 0, se permitirá mover al objeto y la cuenta será reinicializada con el valor que haya en el octeto de existencia.

Dirección (1 octeto)

- todo el tráfico de izquierda a derechas (ie. el tráfico del carril superior) tendrá 0 como valor de dirección.
- todo el tráfico de derecha a izquierda (ie. el tráfico del carril inferior) tendrá 1 como valor de dirección.



Banderín abstracto/real (1 octeto)

- define si el objeto está parcialmente fuera de la pantalla.
- todo el tráfico de izquierda a derecha comenzará con el valor 0 (abstracto), y cambiará a 1 cuando su posición coincida con la 4820H de la pantalla.
- todo el tráfico de derecha a izquierda comenzará con este banderín en valor 1 (real); el objeto tiene una posición que apunta a la pantalla (ie. 48DFH); a medida que el tráfico se mueve fuera de la pantalla, ie. cuando el puntero de posición pasa a 48C0H a 48BFH, cambiará de real a abstracto.

Puntero de posición (2 octetos)

- un puntero de 2 octetos guardando la posición corriente del objeto en la pantalla.

Puntero de forma (2 octetos)

- un puntero de 2 octetos apuntando a la base de datos que define la forma del objeto.

Puntero de Atributos (2 octetos)

- un puntero de 2 octetos apuntando a la base de datos con los atributos del objeto.

Filas (1 octeto)

- define cuantas filas ocupa la forma del objeto.

Columnas (1 octeto)

- define cuantas columnas ocupa la forma del objeto.
- este valor incluye dos columnas de blancos, una en cada extremo del objeto, a fin de evitar que el tráfico se acerque uno a otro.

Ahora puedes teclear la base de datos que permite inicializar los objetos; son los números de línea 2270 a 3010 del listado.

Para meter este módulo, puedes utilizar el monitor o el programa CARGAHEX. Si usas el monitor, recuerda que has de guardar el listado fuente, así como el listado de volcado en código máquina.

Base de datos general

Hasta ahora ya hemos cubierto la base de datos desde 69AFH a 6F0CH (27055 a 28428).



Ahora vamos a construir el resto de la base de datos que denominaremos "base de datos general".

Está organizada de esta manera:

línea $\begin{cases} 500 & \text{a } 630 & \text{SONIDO} \\ 660 & \text{a } 690 & \text{MENSAJE Puntuacion} \\ 720 & \text{a } 1210 & \text{GENERAL} \end{cases}$

<u>Etiqueta</u>	<u>Línea#</u>	<u>Desde(H)</u>	<u>Hasta(H)</u>	<u>Desde(D)</u>	<u>Hasta(D)</u>	<u>Cotejo</u>
PCTON1	500	6F0DH	6F10H	28429	28432	282
PCTON2	510	6F11H	6F14H	28433	28436	166
HOMTON	540	6F15H	6F3CH	28437	28476	2565
SCRMS1	660	6F3DH	6F42H	28477	28482	540
SCORE	670	6F43H	6F48H	28483	28488	280
SCRMS2	680	6F49H	6F53H	28489	28499	732
HISCR	690	6F54H	6F58H	28500	28504	240

Módulo desde 28429 a 28504, 76 octetos, cotejo 4813.

El nombre que se sugiere es "gendb" (base de datos general).

Sólo necesitas meter desde la línea 500 hasta la línea 690.

Desde la línea 720 hasta la línea 1210, memoria 6F59H a 6F82H (2805 a 28546), están todas las variables utilizadas por el programa.

Las líneas de la 1100 a la 1150 son instrucciones con el nemónimo EQU; lo que asigna un valor a la etiqueta correspondiente y es usado por el programa ensamblador. Tu no tienes que meter nada.

Conclusión:

Ahora ya hemos cubierto toda el área de la base de datos desde la dirección 69AFH a la 6F82H (27055 a 28546).

Examina todos los módulos que hemos construido, sus nombres y la memoria que ocupan; antes de proceder a la siguiente etapa de desarrollo del programa.

Hasta ahora habrás desarrollado 3 módulos:

<u>nombre</u>	<u>desde</u>	<u>hasta</u>	<u>longitud</u>	<u>cotejo</u>
shpdb	27055	28040	986	79197
objdb	28289	28428	140	7697
gendb	28429	28504	76	4818

* Nota que la base de datos ocupa casi 1400 octetos!!



ETAPA SEGUNDA: INICIALIZACION

****Apresto de la pantalla****

En este módulo construimos y mostramos la autopista, el marcador de puntos, la rana; así como damos el valor inicial a todas las variables de control.

Lo hacemos en tres partes.

- Primero, limpiamos la pantalla y pintamos la autopista.
- Segundo, pintamos todas las ranas.
- Tercero, mostramos la puntuación.

Este módulo incluye las siguientes rutinas:

<u>Rutina</u>	<u>Línea#</u>	<u>Desde(H)</u>	<u>Hasta(H)</u>	<u>Desde(D)</u>	<u>Hasta(D)</u>	<u>Cotejo</u>
INIT	1240	6F83H	700AH	28547	28682	11996
CLRSR	7060	72D7H	7316H	29399	29462	5236
DRWHWY	1820	700BH	7040H	28685	28736	4609
HIGHWY	1980	7038H	7040H	28728	28736	696
FILHWY	2070	7041H	7054H	28737	28756	2609
LINEUP	2290	7055H	7079H	28757	28793	4325
DISASC	7630	7328H	7349H	29480	29513	3580
SCRIMG	14500	776FH	7786H	30575	30598	1855
FINAL	15390	77FEH	781DH	30718	30747	2089

El módulo se extiende desde 28547 hasta 30747, 2201 octetos.

El nombre que se sugiere es "init", (inicialización).

Has de meter primero las rutinas CLRSR, DRWHWY, FILHWY, FINAL en sus correspondientes celdillas de memoria. Luego metes la rutina INIT. Mete tres octetos con ceros para las siguientes líneas que te mostramos, en lugar de las "llamadas" ya que las rutinas todavía no las hemos desarrollado.

<u>Línea#</u>	<u>dirección(H)</u>	<u>dirección(D)</u>
1430	6FAFH	28591
1470	6FBAH	28602
1490	6FC0H	28608
1530	6FCBH	28619
1570	6FD6H	28630
1590	6FDCH	28636
1630	6FE7H	28647

Luego mete los siguientes códigos en la memoria a partir de 32000.

Guarda el módulo de 28547 hasta 30598 (2052 octetos), antes de ejecutar el código a partir de la 32000.



F3	DI		; interrupciones vedadas
D9	EXX		; conservar HL'
E5	PUSH	HL	
D9	EXX		
CD836F	CALL	INIT	
3E7F	KEY	LD	A, 7FH ; atrapar tecla SPACE
DBFE		IN	A, (FEH)
E601	AND	1	
20F8	JR	NZ, KEY	; repite si no pulsada
CDFE77	CALL	FINAL	; finalización
D9	EXX		; repone HL'
E1	POP	HL	
D9	EXX		
FB	EI		; interrupciones permitidas
C9	RET		

Deberás observar que la pantalla se ennegrece y que aparecen 4 líneas blancas.

A continuación, describimos brevemente lo que cada rutina hace.

INIT

establece el negro como color del borde
coloca los valores iniciales para el banderín de choque de rana, existencia de la rana, el banderín de juego y el número de ranas.
estipula un puntero aleatorio hacia la ROM
coloca la estación de la rana (también posición inicial de la rana) en 50ACH
llama a la que limpia la pantalla
llama a la que pinta la autopista
llama a la que coloca en línea las ranas (5 de ellas)
carga el mensaje de puntuación
expone los puntos
carga el mensaje de campeón
expone los puntos del campeón
estipula que todos los objetos son no-existentes
coloca el banderín de caza, el de sonido de sirena y el de puntuación

DRWHWY

pinta el seto superior de la autopista (32 símbolos a partir de 40A0H)
pinta el seto medianal de la autopista (32 signos a partir de 4860H)
pinta el seto inferior de la autopista (32 signos a partir de 5020H)

* recuerda que la autopista se considera con tinta negra sobre papel en blanco
vacía los dos octetos superiores del seto superior
(por tanto, se quedan en blanco)
vacía los dos octetos inferiores del seto inferior
(ahora también están en blanco)
redibuja los dos octetos medianales del seto central de la autopista.

FILHWY

establece el símbolo de relleno (FFH)
prepara un contador de lazo con 32 (una línea 32 pictos)



dibuja en un picto (8 octetos)
mueve el puntero para que apunte al siguiente picto.

FINAL

coloca en blanco el borde de la pantalla
deja en blanco la pantalla
cambia los atributos de toda la pantalla a papel en blanco y tinta negra.

Si todo es correcto, guarda primero el módulo desde la celdilla 28500 hasta la 30800, (2300 octetos).

Ahora metes las rutinas LINEUP, DRWFRG. Verificas el cotejo y guardas todo el módulo otra vez con el mismo nombre y con la misma dirección.
Luego cambia las direcciones de memoria de 6FAFH (28591) a 6FB1H (28593) para que corresponda a la línea 1430 del listado ensamblado; ie. CD 55 70

Rula 32000 y verás cinco ranas alineándose en el fondo izquierdo de la pantalla

A continuación, describimos lo que estas dos rutinas hacen:

LINEUP

coloca la dirección de la rana a 1 (hacia la derecha)
forma de la rana a FROG2
número de atributo a 2 (verde)
si no quedan ranas: vuelve
si sí:

con las ranas que quedan

apila BC, DE, HL

dibuja la rana citando la rutina DRWFRG

quita de la pila HL, DE, BC

actualiza la posición donde dibujar

DRWFRG

Dibuja la figura usando los convenios comentados anteriormente
calcula el puntero de atributos
rellena los atributos de la rana

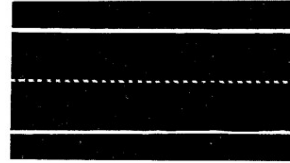


Ahora mete las rutinas DISASC y SCRIMG, comprobando el cotejo y guardando el módulo de nuevo como anteriormente. Luego cambia las direcciones mencionadas en las líneas con números

1470, 1490, 1530, 1570, 1590, 1630

a los valores correctos reflejados en el listado.

Rula 32000 y verás toda la pantalla aderezada con la autopista pintada y mostrando las ranas y la puntuación.



ETAPA 3 (tráfico regular)

En esta etapa desarrollamos el flujo regular de tráfico; ie. todo el tráfico, excepto el coche de policía:

Control de tráfico (incluyendo regeneración del tráfico)
regenerar tráfico
Moviendo el tráfico
control de movimiento
dibujo del tráfico
determinar las figuras a dibujar

A continuación, tienes la tabla de todas las rutinas de este módulo.

<u>Nombre</u>	<u>línea#</u>	<u>desde(H)</u>	<u>hasta(H)</u>	<u>desde(D)</u>	<u>hasta(D)</u>	<u>cotejo</u>
TFCTRL	3090	70BDH	70D8H	28861	28888	2587
REGEN	3320	70D9H	710EH	28889	28942	5673
MOVTRF	3700	710FH	71AEH	28943	29102	14831
MVCTRL	4720	71AFH	7208H	29103	29192	9222
DRAW	5560	7209H	7295H	29193	29333	13923
RSHAPE	6630	7296H	72D6H	29334	29398	6803
RANDNO	15050	77CCH	77DDH	30668	30685	2194

Módulo desde 28861 hasta 30685, 1824 octetos.

Nombre propuesto "regtrf" (tráfico regular).

- De nuevo, no tiene sentido generar el total del cotejo para todo el módulo, ya que la zona de memoria contiene algunas áreas todavía sin desarrollar. Pero es importante que cotejes cada rutina después de teclearla.

Desarrollamos este módulo en dos partes.

- ★ Primeramente, la rutina para dibujo del tráfico.
En segundo lugar, el control del tráfico y el control del dibujo.

Mete las rutinas DRAW, RSHAPE en su correspondiente región de memoria, verifica el cotejo y guárdalas.



Luego introduce el siguiente programa de comprobación comenzando a partir de la celdilla 32000.

F3	DI		
D9	EXX		
E5	PUSH	HL	
D9	EXX		
CD836F	CALL	INIT	
3E03	LD	A, 3	; contar filas
32606F	LD	(ROW), A	; guardar en FILA (row)
3E09	LD	A, 9	; contar columnas
325F6F	LD	(COLUMN), A	; guardar en column
11926C	LD	DE, RTRUCK	; figura de camión a derechas
216A6D	LD	HL, RTATT	; atributos de camión a derechas
226A6F	LD	(ATTPTR), HL	; guardar en ATTPTR
3E01	LD	A, 1	; colocar como real
212248	LD	HL, 4822H	; carril superior
CD0972	CALL	DRAW	; dibujar la figura
3E7F KEY	LD	A, 7FH	; atrapar la tecla
DBFE	IN	A, (OFEH)	
E601	AND	1	
20F8	JR	NZ, KEY	
CDFE77	CALL	FINAL	
D9	EXX		
E1	POP	HL	
D9	EXX		
FB	EI		
C9	RET		

Carga los módulos de base de datos en el orden en que son creados.
Carga el módulo de iniciación.

Carga las rutinas que hayas desarrollado hasta este momento.

Guarda 4000 octetos a partir de la celdilla 27000 en el módulo "rana". Eso incluye todas las rutinas que has desarrollado hasta ahora.

Inscribe y guarda la rutina anterior de prueba en la celdilla 32000.

Rulalo empezando en 32000 y deberás ver toda la pantalla preparada y un camión a derechas en el carril superior.

Puedes cambiar los parámetros del programa anterior entre las instrucciones CALL INIT y CALL DRAW para comprobar todas las otras figuras de objetos.

Ahora te describimos brevemente las dos rutinas.

DRAW

Con lógica similar a DRWFRG

RSHAPE

extrae los 5 bits inferiores del octeto inferior que define el parámetro de posición
resta de 1FH y añade 1
extrae de nuevo los 5 bits inferiores
determina si SKIP o FILL según sea real o abstracto
calcula la posición de atributos y lo guarda en ATTPOS

Teclea las rutinas TFCTRL, REGEN y MVCTRL en su región de memoria y guarda el módulo completo.

Revisa la rutina de prueba como sigue:

		DI	
		EXX	
		PUSH	HL
		EXX	
		CALL	INIT
CD8D70	MOVE	CALL	TFCTRL
CD0F71		CALL	MOVTRF
3E7F		LD	A, 7FH
DBFE		IN	A, (OFEH)
E601		AND	1
20F2		JR	NZ, MOVE
		CALL	FINAL
		EXX	
		POP	HL
		EXX	
		EI	
		RET	

- En este momento, te darás cuenta que guardamos el módulo completo de una etapa mientras estamos desarrollando esa etapa. Una vez que el módulo esté completamente comprobado, ha de ser fundido junto con los módulos anteriores y guardado en el módulo "rana".

Comprobamos los módulos mediante un pequeño programa de prueba que comienza a partir de la celdilla 32000.

Tras haber hecho todo el trabajo doméstico y de limpieza de tus módulos, haz una pasada de comprobación del nuevo módulo "rana". Si todo es correcto, deberás ver toda la pantalla como antes, más todo el tráfico moviéndose muy rápidamente en los dos carriles. Lo hace rápidamente porque no hay retardo en cada ciclo de programa.

TFCTRL

carga el banderín de generación
si no hay regeneración
decrementa la cuenta del banderín
vuelve

si sí lo hay
regenera el primer objeto no existente, usando la rutina REGEN
vuelve



REGEN

guarda el puntero de existencia de la base de datos
genera un número aleatorio de 0 a 5
comprueba los dos primeros caracteres de la posición de pantalla donde
ha de acercarse el objeto
si la suma de los atributos de estas dos posiciones no es igual a 0
entonces vuelve (tapón de tráfico).

si sí lo es
determina la base de datos inicial
carga en la base de datos temporal los datos de trabajo
coloca la cuenta de ciclo de regeneración a dos
vuelve

MOVTRF

para todos los objetos existentes
decrementa la cuenta del ciclo
si la cuenta llega a cero
recarga cuenta desde existencia
mueve un carácter a la izquierda o a la derecha
guarda la nueva posición en NEWPOS
comprueba la correspondencia de atributos del
frente de los objetos
si hay alguno con tinta no cero
si no hay verde
coloca banderín de tapón
si sí lo hay
coloca banderín de choque
si está alzado el banderín de tapón
carga la cuenta de ciclo con 2 (mover un ciclo posterior)
volver
si no está alzado
guardar nueva posición
citar MVCTRL (control de movimiento).

MVCTRL

si se alcanza el borde
cambiar el banderín real/abstracto
si se está moviendo hacia la izquierda
si está en el borde (octeto inferior de la posición = 1FH)
si banderín indica abstracto
lo coloca como no existente y vuelve
si no lo indica
se va a L1

y si no está en el borde
obtiene el final del objeto
si llega al final de la pantalla (octeto de orden inferior
es 0COH)
coloca no existente, vuelve

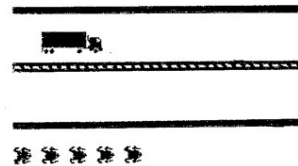


L1: repone la cuenta del ciclo
busca el puntero de la figura
guarda el puntero de atributos en ATTPTR
coge la fila y columna de la figura
dibuja (draw) con la nueva posición.

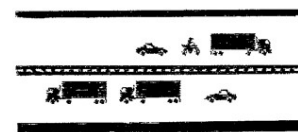
RANDNO

apila HL, BC
averigua dónde está apuntando el puntero aleatorio
actualiza puntero (mueve hacia abajo en la ROM)

Score 0 HIGH SCORE 0



Score 0 HIGH SCORE 0



ETAPA 4 (coche de policía)

En esta etapa, meteremos el coche de policía en el programa.

El coche de policía será generado aleatoriamente y entrará con la sirena sonando. Se mueve cada ciclo y no hay tapón de tráfico en su carrera. Adelantará a cualquier vehículo regular que encuentre por delante.

Por eso es por lo que el programa necesita conservar lo que está detrás del coche de policía y volverlo a mostrar cuando ha pasado el coche.

Las rutinas pertenecientes a este módulo son.

<u>Nombre</u>	<u>línea#</u>	<u>desde(H)</u>	<u>hasta(H)</u>	<u>desde(D)</u>	<u>hasta(D)</u>	<u>cotejo</u>
RESPC	9560	7450H	74C1H	29776	29889	11011
POLICE	7930	734AH	73DEH	29514	29662	15769
STRPC	8830	73DFH	744FH	29663	29775	10615

Hay rutinas que son citadas en este módulo y que han sido desarrolladas en módulos anteriores.

El módulo va desde 29514 hasta 29889, y son 376 octetos.

El nombre que se sugiere es "police".

Introduce las rutinas POLICE y STRPC. Verificalas y guárdalas en el módulo "police".

Luego revisa el programa de prueba para que ahora sea:

		DI	
		EXX	
		PUSH	HL
		EXX	
		CALL	INIT
	MOVE	CALL	TFCTRL
		CALL	MOVTRF
CD4A73	MOVEI	CALL	POLICE
		LD	A, 7FH
		IN	A, (OFEH)
		AND	I
20F5		JR	NZ, MOVEI
		CALL	FINAL
		EXX	
		POP	HL
		EXX	
		EI	
		RET	

Carga el módulo "rana", luego el módulo "police".



Pasa el programa de comprobación. Debieras ver el coche de policía moviéndose muy rápidamente por la autopista.

Si quieres añadir otros vehículos, cambia el salto relativo para que sea

JR NZ, MOVE

Recuerda que tienes que recalcular el desplazamiento del salto (EFH).

A medida que corre, verás que el coche de policía borra la figura del vehículo que adelanta. Puede que sea tan rápido que no lo observes. Pero si podrás decir, cuando algún vehículo comienza a acercarse al vehículo que va delante de él.

Observemos cuidadosamente estas rutinas:

POLICE

```
si el coche de policía no existe
  saca un número aleatorio
  si no es múltiplo de 3!
    vuelve
  si sí lo es
    coloca el banderín de caza
    determina aleatoriamente si es el carril superior o inferior
    carga la correspondiente base de datos inicial.
obtiene la dirección
guarda el puntero de posición
recupera la posición
mueve y guarda NEWPOS
establece fila, column, banderín real/abstracto y posición, antes de citar
RSHAPE
obtiene el ATTPOS resultante y comprueba si la cabecera de la figura
  corresponde a verde
si el atributo es verde
  alza el banderín de choque
  blanquea el frente del coche de policía
cita STRPC (para conservar lo que está detrás del coche de policía).
actualiza la base de datos de la posición
cita MVCTRL (para mover dentro y fuera de la pantalla)
desactiva el banderín de caza si es no-existente.
```

STRPC

```
hace que HL apunte a NEWPOS
hace que DE apunte a PCSTR (datos del coche de policía)
guarda la posición y 5 octetos de información a partir de la variable
ROW (fila)
deposita según el formato SKIP/FILL
  primero toda la región de memoria
  luego la región de atributos
```

Mete la rutina RESPC y fúndela con las dos rutinas anteriores. Guarda el módulo completo como "police".



CD8777

```
CALL RESPC
CALL MOVTRF
CALL POLICE
CALL SIREN
LD A, 7FH
IN A, (0FEH)
AND 1
JR NZ, MOVE
```

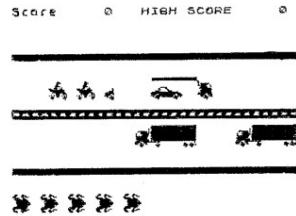
Rula el programa de prueba y hallarás que todo el flujo de tráfico se ha ralentizado. Es así, a causa de un retardo constante, bien por efecto de la salida de sonido, o bien por un lazo de retardo en la cuenta atrás.

SIREN

```
atrapa si pulsada tecla (ENTER)
si está pulsada
    cambia de sí-sirena a no-sirena o viceversa
si no hay sonido
    va a DELAY
si sí hay sonido
    si no hay coche policía
        va a DELAY
    si sí hay coche
        determina la base de datos correcta
        carga sobre DE, HL
        cita 03B5H
        vuelve
```

DELAY: cuenta 6144 regresivamente.

Funde este módulo con "rana" en la memoria y vuelve a cargarlo en "rana".



ETAPA 5 (rana)

En esta etapa desarrollamos las rutinas que conciernen a la rana.

Necesitamos regenerar la rana cuando muere cualquiera de ellas.

- Necesitamos manejar el movimiento de la rana, guardar lo que está detrás de ella y volverlo a presentar cuando la rana pase.

Necesitamos también tratar el choque de la rana o la llegada a casa, y calcular la puntuación.

Construimos este módulo en tres partes, como sigue:

- ▶ Regenerar y mover la rana
- ▶ Depositar y recuperar lo que está detrás de ella
- ▶ Tratar el choque, la llegada y la puntuación.

Todas las rutinas pertenecientes a este módulo son:

<u>Nombre</u>	<u>línea#</u>	<u>desde(H)</u>	<u>hasta(H)</u>	<u>desde(D)</u>	<u>hasta(D)</u>	<u>octeto</u>
FROG	10280	74C2H	74E2H	29890	29922	3818
REGFRG	10520	74E3H	750FH	29923	29967	4079
MOVFRG	10770	7510H	75D5H	29968	30165	19943
RESFRG	11870	75D6H	7627H	30166	30247	8492
STRFRG	12440	7628H	7690H	30248	30352	10136
CRASH	13160	7691H	76A6H	30353	30374	2767
FRGDIE	13280	76A7H	7707H	30375	30471	9965
FRGTON	13890	7708H	771CH	30472	30492	2435
CALSCR	14040	771DH	776EH	30493	30574	8106

El módulo ocupa desde 29890 hasta 30574, y son 685 octetos.

El nombre sugerido es "frgrtn" (frog rutina).

Mete las rutinas FROG, REGFRG, MOVFRG, RESFRG, STRFRG y CRASH.

Revisa el programa de prueba para que sea como sigue:

```

      .
      .
      .
MOVE  CALL    INIT
      CALL    TFCTRL
      CALL    RESPC
      CALL    MOVTRF
      CALL    POLICE
      CALL    FROG
      CALL    STREN
  
```



```
LD    A, 7FH
IN    A, (0FEH)
AND   1
JR    NZ, MOVE
.
```

Dado que no has introducido todavía la rutina FRGDIE, sustituye la codificación de la línea 13190 del listado ensamblado, por los valores:

```
00, 00, 00
```

Pasa el programa de prueba y verás que puedes mover la rana. Los controles son "i"=arriba, "a"=abajo, "i"=izquierda, "p"=derecha.

Cuando la rana choca, simplemente desaparece; porque la rutina FRGDIE que trata la muerte de la rana todavía no ha sido inscrita en memoria.

La descripción de estas rutinas es la siguiente:

FROG

```
Si rana choca
  va CRH
si no choca
  fija banderín-puntuación a sin-puntuación (0)
  cita REGFRG
  decrementa cuenta de ciclo
  si cuenta no es cero
    vuelve
  si sí es cero
    arredra cuenta de ciclo
    cita MOVFRG
    si no hay choque
      vuelve
```

```
CRH: cita rutina choque (CRASH)
      vuelve
```

REGFRG

```
si rana es no-existente
  carga base de datos iniciales de rana en región de trabajo
  cambia estación de rana a tres posiciones a la izquierda
  inicializa OLDFRG y NEWFRG con FRGPOS
  inicializa con ceros el área de almacenamiento de la rana
  vuelve
```

MOVFRG

```
inicializa registros
  C - movimiento absoluto
  B - dirección de la rana
  DE- figura de rana
```



```

comprueba movimiento de la rana
    l - arriba, a - abajo, i - izquierda, p - derecha
conserva figura, dirección
si movimiento absoluto es cero
    vuelve
si no es cero
    repone posición vieja de rana
    calcula posición nueva y guardala
verifica posición en pantalla arriba, derecha, fondo, estación de la rana
si es válida
    guarda posición en NEWFROG
    fija banderín de puntuación
recupera OLDFRG y
si OLDFRG es igual a NEWFRG
    vuelve
si no es igual
    cita RESFRG
    hace OLDFRG igual a NEWFRG
    cambia dirección conservada, puntero de figura a base de datos
    de rana
    cita STRFRG
    vuelve

```

RESFRG

repone lo situado detrás de la rana basándose en la posición de OLDFRG, primero pantalla y luego atributos.

STRFRG

conserva lo situado detrás de la rana basándose en la posición de NEWFRG dibuja rana además de tráfico.

CRASH

```

baja banderín de choque
fija rana a no-existente
cita FRGDIE (tratamiento de la muerte)
cita RESFRG (restablecer lo tapado por la rana)
decrementa número de ranas

```

Después de guardar todos estos módulos, mete las rutinas CALSCR, FRGDIE y FRGTON.

Para comprobar la rutina que trata el choque, sustituye la línea 13190 del listado por la siguiente instrucción:

7698	CDA776	CALL	FRGDIE	(30360)
------	--------	------	--------	---------

Revisa el programa de prueba para que sea:

```
CALL    FROG
CALL    CALSCR
CALL    SIREN
```

Funde la rutina en el módulo "frgrtn" y ejecuta todo a partir de la celdilla 32000 junto con el módulo "rana".

Cuando la rana choca, parpadeará en rojo y se desvanecerá.

FRGDIE

```
comprueba si rana llega a casa o muere
pone sonido mortuario, atributo color rojo
si llega a casa
    suma uno al tercer dígito de puntuación
    (bono de 100 puntos)
    cita DISSCR (muestre puntuación)
    pone sonido hogareño, atributo color amarillo
dibuja rana basándose en OLDFRG, FROGSH, y atributos recientes,
citando DRWFRG
parpadea rana con el atributo cinco veces
```

FRGTON

```
cita TONE1 (código de tono de la rutina SIREN)
sube y baja la base de datos de tono dependiendo del atributo
    usado para parpadeo de rana (si amarillo baja base de datos)
    (si rojo sube base de datos)
```

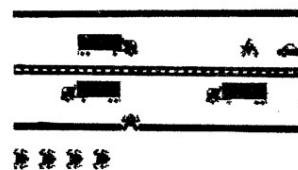
CALSCR

```
si rana no existe
    vuelve
si sí existe
    si sube
        suma uno al 10º dígito de puntuación
        (10 puntos)
    si no sube
        si no está dentro de autopista
            vuelve
        si sí está dentro
            suma uno al 10º dígito
```

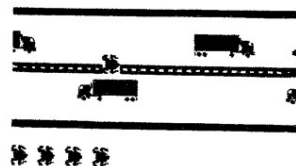


reajusta todos los dígitos de puntuación
prepara la imagen de puntuación
muestra puntuación

Score 0 HIGH SCORE 0



Score 100 HIGH SCORE 0



ETAPA 6 (control)

En esta etapa, desarrollamos la rutina que controla el programa completo.

El control principal es que cuando finaliza el juego, se actualiza la puntuación del campeón y se reanuda automáticamente el juego.

- En cualquier momento, el usuario puede acabar el juego pulsando la barra espaciadora.

Te darás cuenta que las líneas 180 a 440 del listado se parecen bastante al programa de prueba que hemos estado usando.

Las rutinas que quedan por programar son las siguientes:

<u>nombre</u>	<u>línea#</u>	<u>desde(H)</u>	<u>hasta(H)</u>	<u>desde(D)</u>	<u>hasta(D)</u>	<u>cotejo</u>
START	180	6978H	69AEH	27000	27054	8427
OVER	15200	77DEH	77FDH	30686	30717	2491

Ahora inscribe esas rutinas en la memoria. Guárdalas junto con el módulo "rana" y guarda el módulo completo como "rana".

Run 27000 en lugar de 32000 y tendrás todo el programa funcionando.

OVER

```

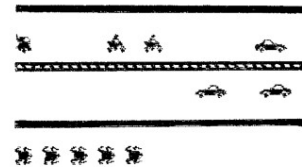
compara todos los dígitos de HISCR y SCORE + 1
con el primer dígito distinto
si dígito HISCR es inferior
  cambia HISCR por SCORE + 1
si no es inferior
  vuelve

```

vuelve

Enhorabuena, y espero que hayas disfrutado con el desarrollo del programa LA RANA SUICIDA.

Score 1140 HIGH SCORE 1140



```

00100 :***** FREEMAN FROG *****
00110 :
00120 :
00130 :
00140 :
00150 :
697A 00160 ORG 27000
00170 :
697B F3 00180 START DI ;DISABLE BASIC SYSTEM AFFECTING
697C D9 00190 EXX ;THE KEYBOARD SCANNING
697D E5 00200 PUSH HL ;PRESERVE THE HL REGISTER PAIR
697E D9 00210 EXX ;POP BACK BEFORE RETURN
697F CD836F 00220 AGAIN CALL INIT ;INITIALISATION
6980 CD8070 00230 MOVE CALL TFCTRL ;TRAFFIC CONTROL ROUTINE
6981 CD5074 00240 CALL RESPC ;RESTORE UNDERNEATH
6982 CD0F71 00250 CALL MOVTRF ;MOVE TRAFFIC
6983 CD4A73 00260 CALL POLICE ;POLICE CAR ROUTINE
6984 CD0274 00270 CALL FROG ;FROG MODULE
6985 CD1D77 00280 CALL CALSCR ;CALCULATE AND DISPLAY SCORE
6986 CD8777 00290 CALL SIREN ;SIREN OR DELAY
6987 3A776F 00300 LD A,(RAMFL6) ;FINISH WHEN NO FROG
6988 A7 00310 AND A
6989 2005 00320 JR NZ,CONTIN
698A CDDE77 00330 CALL OVER ;HIGHSCORE MANAGEMENT
698B 18DD 00340 JR AGAIN ;NEW GAME AGAIN
698C 3E7F 00350 CONTIN LD A,7FH ;TRAP SPACE KEY PRESSED
698D DBFE 00360 IN A,(0FEH) ;SCAN KEYBOARD
698E E601 00370 AND 1
698F 20D9 00380 JR NZ,MOVE
6990 CDFE77 00390 CALL FINAL ;RESET SCREEN AND BORDER COLOUR
6991 D9 00400 EXX
6992 E1 00410 POP HL ;RETRIEVE HL
6993 D9 00420 EXX
6994 FB 00430 EI ;ENABLE INTERRUPTS
6995 C9 00440 RET ;RETURN TO BASIC SYSTEM
00450 :
00460 :
00470 :
69AF 00100 :***** FROGDB/ASM *****
00110 :
69AF D769 00120 FRGSHIP DEFW FROG1 ;UP FROG
69B1 D769 00130 DEFW FROG2 ;RIGHT FROG
69B3 F769 00140 DEFW FROG3 ;DOWN FROG
69B5 176A 00150 DEFW FROG4 ;LEFT FROG
69B7 6F 00160 FROG1 DB 111,15,31,159,220,216,120,48
OF 1F 9F DC DB 7B 30
69BF F6 00170 DB 246,240,248,249,59,27,30,12
FO FB F9 3B 1B 1E 0C
69C7 00 00180 DB 0,1,35,37,111,79,223,255
01 23 25 6F 4F DF FF
69CF 00 00190 DB 0,128,196,164,246,242,251,255
80 C4 A4 F6 F2 FB FF
69D7 1F 00200 FROG2 DB 31,31,31,127,252,193,113,56
1F 1F 7F FC C1 71 3B
69DF FE 00210 DB 254,244,248,240,192,156,240,192
F4 FB FO C0 9C FO C0
69E7 38 00220 DB 56,113,193,252,127,31,31,31

```



71 C1 FC 7F 1F 1F 1F		
69EF C0 00230	DB	192, 240, 156, 192, 240, 248, 244, 254
F0 9C C0 F0 FB F4 FE		
69F7 FF 00240 FROG3	DB	255, 223, 79, 111, 37, 35, 1, 0
DF 4F 6F 25 23 01 00		
69FF FF 00250	DB	255, 251, 242, 246, 164, 196, 128, 0
FB F2 F6 AA C4 B0 00		
6A07 30 00260	DB	48, 120, 216, 220, 159, 31, 15, 111
7B DB DC 9F 1F 0F 6F		
6A0F 0C 00270	DB	12, 30, 27, 59, 249, 248, 240, 246
1E 1B 3B F9 FB F0 F6		
6A17 7F 00280 FROG4	DB	127, 47, 31, 15, 3, 57, 15, 3
2F 1F 0F 03 39 0F 03		
6A1F F0 00290	DB	240, 240, 248, 254, 63, 131, 142, 28
F0 FB FE 3F B3 BE 1C		
6A27 03 00300	DB	3, 15, 57, 3, 15, 31, 47, 127
0F 39 03 0F 1F 2F 7F		
6A2F 1C 00310	DB	28, 142, 131, 63, 254, 248, 240, 240
BE B3 3F FE FB F0 F0		
00320 ;		
00330 ;		
6A37 00 00340 LBIKE	DB	0, 0, 0, 0, 0, 0, 0, 0
00 00 00 00 00 00 00		
6A3F 1F 00350	DB	31, 63, 115, 81, 169, 112, 112, 32
3F 73 51 A9 70 70 20		
6A47 FE 00360	DB	254, 252, 252, 234, 213, 206, 14, 4
FC FC EA D5 CE 0E 04		
6A4F 00 00370	DB	0, 0, 0, 0, 0, 0, 0, 0
00 00 00 00 00 00 00		
6A57 00 00380	DB	0, 0, 0, 0, 0, 0, 0, 0
00 00 00 00 00 00 00		
6A5F 01 00390	DB	1, 3, 1, 0, 3, 4, 14, 31
03 01 00 03 04 0E 1F		
6A67 B0 00400	DB	128, 192, 192, 224, 224, 112, 119, 255
C0 C0 E0 E0 70 77 FF		
6A6F 00 00410	DB	0, 0, 0, 0, 0, 0, 0, 0
00 00 00 00 00 00 00		
00420 ;		
6A77 00 00430 LBATT	DB	0, 7, 7, 0
07 07 00		
6A7B 00 00440	DB	0, 7, 7, 0
07 07 00		
00450 ;		
6A7F 00 00460 RBIKE	DB	0, 0, 0, 0, 0, 0, 0, 0
00 00 00 00 00 00 00		
6A87 7F 00470	DB	127, 63, 63, 87, 171, 115, 112, 32
3F 3F 57 AB 73 70 20		
6A8F FB 00480	DB	248, 252, 206, 138, 149, 14, 14, 4
FC CE BA 95 0E 0E 04		
6A97 00 00490	DB	0, 0, 0, 0, 0, 0, 0, 0
00 00 00 00 00 00 00		
6A9F 00 00500	DB	0, 0, 0, 0, 0, 0, 0, 0
00 00 00 00 00 00 00		
6AA7 01 00510	DB	1, 3, 3, 7, 7, 14, 238, 255
03 03 07 07 0E EE FF		
6AAF B0 00520	DB	128, 192, 128, 0, 192, 32, 112, 248





6B6B	00	000040	DB	0,0,0,0,0,0,1,3
	00	00 00 00 00 01 03		
6B73	00	00850	DB	0,0,0,0,0,252,134,131
	00	00 00 00 FC 86 83		
6B78	00	00860	DB	0,0,0,0,0,0,0,0
	00	00 00 00 00 00 00		
6B83	00	00870	DB	0,0,0,0,0,0,0,0
	00	00 00 00 00 00 00		
6B88	00	00880	DB	0,0,0,0,0,0,0,0
	00	00 00 00 00 00 00		
	00	0090		
6B93	00	00900 RCATT	DB	0,2,2,2,2,0
	02	02 02 02 00		
6B99	00	00910	DB	0,2,2,0,0,0
	02	02 00 00 00		
		00920		
		00930		
6B9F	00	00940 LTRUCK	DB	0,0,0,0,0,0,0,0
	00	00 00 00 00 00 00		
6BA7	1F	00950	DB	31,31,31,62,61,59,3,1
	1F	1F 3E 3D 3B 03 01		
6BAF	FB	00960	DB	248,252,254,127,184,216,192,128
	FC	FE 7F B8 DB C0 80		
6BB7	FF	00970	DB	255,255,255,255,6,15,15,6
	FF	FF FF 06 0F 06		
6BBF	FF	00980	DB	255,255,255,0,0,0,0,0
	FF	FF 00 00 00 00 00		
6BC7	FF	00990	DB	255,255,255,0,0,0,0,0
	FF	FF 00 00 00 00 00		
6BCF	FF	01000	DB	255,255,255,0,6,15,15,6
	FF	FF 00 06 0F 06		
6BD7	FE	01010	DB	254,254,254,4,50,122,122,48
	FE	FE 04 32 7A 7A 30		
6BDF	00	01020	DB	0,0,0,0,0,0,0,0
	00	00 00 00 00 00 00		
6BE7	00	01030	DB	0,0,0,0,0,0,0,0
	00	00 00 00 00 00 00		
6BEF	00	01040	DB	0,0,7,9,17,17,31,31
	00	07 09 11 11 1F 1F		
6BF7	02	01050	DB	2,2,250,250,254,252,252,248
	02	FA FA FE FC FC FB		
6BFF	FF	01060	DB	255,255,255,255,255,255,255
	FF	FF FF FF FF FF FF		
6C07	FF	01070	DB	255,255,255,255,255,255,255
	FF	FF FF FF FF FF FF		
6C0F	FF	01080	DB	255,255,255,255,255,255,255
	FF	FF FF FF FF FF FF		
6C17	FF	01090	DB	255,255,255,255,255,255,255
	FF	FF FF FF FF FF FF		
6C1F	FE	01100	DB	254,254,254,254,254,254,254
	FE	FE FE FE FE FE FE		
6C27	00	01110	DB	0,0,0,0,0,0,0,0
	00	00 00 00 00 00 00		
6C2F	00	01120	DB	0,0,0,0,0,0,0,0
	00	00 00 00 00 00 00		
6C37	00	01130	DB	0,0,0,0,0,0,0,0



6C3F	00 00 00 00 00 00	DB	0,0,0,0,0,0,0,0
00	01140		
6C47	00 00 00 00 00 00	DB	0,0,0,0,0,255,255,255
00	01150		
6C4F	00 00 00 00 FF FF FF	DB	0,0,0,0,0,255,255,255
00	01160		
6C57	00 00 00 00 FF FF FF	DB	0,0,0,0,0,255,255,255
00	01170		
6C5F	00 00 00 00 FF FF FF	DB	0,0,0,0,0,255,255,255
00	01180		
6C67	00 00 00 00 FF FF FF	DB	0,0,0,0,0,254,254,254
00	01190		
6C6F	00 00 00 00 FE FE FE	DB	0,0,0,0,0,0,0,0
00	01200		
00 00 00 00 00 00	01210		
00 00 00 00 00 00	01220		
6C77	00 03 05 05 05 05	DB	0,3,3,5,5,5,5,5,0
00	01230 LIATT		
03 03 05 05 05 05	01240	DB	0,3,3,5,5,5,5,5,0
00	01250		
6C89	00 00 05 05 05 05	DB	0,0,0,5,5,5,5,5,0
00	01260		
00 00 00 00 00 00	01270		
6C92	00 00 00 00 00 00	DB	0,0,0,0,0,0,0,0
00	01280 RTRUCK		
6C9A	7F 7F 20 4C 5E 5E 0C	DB	127,127,127,32,76,94,94,12
00	01290		
6CA2	FF FF 00 60 F0 F0 60	DB	255,255,255,0,96,240,240,96
00	01300		
6CAA	FF FF 00 00 00 00 00	DB	255,255,255,0,0,0,0,0
00	01310		
6CB2	FF FF 00 00 00 00 00	DB	255,255,255,0,0,0,0,0
00	01320		
6CBA	FF FF 00 00 00 00 00	DB	255,255,255,255,96,240,240,96
00	01330		
6CC2	3F 7F FE 1D 1B 03 01	DB	31,63,127,254,29,27,3,1
00	01340		
6CCA	FB FB 7C BC DC C0 80	DB	248,248,248,124,188,220,192,128
00	01350		
6CD2	00 00 00 00 00 00 00	DB	0,0,0,0,0,0,0,0
00	01360		
6CDA	00 00 00 00 00 00 00	DB	0,0,0,0,0,0,0,0
00	01370		
6CE2	7F 7F 7F 7F 7F 7F 7F	DB	127,127,127,127,127,127,127
00	01380		
6CEA	FF FF FF FF FF FF FF	DB	255,255,255,255,255,255,255
00	01390		
6CF2	FF FF FF FF FF FF FF	DB	255,255,255,255,255,255,255
00	01400		
6CFA	FF FF FF FF FF FF FF	DB	255,255,255,255,255,255,255
00	01410		
6D02	FF FF FF FF FF FF FF	DB	255,255,255,255,255,255,255
00	01420		
6D0A	40 01430	DB	64,64,95,95,127,63,63,31



993

6E39	0000	01830	DEFW	0	
6E3B	00	01840	DB	0,0	
00					
6E3D	00	01850	DB3EXT	DB	0,0,0,0
00	00 00				
6E41	0000	01860	DEFW	0	
6E43	0000	01870	DEFW	0	
6E45	0000	01880	DEFW	0	
6E47	00	01890	DB	0,0	
00					
6E49	00	01900	DB4EXT	DB	0,0,0,0
00	00 00				
6E4D	0000	01910	DEFW	0	
6E4F	0000	01920	DEFW	0	
6E51	0000	01930	DEFW	0	
6E53	00	01940	DB	0,0	
00					
6E55	00	01950	DB5EXT	DB	0,0,0,0
00	00 00				
6E59	0000	01960	DEFW	0	
6E5B	0000	01970	DEFW	0	
6E5D	0000	01980	DEFW	0	
6E5F	00	01990	DB	0,0	
00					
6E61	00	02000	DB6EXT	DB	0,0,0,0
00	00 00				
6E65	0000	02010	DEFW	0	
6E67	0000	02020	DEFW	0	
6E69	0000	02030	DEFW	0	
6E6B	00	02040	DB	0,0	
00					
		02050	;		
		02060	;		
6E6D	00	02070	PCAREXT	DEFB	0
6E6E	00	02080	PCARCYC	DEFB	0
6E6F	00	02090	PCARDIR	DEFB	0
6E70	00	02100	PCARRAP	DEFB	0
6E71	0000	02110	PCARPOS	DEFW	0
6E73	0000	02120	PCARSHP	DEFW	0
6E75	0000	02130	PCARATT	DEFW	0
6E77	02	02140	PCARRAW	DEFB	2
6E7B	06	02150	PCARCOL	DEFB	6
		02160	;		
		02170	;		
6E79	00	02180	FRGEXT	DEFB	0
6E7A	00	02190	FRGCYC	DEFB	0
6E7B	00	02200	FRGDIR	DEFB	0
6E7C	0000	02210	FRGPOS	DEFW	0
6E7E	0000	02220	FRGSH	DEFW	0
6E80	00	02230	FRGATR	DEFB	0
		02240	;		
		02250	;		
6E81	0B	02260	FRGDB	DB	8,8,1
0B 01					
6E84	AC50	02270	FRGSTN	DEFW	50ACH
6E86	B769	02280	DEFW	FRGB1	

;POLICE CAR DATABASE
 ;FROG DATABASE
 ;0:UP 1:RHT 2:DOWN 3:LFT
 ;INITIAL POSITION OF FROG



6EB8 04	02290	DB	4	;ATTR. TOTAL 8 CHARS
	02300 ;			
6EB9 956E	02310 ;			
6EB8 A16E	02320 DBINDEX	DEFW	RBDB	;RIGHT BYTE DB
6EBD AD6E	02330	DEFW	LBDB	;LEFT BIKE DB
6EBF B96E	02340	DEFW	RCDB	;RIGHT CAR DB
6E91 C56C	02350	DEFW	LCDB	;LEFT CAR DB
6E93 D16E	02360	DEFW	RTDB	;RIGHT TRUCK DB
	02370	DEFW	LTDB	;LEFT TRUCK DB
	02380 ;			
	02390 ;			
6E95 02	02400 RBDB	DB	2,1,0,0	;EXT CNT DIR RAF
01 00 00				
6E99 1D4B	02410	DEFW	4B1DH	;POS
6E9B 7F6A	02420	DEFW	RB1KE	;RIGHT BIKE
6E9D BF6A	02430	DEFW	RBATT	;ATTRIBUTE
6E9F 02	02440	DB	2,4	;ROW COL
04				
	02450 ;			
	02460 ;			
6EA1 02	02470 LBDB	DB	2,1,1,1	
01 01 01				
6EA5 DF4B	02480	DEFW	4BDFH	
6EA7 376A	02490	DEFW	LB1KE	
6EA9 776A	02500	DEFW	LBATT	
6EAB 02	02510	DB	2,4	
04				
	02520 ;			
	02530 ;			
6EAD 03	02540 RCDB	DB	3,1,0,0	
01 00 00				
6EB1 1B4B	02550	DEFW	4B1BH	
6EB3 336B	02560	DEFW	RCAR	
6EB5 936B	02570	DEFW	RCATT	
6EB7 02	02580	DB	2,6	
06				
	02590 ;			
	02600 ;			
6EB9 03	02610 LCDB	DB	3,1,1,1	
01 01 01				
6EBD DF4B	02620	DEFW	4BDFH	
6EBF C76A	02630	DEFW	LCAR	
6EC1 276B	02640	DEFW	LCATT	
6EC3 02	02650	DB	2,6	
06				
	02660 ;			
	02670 ;			
6EC5 06	02680 RTDB	DB	6,1,0,0	
01 00 00				
6EC9 1B4B	02690	DEFW	4B1BH	
6ECB 926C	02700	DEFW	RTRUCK	
6ECD 6A6D	02710	DEFW	RTATT	
6ECF 03	02720	DB	3,9	
09				
	02730 ;			
	02740 ;			



6ED1	06	02750	LTDB	DB	6,1,1,1,1	
	01 01 01					
6ED5	DF48	02760		DEFW	48DFH	
6ED7	9F6B	02770		DEFW	LTRUCK	
6ED9	776C	02780		DEFW	LTATT	
6EDB	03	02790		DB	5,9	
	09					
		02800				
		02810				
6EDD	01	02820	LPCDB	DB	1,1,1,1,1	
	01 01 01					
6EE1	DF48	02830		DEFW	48DFH	
6EE3	C76A	02840		DEFW	LCAR	
6EE5	E96E	02850		DEFW	LPCATT	
6EE7	02	02860		DB	2,6	
	06					
		02870				
		02880				
6EE9	00	02890	LPCATT	DB	0,5,5,5,5,0	
	05 05 05	05 00				
6EEF	00	02900		DB	0,0,0,5,5,0	
	00 00 05	05 00				
		02910				
		02920				
6EF5	01	02930	RPCDB	DB	1,1,0,0	
	01 00 00					
6EF9	1B4B	02940		DEFW	481BH	
6EFB	336B	02950		DEFW	RCAR	
6EFD	016F	02960		DEFW	RPCATT	
6EFF	02	02970		DB	2,6	
	06					
		02980				
		02990				
6F01	00	03000	RPCATT	DB	0,5,5,5,5,0	
	05 05 05	05 00				
6F07	00	03010		DB	0,5,5,0,0,0	
	05 05 00	00 00				
		03020				
		03030				
		00480				
		00490				
6F0D	29	00500	PCTON1	DB	41,0,0F0H,1	:FIRST POLICE CAR TONE
	00 F0 01					
6F11	17	00510	PCTON2	DB	23,0,8CH,3	:SECOND POLICE CAR TONE
	00 BC 03					
		00520				
		00530				
6F15	46	00540	HOMTON	DB	46H,0,0C7H,4	:FROG REACH HOME TONE
	00 C7 04					
6F19	5D	00550		DB	5DH,0,8CH,3	
	00 BC 03					
6F1D	7C	00560		DB	7CH,0,0A1H,2	
	00 A1 02					
6F21	BA	00570		DB	0AAH,0,0F1H,1	
	00 F1 01					
6F25	DE	00580		DB	0DEH,0,6DH,1	



```

00 6D 01
6F29 2E 00590 DB 2BH,1,9,1
01 09 01 00600 DB 8BH,1,0BFH,0
01 BF 00 00610 DB 0FH,2,8BH,0
6F31 0F 02 9B 00 00620 DB 0C0H,2,5EH,0
6F35 C0 02 5E 00 00630 DIETON DB 84H,3,43H,0 ;FROG DYING TONE, REVERSE
6F39 84 03 43 00 00640 ;
00650 ;
6F3D 53 00660 SCRMS1 DM 'Score '
63 6F 72 65 20
6F43 30 00670 SCORE DB 30H,30H,30H,30H,30H,30H
30 30 30 30 30
6F49 4B 00680 SCRMS2 DM 'HIGH SCORE '
49 47 4B 20 53 43 4F 52
45 20
6F54 30 00690 HISCR DB 30H,30H,30H,30H,30H
30 30 30 30
00700 ;
00710 ;
0005 00 00720 IMAGE DS 5 ;PRINTING IMAGE OF SCORE
6F5E 00 00730 UPDOWN DEFB 0 ;SET WHEN FROG MOVES UP OR DOWN
00740 ;
00750 ;
6F5F 00 00760 COLUMN DB 0 ;VARIABLE STORING SHAPE COLUMN
6F60 00 00770 ROW DB 0 ;VARIABLE STORING SHAPE ROW
6F61 00 00780 SKIP DEFB 0 ;CHAR SKIPPING DURING DRAW
6F62 00 00790 FILL DEFB 0 ;CHAR DRAWN
6F63 0000 00800 ATTPOS DEFW 0 ;HOLDING THE ATTRIBUTE FILE PTR
6F65 00 00810 ATTR DB 0 ;ATTR OF CHARACTER BLOCK DRAWN
6F66 0000 00820 DRMPDS DEFW 0 ;DRAW POSITION
6F68 0000 00830 STRPOS DEFW 0 ;STORE POSITION
00840 ;
00850 ;
6F6A 0000 00860 ATTPTR DEFW 0
6F6C 0000 00870 NEWPOS DEFW 0 ;NEW TRAFFIC OBJECT POSITION
6F6E 0000 00880 POSPTR DEFW 0 ;TRAFFIC POSITION DATABASE PTR
6F70 00 00890 GENFLG DEFB 0 ;TRAFFIC REGENERATION FLAG
00900 ;
00910 ;
6F71 00 00920 JAMFLG DEFB 0 ;SET TO 1 AS TRAFFIC MOVE JAM
00930 ;
00940 ;
6F72 00 00950 CHASE DEFB 0 ;SET WHEN POLICE CAR APPEARS
6F73 00 00960 SOUNDIF DEFB 0 ;SET WHEN USER WANT SIREN SOUND
6F74 00 00970 TONFLG DEFB 0 ;DETERMINE WHICH SIREN TONE
6F75 0000 00980 RND DEFW 0 ;POINTER TO ROM FOR RANDOM NO
00990 ;
01000 ;
6F77 01 01010 GAMFLG DEFB 1 ;END IF ZERO
6F7B 0000 01020 OLDFRG DEFW 0 ;OLD FROG POS
6F7A 0000 01030 NEWFRG DEFW 0 ;NEW FROG POS

```



```

6F7C 00      01040 CRHFLG  DEFB  0      ;SET TO 1 WHEN FROG WAS CRASH
6F7D 00      01050 TEMDIR  DEFB  0      ;FROG TEMPORARY NEW DIRECTION
6F7E 0000    01060 TEMPOS  DEFW  0      ;FROG TEMPORARY NEW POSITION
6F80 0000    01070 TEMSHF  DEFW  0      ;FROG TEMPORARY NEW SHAPE
              01080 ;
              01090 ;
5020          01100 BOTHY1  EDU  5020H  ;0,38, 0,39
5120          01110 BOTHY2  EDU  5120H
46A0          01120 TOPHY1  EDU  46A0H  ;0,128, 0,129
47A0          01130 TOPHY2  EDU  47A0H
4B60          01140 MIDHY1  EDU  4B60H  ;x,83, x,84
4C60          01150 MIDHY2  EDU  4C60H
              01160 ;
              01170 ;
3C00          01180 CHRSET  EDU  3C00H  ;FIRST 256 BYTES NOTHING
              01190 ;
              01200 ;
6F82 05      01210 NUMFRG  DEFB  5      ;NUMBER OF FROG
              01220 ;
              01230 ;
6F83 AF      01240 INIT    XOR  A      ;000 FOR D2 D1 D0
6F84 D3FE    01250        OUT  (OFEH),A ;SET BORDER COLOUR
6F86 32485C  01260        LD   (23624),A ;TO BLACK
6F89 327C6F  01270        LD   (CRHFLG),A
6F8C 32796E  01280        LD   (FRGEXT),A ;SET FROG NON EXIST
6F8F 3C      01290        INC  A      ;SET GAME FLAG
6F90 32776F  01300        LD   (GAMFLG),A
6F93 3E05    01310        LD   A,5     ;INITIALISE FROG NO
6F95 32826F  01320        LD   (NUMFRG),A
6F98 ED5F    01330        LD   A,R     ;GENERATE RANDOM PTR
6F9A E63F    01340        AND  3FH    ;FOR THIS CYCLE
6F9C 67      01350        LD   H,A     ;PTR POINTS TO ROM
6F9D ED5F    01360        LD   A,R
6F9F 6F      01370        LD   L,A
6FA0 22756F  01380        LD   (RND),HL
6FA3 21A050  01390        LD   HL,500EH
6FA6 22846E  01400        LD   (FRGSTN),HL ;INIT FROG STATION
6FA9 C0D772  01410        CALL CLRSCR ;CLEAR SCREEN ROUTINE
6FAC C0D870  01420        CALL DRMHY  ;DRAW HIGHWAY
6FAF C05570  01430        CALL LINEUP ;LINE UP ALL EXIST FROGS
6FB2 210040  01440        LD   HL,4000H ;MESSAGE LOCATION
6FB5 113D6F  01450        LD   DE,SCRM5 ;LOAD SCORE MESSAGE
6FB8 0606    01460        LD   B,6
6FBA C02873  01470        CALL DISASC ;DISPLAY ASCII CHARACTER
6FBD 21446F  01480        LD   HL,SCORE+1 ;PRINT SCORE
6FC0 C06F77  01490        CALL SCRIMG ;CONVERT TO PRINTABLE IMAGE
MS
6FC3 210640  01500        LD   HL,4006H
6FC6 11596F  01510        LD   DE,IMAGE
6FC9 0405    01520        LD   B,5
6FCB C02873  01530        CALL DISASC
6FCE 210E40  01540        LD   HL,400EH  ;HIGH SCORE MESSAGE
6FD1 11496F  01550        LD   DE,SCRM5
6FD4 0608    01560        LD   B,11
6FD6 C02873  01570        CALL DISASC
6FD9 21546F  01580        LD   HL,HISCR
6FDC C06F77  01590        CALL SCRIMG

```



```

6FDF 211940 01600 LD HL,4019H
6FE2 11596F 01610 LD DE,IMAGE
6FE5 0605 01620 LD B,5
6FE7 CD2873 01630 CALL DISABC
6FEA 21259E 01640 LD HL,OBJEXT ;SET ALL OBJ NONEXIST
6FED 110C00 01650 LD DE,12
6FF0 0607 01660 LD R,7
6FF2 6F 01670 XOR A
6FF3 77 01680 LD (HL),A
6FF4 19 01690 ADD HL,DE
6FF5 10FC 01700 DJNZ INTLP1
6FF7 32726F 01710 LD (CHASE),A ;SET NO POLICE CAR CHASE
6FFA 3C 01720 INC A
6FFB 32736F 01730 LD (SOUND),A ;SET SIREN ON
6FFE 21436F 01740 LD HL,SCORE ;INITIALISE SCORE TO
7001 11446F 01750 LD DE,SCORE+1 ;ASCII ZERO ie 30H
7004 0E05 01760 LD C,5
7006 3630 01770 LD (HL),30H
7008 EDB0 01780 LDIR ;INIT SCORE TO 30H
700A C9 01790 RET
01800 ;
01810 ;
700B 21A040 01820 DRHWY LD HL,40A0H ;FILL TOP HWY
700E CD4170 01830 CALL FILHWY
7011 216048 01840 LD HL,4860H ;FILL MIDDLE HWY
7014 CD4170 01850 CALL FILHWY
7017 212050 01860 LD HL,5020H ;FILL BOTTOM HWY
701A CD4170 01870 CALL FILHWY
701D 21A046 01880 LD HL,TOPHY1 ;REVERSE BUILT HIGHWAY
7020 11A047 01890 LD DE,TOPHY2
7023 6F 01900 XOR A
7024 CD3870 01910 CALL HIGHWY
7027 212050 01920 LD HL,BOTHY1
702A 112051 01930 LD DE,BOTHY2
702D CD3870 01940 CALL HIGHWY
7030 216048 01950 LD HL,MIDHY1
7033 11604C 01960 LD DE,MIDHY2
7036 3EC3 01970 LD A,195 ;BIN 11000011
703B 0620 01980 HIGHWY LD B,32 ;32*8 BITS
703A 77 01990 HWYLOP LD (HL),A
703B 12 02000 LD (DE),A
703C 23 02010 INC HL
703D 13 02020 INC DE
703E 10FA 02030 DJNZ HWYLOP
7040 C9 02040 RET
02050 ;
02060 ;
7041 3EFF 02070 FILHWY LD A,OFFH
7043 D9 02080 EXX
7044 0620 02090 LD B,32
7046 D9 02100 FILHYL EXX
7047 E5 02110 PUSH HL
7048 0608 02120 LD B,8
704A 77 02130 FILCHR LD (HL),A
704B 24 02140 INC H
704C 10FC 02150 DJNZ FILCHR

```



```

704E E1      02160      POP      HL
704F 23      02170      INC      HL
7050 09      02180      EXX
7051 10F3    02190      DJNZ     FILHYL
7053 09      02200      EXX
7054 C9      02210      RET
              02220 ;
              02230 ;
              02240 ;***** LINEUP *****
              02250 ;
              02260 ;      draw all frogs left on the screen
              02270 ;
              02280 ;
7055 3E01    02290      LINEUP LD      A,1      ;RIGHT FROG
7057 327B6E  02300      LD      (FRGDIR),A
705A 11D769  02310      LD      DE,FRG2     ;RIGHT FROG SHAPE
705D 2AB46E  02320      LD      HL,(FRGSTN) ;FROG STATION
7060 3E04    02330      LD      A,4      ;(PAPER 0)B+(INK 4)
7062 32A54F  02340      LD      (ATTR),A
7065 3AB26F  02350      LD      A,(NUMFRG) ;NUMBER OF FROG
7068 A7      02360      AND      A      ;TEST FOR NO FROG LEFT
7069 C8      02370      RET      Z
706A 47      02380      LD      B,A      ;NUMBER OF FROG TIMES
706B C5      02390      DRAWLN PUSH    BC
706C D5      02400      PUSH    DE
706D E5      02410      PUSH    HL
706E CD7A70  02420      CALL    DRWFRG ;DRAW FROG ROUTINE
7071 E1      02430      POP      HL
7072 D1      02440      POP      DE
7073 2B      02450      DEC      HL
7074 2B      02460      DEC      HL
7075 2B      02470      DEC      HL
7076 C1      02480      POP      BC
7077 10F2    02490      DJNZ     DRAWLN
7079 C9      02500      RET
              02510 ;
              02520 ;
              02530 ;***** DRWFRG *****
              02540 ;
              02550 ;      similar to DRAW routine
              02560 ;
707A 3E02    02570      DRWFRG LD      A,2      ;TWO ROW FROG SHAPE
707C 08      02580      EX       AF,AF'
707D E5      02590      PUSH    HL      ;STORE POS PTR
707E E5      02600      FRGLP0 PUSH    HL
707F 0E02    02610      LD      C,2      ;COLUMN COUNT
7081 E5      02620      FRGLP1 PUSH    HL
7082 0608    02630      LD      B,B      ;DRAW CHARACTER
7084 1A      02640      FRGLP2 LD      A,(DE)
7085 77      02650      LD      (HL),A
7086 13      02660      INC      DE
7087 24      02670      INC      H      ;NEXT BYTE OF THE CHAR
7088 10FA    02680      DJNZ     FRGLP2
708A E1      02690      POP      HL      ;CURRENT POINTER
708B 23      02700      INC      HL      ;MOVE TO NEXT CHAR POS
708C 0D      02710      DEC      C      ;DECR COLUMN COUNT

```



```

70BD 20F2      02720      JR      NZ,FRGLP1
70BF E1        02730      POP     HL
7090 08        02740      EX      AF,AF'
7091 3D        02750      DEC     A
7092 0E20      02760      LD      C,32
7094 280E      02770      JR      Z,FRGATT
7096 08        02780      EX      AF,AF'
7097 A7        02790      AND     A
7098 ED42      02800      SBC     HL,BC
709A CB44      02810      BIT     0,H
709C 28E0      02820      JR      Z,FRGLP0
709E 7C        02830      LD      A,H
709F D607      02840      SUB     7
70A1 67        02850      LD      H,A
70A2 18DA      02860      JR      FRGLP0
70A4 E1        02870      POP     HL
70A5 7C        02880      LD      A,H
70A6 E618      02890      AND     18H
70A8 CB2F      02900      SRA     A
70AA CB2F      02910      SRA     A
70AC CB2F      02920      SRA     A
70AE C658      02930      ADD     A,58H
70B0 67        02940      LD      H,A
70B1 3A656F    02950      LD      A,(ATTR)
70B4 77        02960      LD      (HL),A
70B5 23        02970      INC     HL
70B6 77        02980      LD      (HL),A
70B7 ED42      02990      SBC     HL,BC
70B9 77        03000      LD      (HL),A
70BA 2B        03010      DEC     HL
70BB 77        03020      LD      (HL),A
70BC C9        03030      RET
03040 ;
03050 ;***** TFCTRL *****
03060 ;
03070 ; Traffic control routine
03080 ;
70BD 21706F    03090      TFCTRL LD      HL,GENFLG ;CHECK REGENERATION FLAG
70C0 AF        03100      XOR     A
70C1 BE        03110      CP      (HL)
70C2 2802      03120      JR      Z,GENER
70C4 35        03130      DEC     (HL)
70C5 C9        03140      RET
70C6 21256E    03150      GENER LD      HL,DB1EXT ;START OF TRAFFIC DB
70C9 110C00    03160      LD      DE,12 ;12 BYTE DATABASE
70CC 0606      03170      LD      B,6 ;6 DB PAIRS
70CE BE        03180      TCTRLP CP      (HL)
70CF 2004      03190      JR      NZ,NSPACE
70D1 CDD970    03200      CALL   REGEN ;REGENERATION ROUTINE
70D4 C9        03210      RET
70D5 19        03220      NSPACE ADD     HL,DE
70D6 10F6      03230      DJNZ   TCTRLP
70DB C9        03240      RET
03250 ;
03260 ;
03270 ;***** REGEN *****

```



```

03280 :
03290 : regeneration of TRAFFIC
03300 : INPUT: HL=>DB PAIRS
03310 :

70D9 E5 03320 REGEN PUSH HL
70DA CDC77 03330 RAND1 CALL RANDNO ;RANDOM NUMBER ROUTINE
70DB E607 03340 AND 7 ;GENERATE RANDOM NUMBER
70DC FE06 03350 CP 6 ;FROM 0 TO 5
70E1 30F7 03360 JR NC,RAND1
70E3 012159 03370 LD BC,5921H ;TWO CHAR TEST
70E6 212059 03380 LD HL,5920H ;TEST JAM
70E9 CB47 03390 BIT 0,A ;ODD NUMBER IS LEFT
70EB 2804 03400 JR Z,RTRAF ;RIGHT TRAFFIC
70ED 2EDF 03410 LD L,0DFH
70EF 0EDE 03420 LD C,0DEH
70F1 87 03430 RTRAF ADD A,A ;GET DBINDEX PTR IN DE
70F2 5F 03440 LD E,A
70F3 0A 03450 LD A,(BC) ;TEST 2 CHAR AHEAD
70F4 86 03460 ADD A,(HL)
70F5 A7 03470 AND A ;ZERO PAPER, ZERO INK
70F6 2802 03480 JR Z,LOADDB ;IF 0, INITIALISE NEW OBJ
70F8 E1 03490 POP HL ;IF JAM, RETURN
70F9 C9 03500 RET
70FA 57 03510 LOADDB LD D,A ;A=0
70FB 21896E 03520 LD HL,DBINDEX ;GET DB
70FE 19 03530 ADD HL,DE
70FF 5E 03540 LD E,(HL) ;GET CORR DATABASE
7100 23 03550 INC HL
7101 56 03560 LD D,(HL)
7102 EB 03570 EX DE,HL ;SOURCE
7103 D1 03580 POP DE ;DESTINATION
7104 010C00 03590 LD BC,12
7107 EDB0 03600 LDIR
7109 3E02 03610 LD A,2 ;SET REGENERATION FLAG
710B 32706F 03620 LD (GENFL6),A ;SKIP FOR 2 CYCLES
710E C9 03630 RET

03640 :
03650 :
03660 : ***** MOVTRF *****
03670 :
03680 : MOVE TRAFFIC ROUTINE
03690 :

710F D9 03700 MOVTRF EXX
7110 21256E 03710 LD HL,OR1EXT
7113 110C00 03720 LD DE,12
7116 0606 03730 LD B,6
7118 E5 03740 MTRFLP PUSH HL
7119 D9 03750 EXX
711A E1 03760 POP HL
711B 7E 03770 LD A,(HL) ;EXISTENCE
711C A7 03780 AND A ;SKIP WHEN NO EXIST
711D CAA771 03790 JP Z,NXTMOV
7120 23 03800 INC HL ;CYCLE COUNT
7121 35 03810 DEC (HL) ;DECR CYCLE COUNT
7122 C2A771 03820 JP NZ,NXTMOV
7125 23 03830 INC HL ;DIRECTION

```


7126 7E	03840	LD	A, (HL)	:G L TO R, 1 R TO L
7127 23	03850	INC	HL	
7128 23	03860	INC	HL	
7129 226E6F	03870	LD	(POS PTR), HL	:POS PTR
712C 5E	03880	LD	E, (HL)	:RESTORE POS
712D 23	03890	INC	HL	
712E 56	03900	LD	D, (HL)	
712F 1C	03910	INC	E	:MOVE RIGHT
7130 A7	03920	AND	A	
7131 2B02	03930	JR	Z, LDPOS	
7133 1D	03940	DEC	E	:MOVE LEFT
7134 1D	03950	DEC	E	:MOVE LEFT
7135 ED536C6F	03960 LDPOS	LD	(NEWPOS), DE	
7139 08	03970	EX	AF, AF?	
713A 010500	03980	LD	BC, 5	:RESTORE OBJ LENGHT
713D 09	03990	ADD	HL, BC	
713E 7E	04000	LD	A, (HL)	:ROW
713F 32606F	04010	LD	(ROW), A	
7142 23	04020	INC	HL	
7143 7E	04030	LD	A, (HL)	:COLUMN
7144 325F6F	04040	LD	(COLUMN), A	
7147 3D	04050	DEC	A	
7148 4F	04060	LD	C, A	
7149 08	04070	EX	AF, AF?	
714A A7	04080	AND	A	:TEST DIRECTION
714B EB	04090	EX	DE, HL	
714C 2008	04100	JR	NZ, RTOL	:RIGHT TO LEFT
714E 09	04110	ADD	HL, BC	:FIND HEAD OF TRUCK
714F 7D	04120	LD	A, L	:LOB
7150 FE40	04130	CP	40H	:TEST RIGHT EDGE
7152 3046	04140	JR	NC, MOVEOK	:SKIP TEST AHEAD IF OFF
7154 1805	04150	JR	TESTAH	:TEST AHEAD
7156 7D	04160 RTOL	LD	A, L	:NEW POS, AHEAD AS WELL
7157 FEC0	04170	CP	0C0H	:TEST LEFT EDGE
7159 383F	04180	JR	C, MOVEOK	:SKIP TEST AHEAD
715B 7C	04190 TESTAH	LD	A, H	:COVERT TO ATTR
715C EA18	04200	AND	18H	
715E CB2F	04210	SRA	A	
7160 CB2F	04220	SRA	A	
7162 CB2F	04230	SRA	A	
7164 C658	04240	ADD	A, 58H	
7166 67	04250	LD	H, A	
7167 012000	04260	LD	BC, 32	
716A AF	04270	XOR	A	
716B 32716F	04280	LD	(JAMFLG), A	:INITIALISE JAM FLAG
716E 3A606F	04290	LD	A, (ROW)	
7171 08	04300 TAHL0P	EX	AF, AF?	
7172 7E	04310	LD	A, (HL)	:RETRIEVE ATTR
7173 E607	04320	AND	7	
7175 280E	04330	JR	Z, TFROG1	:JUMP IF BLACK INK
7177 FE04	04340	CP	4	:TEST FOR GREEN, FROG
7179 2007	04350	JR	NZ, JAM1	:JAM IF NOT A FROG
717B 3E01	04360	LD	A, 1	:MOVE IF IT IS FROG
717D 327C6F	04370	LD	(CRHFLG), A	:SET FROG CRASH
7180 1803	04380	JR	TFROG1	
7182 32716F	04390 JAM1	LD	(JAMFLG), A	:SET JAMFLG NON ZERO



```

7185 A7 04400 TFROG1 AND A
7186 ED42 04410 SBC HL,BC
7188 08 04420 EX AF,AF'
7189 3D 04430 DEC A ;UPDATE ROW
718A 20E5 04440 JR NZ,TAHLOP
718C 3A716F 04450 LD A,(JAMFLG) ;TEST TRAFFIC JAM
718F A7 04460 AND A
7190 2808 04470 JR Z,MOVEOK ;MOVE IF NO JAM
7192 D9 04480 EXX ;ELSE STOP MOVE ONE CYCLE
7193 23 04490 INC HL
7194 34 04500 INC (HL) ;LOAD 2 TO CYCLE COUNT
7195 34 04510 INC (HL)
7196 2B 04520 DEC HL
7197 D9 04530 EXX
7198 180D 04540 JR NXTMOV
719A 2A6E6F 04550 MOVEOK LD HL,(POSPTR) ;RETRIEVE PTR TO POS
719D ED5B6C6F 04560 LD DE,(NEWPOS)
71A1 73 04570 LD (HL),E ;STORE NEWPOS IN DB
71A2 23 04580 INC HL
71A3 72 04590 LD (HL),D
71A4 CDAF71 04600 CALL MVCTRL ;MOVEMENT CONTROL
71A7 D9 04610 NXTMOV EXX
71A8 19 04620 ADD HL,DE
71A9 05 04630 DEC B
71AA C21871 04640 JF NZ,MTRFLP
71AD D9 04650 EXX
71AE C9 04660 RET

04670 :
04680 : ***** MVCTRL *****
04690 :
04700 : Traffic movement control routine
04710 :
71AF 2B 04720 MVCTRL DEC HL
71B0 2B 04730 DEC HL ;DE=>NEWPOS, HL=>DB PTR
71B1 7B 04740 LD A,E ;LOB POS
71B2 E61F 04750 AND 1FH ;TEST EDGE
71B4 2005 04760 JR NZ,CHGRAF ;CHANGE REAL ABS FLAG
71B6 7E 04770 LD A,(HL)
71B7 3C 04780 INC A
71B8 E601 04790 AND 1
71BA 77 04800 LD (HL),A
71BB 2B 04810 CHGRAF DEC HL ;PT DIR
71BC 7E 04820 LD A,(HL)
71BD A7 04830 AND A
71BE 200F 04840 JR NZ,TOLEFT ;RIGHT TO LEFT
71C0 7B 04850 LD A,E
71C1 E61F 04860 AND 1FH ;IF TO RIGHT AND ABS
71C3 201B 04870 JR NZ,DRWOBJ
71C5 23 04880 INC HL ;GET RAF
71C6 7E 04890 LD A,(HL)
71C7 2B 04900 DEC HL ;PT TO DIR
71C8 A7 04910 AND A ;IF ABSTRACT, DIES
71C9 2015 04920 JR NZ,DRWOBJ
71CB D9 04930 EXX
71CC 77 04940 LD (HL),A ;SET NON EXISTENCE
71CD D9 04950 EXX

```



```

71CE C9 04960 RET
71CF 3A5F6F 04970 TOLEFT LD A, (COLUMN)
71D2 4F 04980 LD C, A
71D3 EB 04990 EX DE, HL ; TEST END OF OBJECT
71D4 09 05000 ADD HL, BC ; TOUCHES LEFT EDGE
71D5 7D 05010 LD A, L
71D6 FEC0 05020 CP 0C0H
71D8 EB 05030 EX DE, HL
71D9 2005 05040 JR NZ, DRWOBJ
71DB D9 05050 EXX ; OBJECT NONEXIST AS
71DC 3600 05060 LD (HL), 0 ; IT MOVES OFF SCREEN
71DE D9 05070 EXX
71DF C9 05080 RET
71E0 D9 05090 DRWOBJ EXX
71E1 7E 05100 LD A, (HL)
71E2 23 05110 INC HL
71E3 77 05120 LD (HL), A ; REFILL CYCLE COUNT
71E4 2B 05130 DEC HL
71E5 D9 05140 EXX
71E6 23 05150 INC HL
71E7 E5 05160 PUSH HL ; PT TO RAF
71E8 23 05170 INC HL
71E9 23 05180 INC HL
71EA 23 05190 INC HL
71EB 5E 05200 LD E, (HL) ; RETRIEVE SHAPE PTR
71EC 23 05210 INC HL
71ED 56 05220 LD D, (HL)
71EE 23 05230 INC HL
71EF 4E 05240 LD C, (HL) ; RETRIEVE ATTR PTR
71F0 23 05250 INC HL
71F1 44 05260 LD B, (HL)
71F2 ED436A6F 05270 LD (ATTPTR), BC
71F6 23 05280 INC HL
71F7 7E 05290 LD A, (HL)
71FB 32606F 05300 LD (ROW), A
71FC 23 05310 INC HL
71FD 7E 05320 LD A, (HL)
71FD 325F6F 05330 LD (COLUMN), A
7200 E1 05340 POP HL
7201 7E 05350 LD A, (HL) ; RAFLAG
7202 2A6C6F 05360 LD HL, (NEWPOS)
7205 CD0972 05370 CALL DRAW
7208 C9 05380 RET
05390 ;
05400 ;
05410 ;
05420 ; ***** DRAW *****
05430 ;
05440 ; INPUT : HL=>START OF DISPLAY POS
05450 ; DE=>PTR TO SHAPE DB
05460 ; A =>POSITION REAL/ABSTRACT FLAG
05470 ; C =>NO. OF COL TO BE DISPLAY
05480 ; COL PASS AS VAR
05490 ;
05500 ; VAR COLUMN, ROW, ATTR, DRWPOS
05510 ; SKIP, FILL

```



```

05520 :
05530 :
05540 : REG : A,BC,DE,HL,A'
05550 :
7209 CD9672 05560 DRAW CALL RSHAPE
720C 3A606F 05570 LD A,(ROW) ;RETURN ROW/COL ATTPTR
720F 08 05580 EX AF,AF'
7210 D5 05590 LPO PUSH DE
7211 E5 05600 PUSH HL ;STORE LINE PTR
7212 3A616F 05610 LD A,(SKIP)
7215 4F 05620 LD C,A
7216 0600 05630 LD B,0
7218 09 05640 ADD HL,BC ;SKIP POS PTR
7219 87 05650 ADD A,A ;MULTIPLE OF 8 BYTES
721A 87 05660 ADD A,A
721B 87 05670 ADD A,A
721C 4F 05680 LD C,A ;SKIP SHAPE PTR
721D EB 05690 EX DE,HL
721E 09 05700 ADD HL,BC
721F EB 05710 EX DE,HL
7220 CB44 05720 BIT 0,H ;CROSS SCREEN SECTION?
7222 2B04 05730 JR Z,NOSKIP
7224 3E07 05740 LD A,7
7226 B4 05750 ADD A,H ;IF YES, MOVE UP
7227 67 05760 LD H,A
7228 3A626F 05770 NOSKIP LD A,(FILL)
722B A7 05780 AND A
722C 2B11 05790 JR Z,NXT
722E 4F 05800 LD C,A ;COLUMN TO BE FILLED
722F E5 05810 LP1 PUSH HL ;FILL CHARACTER
7230 060B 05820 LD B,B
7232 1A 05830 LP2 LD A,(DE) ;FILL CHARACTER BYTES
7233 77 05840 LD (HL),A
7234 13 05850 INC DE
7235 24 05860 INC H
7236 10FA 05870 DJNZ LP2
7238 E1 05880 POP HL
7239 0D 05890 DEC C
723A 2B03 05900 JR Z,NXT
723C 23 05910 INC HL ;NEXT CHARACTER
723D 1BF0 05920 JR LP1
723F 08 05930 NXT EX AF,AF'
7240 E1 05940 POP HL ;RESTORE LINE PTR
7241 D1 05950 POP DE ;SHAPE DB PTR
7242 3D 05960 DEC A ;UPDATE ROW COUNT
7243 2B1A 05970 JR Z,LDATTR
7245 08 05980 EX AF,AF'
7246 A7 05990 AND A ;CLEAR CARRY
7247 0E20 06000 LD C,20H
7249 ED42 06010 SRC HL,BC ;ONE LINE UP
724B CB44 06020 BIT 0,H ;CROSS SCREEN SECTION?
724D 2B04 06030 JR Z,MODDB
724F 7C 06040 LD A,H
7250 D607 06050 SUB 7
7252 67 06060 LD H,A
7253 3A5F6F 06070 MODDB LD A,(COLUMN)

```



```

7256 87 06090 ADD A,A
7257 87 06090 ADD A,A
7258 87 06100 ADD A,A ;UPDATE SHAPE DB
7259 4F 06110 LD C,A
725A EB 06120 EX DE,HL
725B 09 06130 ADD HL,BC
725C EB 06140 EX DE,HL
725D 18B1 06150 JR LPO
725F 2A636F 06160 LDATTR LD HL,(ATTRPOS)
7262 ED5B6A6F 06170 LD DE,(ATTRPTR)
7266 3A606F 06180 LD A,(ROW)
7269 08 06190 ATROW EX AF,AF?
726A D5 06200 PUSH DE
726B E5 06210 PUSH HL
726C 3A616F 06220 LD A,(SKIP)
726F 4F 06230 LD C,A
7270 0600 06240 LD B,0
7272 09 06250 ADD HL,BC ;SKIP ATTRIBUTE FILE
7273 EB 06260 EX DE,HL
7274 09 06270 ADD HL,BC ;SKIP ATTRIBUTE DATABASE
7275 EB 06280 EX DE,HL
7276 3A626F 06290 LD A,(FILL)
7279 A7 06300 AND A
727A 2B07 06310 JR Z,SKIPAT ;SKIP ATTRIBUTE
727C 47 06320 LD B,A ;FILL ATTRIBUTES
727D 1A 06330 ATTR2 LD A,(DE)
727E 77 06340 LD (HL),A
727F 23 06350 INC HL
7280 13 06360 INC DE
7281 10FA 06370 DJNZ ATTR2
7283 E1 06380 SKIPAT POP HL
7284 D1 06390 POP DE
7285 3A5F6F 06400 LD A,(COLUMN)
728B A7 06410 AND A ;CLEAR CARRY
7289 0E20 06420 LD C,20H
728B ED42 06430 SBC HL,BC ;NEXT ATTRIBUTE LINE UP
728D 4F 06440 LD C,A
728E EB 06450 EX DE,HL
728F 09 06460 ADD HL,BC ;UPDATE ATTRIBUTE DB
7290 EB 06470 EX DE,HL
7291 0B 06480 EX AF,AF?
7292 3D 06490 DEC A
7293 20D4 06500 JR NZ,ATROW
7295 C9 06510 RET
06520 ;
06530 ;
06540 ;***** RSHAPE *****
06550 ;
06560 ; INPUT: HL=>POSITION
06570 ; A =>REAL/ABSTRACT FLAG
06580 ; DE=>SHAPE PTR
06590 ; COLUMN
06600 ;
06610 ; OUTPUT: SKIP, FILL, ATTRPOS
06620 ;
7296 E5 06630 RSHAPE PUSH HL

```



7297 0B	06640	EX	AF,AF'	;REAL SHAPE
7298 261F	06650	LD	H,1FH	
729A 7C	06660	LD	A,H	
729B AS	06670	AND	L	;TRAP LOWER 5 BITS
729C 6F	06680	LD	L,A	
729D 7C	06690	LD	A,H	
729E 95	06700	SUB	L	;SUBTRACT FROM 1FH
729F 3C	06710	INC	A	
72A0 A4	06720	AND	H	;ADJUST FOR ZERO DIFF
72A1 6F	06730	LD	L,A	
72A2 0B	06740	EX	AF,AF'	
72A3 A7	06750	AND	A	;0=>ABSTRACT, 1=>REAL
72A4 3A5F6F	06760	LD	A,(COLUMN)	
72A7 200A	06770	JR	NZ,REAL	
72A9 95	06780	SUB	L	
72AA 32626F	06790	LD	(FILL),A	
72AD 7D	06800	LD	A,L	;RELOAD ABS DIFF
72AE 32616F	06810	LD	(SKIP),A	
72B1 1B11	06820	JR	CALATT	
72B3 BD	06830 REAL	CP	L	;TAKE MIN OF COL/FILL
72B4 3B07	06840	JR	C,TOOBIG	;FILL MORE THAN COL
72B6 7D	06850	LD	A,L	
72B7 A7	06860	AND	A	
72B8 2003	06870	JR	NZ,TOOBIG	
72BA 3A5F6F	06880	LD	A,(COLUMN)	
72BD 32626F	06890 TOOBIG	LD	(FILL),A	
72D0 AF	06900	XOR	A	
72C1 32616F	06910	LD	(SKIP),A	
72C4 E1	06920 CALATT	POP	HL	;CALCULATE ATT PTR
72C5 E5	06930	PUSH	HL	
72C6 7C	06940	LD	A,H	
72C7 E61B	06950	AND	1BH	
72C9 CB2F	06960	SRA	A	
72CB CB2F	06970	SRA	A	
72CD CB2F	06980	SRA	A	
72CF C65B	06990	ADD	A,5BH	
72D1 67	07000	LD	H,A	
72D2 22636F	07010	LD	(ATTPOS),HL	
72D5 E1	07020	POP	HL	
72D6 C9	07030	RET		
	07040 ;			
	07050 ;			
72D7 210040	07060 CLRSCR	LD	HL,4000H	;HL=>START OF SCREEN
72DA 110140	07070	LD	DE,4001H	
72DD 01FF17	07080	LD	BC,6143	;SIZE OF SCREEN 17FFH
72E0 AF	07090	XOR	A	;BLANK ACREEN
72E1 77	07100	LD	(HL),A	
72E2 ED80	07110	LDIR		
72E4 21005B	07120	LD	HL,5800H	;SET FIRST LINE FOR SCORE
72E7 11015B	07130	LD	DE,5801H	;OF ATTRIBUTE FILE
72EA 011F00	07140	LD	BC,31	
72ED 3607	07150	LD	(HL),7	;INK SEVEN
72EF ED80	07160	LDIR		
72F1 11205B	07170	LD	HL,5B20H	;SET ATTRIBUTE
72F4 11215B	07180	LD	DE,5B21H	;START FROM SECOND LINE
72F7 01DF02	07190	LD	BC,735	



```

72FA 77      07200      LD      (HL),A      ;(PAPER 0)*8 + (INK 0)
72FB EDB0    07210      LDIR
72FD 21A05B  07220      LD      HL,5BA0H    ;SET HIGHWAY
7300 116059  07230      LD      DE,5960H    ;HIGH, MIDDLE, BOTTOM
7303 01205A  07240      LD      BC,5A20H
7306 5E5B    07250      LD      A,56        ;(PAPER 7)*8 + (INK 0)
7308 D9      07260      EXX
7309 0620    07270      LD      B,32        ;FILL ONE LINE
730B D9      07280      HMVATT EXX
730C 77      07290      LD      (HL),A
730D 12      07300      LD      (DE),A
730E 02      07310      LD      (BC),A
730F 23      07320      INC     HL
7310 13      07330      INC     DE
7311 03      07340      INC     BC
7312 D9      07350      EXX
7313 10F6    07360      DJNZ   HMVATT
7315 D9      07370      EXX
7316 C9      07380      RET
          07390      ;
          07400      ;
7317 E5      07410      SHAPE  PUSH   HL      ;SAVE HL PTR
7318 5A7B6E  07420      LD      A,(FRGDIR)
731B B7      07430      ADD     A,A
731C 21AF69  07440      LD      HL,FRGSH
731F 1600    07450      LD      D,0
7321 5F      07460      LD      E,A
7322 19      07470      ADD     HL,DE      ;PTR TO POS OF SHAPE
7323 5E      07480      LD      E,(HL)    ;DE RETURN SHAPE PTR
7324 23      07490      INC     HL
7325 56      07500      LD      D,(HL)
7326 E1      07510      POP     HL
7327 C9      07520      RET
          07530      ;
          07540      ;
          07550      ;***** DISASC *****
          07560      ;
          07570      ;      display ASCII value from character set
          07580      ;      NB:---- store DE, the message pointer
          07590      ;      HL stays the same after display
          07600      ;      used BC register as well
          07610      ;
          07620      ;
7328 C5      07630      DISASC  PUSH   BC
7329 D5      07640      PUSH   DE
732A E5      07650      PUSH   HL
732B 1A      07660      LD      A,(DE)    ;LOAD ASCII CHAR
732C 6F      07670      LD      L,A
732D 2600    07680      LD      H,0
732F 29      07690      ADD     HL,HL
7330 29      07700      ADD     HL,HL
7331 29      07710      ADD     HL,HL
7332 EB      07720      EX      DE,HL
7333 21003C  07730      LD      HL,CHRSET ;START OF CHARACTER SET
7336 19      07740      ADD     HL,DE
7337 EB      07750      EX      DE,HL

```



733B E1	07760	POP	HL	
7339 060B	07770	DRWCHR	LD	B, B ; DRAW CHARACTER
733B E5	07780	PUSH	HL	
733C 1A	07790	CHARLP	LD	A, (DE)
733D 77	07800	LD	(HL), A	
733E 13	07810	INC	DE	
733F 24	07820	INC	H	
7340 10FA	07830	DJNZ	CHARLP	
7342 E1	07840	POP	HL	
7343 D1	07850	POP	DE	
7344 23	07860	HL		; POS PTR
7345 13	07870	INC	DE	; MESSAGE PTR
7346 C1	07880	POP	BC	
7347 10DF	07890	DJNZ	DISASC	
7349 C9	07900	RET		
	07910 ;			
	07920 ;			
734A D9	07930	POLICE	EXX	
734B 216D6E	07940	LD	HL, PCAREXT	
734E 7E	07950	LD	A, (HL)	; TEST POLICE CAR EXIST
734F E5	07960	PUSH	HL	
7350 D9	07970	EXX		
7351 A7	07980	AND	A	
7352 2023	07990	JR	NZ, MOVPC	; MOVE POLICE CAR
735A D1	08000	POP	DE	; DB EXT PTR
7355 CDCC77	08010	CALL	RANDNO	; MOVE WHEN MULTIPLE OF
735B E61F	08020	AND	1FH	; 31
735A FE1F	08030	CP	1FH	
735C C0	08040	RET	NZ	
735D 3E01	08050	LD	A, 1	; SET CHASE FLAG
735F 32726F	08060	LD	(CHASE), A	
7362 21F56E	08070	LD	HL, RPCDB	; RIGHT PC
7365 CDCC77	08080	CALL	RANDNO	
7368 E601	08090	AND	1	
736A 2B03	08100	JR	Z, RHTPC	
736C 21DD6E	08110	LD	HL, LPCDB	
736F 010C00	08120	RHTPC	LD	BC, 12
7372 EDB0	08130	LD1R		
7374 D9	08140	EXX		
7375 E5	08150	PUSH	HL	
7376 D9	08160	EXX		
7377 E1	08170	MOVPC	POP	; EXISTENCE PTR
737B 23	08180	INC	HL	
7379 23	08190	INC	HL	; DIRECTION
737A 7E	08200	LD	A, (HL)	
737B 47	08210	LD	B, A	; STORE DIR
737C 23	08220	INC	HL	
737D 23	08230	INC	HL	; POSPTR
737E 226E6F	08240	LD	(POSPTR), HL	
7381 5E	08250	LD	E, (HL)	
7382 23	08260	INC	HL	
7383 56	08270	LD	D, (HL)	
7384 1C	08280	INC	E	; ASSUME MOVE RIGHT
7385 A7	08290	AND	A	
7386 2B02	08300	JR	Z, PCMRHT	; POLICE CAR MOVE RIGHT
738B 1D	08310	DEC	E	




```

7389 1D 08320 DEC E
738A ED536C6F 08330 PCMHHT LD (NEWPOS),DE
738E 3E02 08340 LD A,2 ;TWO ROW
7390 32606F 08350 LD (ROW),A
7393 3E06 08360 LD A,6
7395 325F6F 08370 LD (COLUMN),A
7398 C5 08380 PUSH BC ;DIRECTION
7399 3A706E 08390 LD A,(PCARRAP) ;REAL/ABS FLAG
739C EB 08400 EX DE,HL ;RET SKIP/FILL,ATTR
739D CD9672 08410 CALL RSHAPE
73A0 2A636F 08420 LD HL,(ATTPOS)
73A3 F1 08430 POP AF
73A4 A7 08440 AND A ;IF 1,OK
73A5 2004 08450 JR NZ,PCTAH ;POLICE CAR TEST AHEAD
73A7 010500 08460 LD BC,5
73AA 09 08470 ADD HL,BC
73AB 7E 08480 LD A,(HL)
73AC E607 08490 PCTAH AND 7
73AE 012000 08500 LD BC,32
73B1 A7 08510 AND A
73B2 ED42 08520 SBC HL,BC
73B4 FE04 08530 CP 4
73B6 2807 08540 JR Z,ISFRG2
73B8 7E 08550 LD A,(HL)
73B9 E607 08560 AND 7
73BB FE04 08570 CP 4
73BD 2009 08580 JR NZ,NFROG2
73BF 3E01 08590 ISFRG2 LD A,1
73C1 327C6F 08600 LD (CRHFLG),A ;SET CRASH FLAG
73C4 3D 08610 DEC A ;BLANK COLOUR
73C5 77 08620 LD (HL),A ;BLANK FRONT OF PC
73C6 09 08630 ADD HL,BC
73C7 77 08640 LD (HL),A ;** SHOULD BLANK FRONT*
73C8 CDDF73 08650 NFROG2 CALL STRPC ;STORE NEW UNDERNEATH
73CB 2A6E6F 08660 LD HL,(POSPTR)
73CE ED5B6C6F 08670 LD DE,(NEWPOS)
73D2 73 08680 LD (HL),E
73D3 23 08690 INC HL
73D4 72 08700 LD (HL),D
73D5 CDAF71 08710 CALL MVCTRL
73D8 D9 08720 EXX ;IF NON-EXIST
73D9 7E 08730 LD A,(HL)
73DA 32726F 08740 LD (CHASE),A
73DD D9 08750 EXX
73DE C9 08760 RET

08770 ;
08780 ;
08790 ;***** STRPC *****
08800 ;
08810 ; STORE UNDERNEATH POLICE CAR
08820 ;

73DF 2A6C6F 08830 STRPC LD HL,(NEWPOS) ;POS PTR
73E2 11AD6D 08840 LD DE,PCSTR ;STORAGE LOC
73E5 EB 08850 EX DE,HL
73E6 73 08860 LD (HL),E ;STORE POSITION
73E7 23 08870 INC HL

```

73EB 72	088B0	LD	(HL),D	
73E9 23	08890	INC	HL	
73EA EB	08900	EX	DE,HL	
73EB 21606F	08910	LD	HL,ROW	
73EE 7E	08920	LD	A,(HL)	
73EF 010500	08930	LD	BC,5	
73F2 EDB0	08940	LDIR		
73F4 08	08950	EX	AF,AF'	
73F5 2A6C6F	08960	LD	HL,(NEWPOS)	
73F8 E5	08970	SPCLP1	PUSH	
73F9 3A616F	08980	LD	A,(SKIP)	
73FC 4F	08990	LD	C,A	
73FD 09	09000	ADD	HL,BC	
73FE CB44	09010	BIT	0,H	
7400 2804	09020	JR	Z,NSSPS	
7402 7C	09030	LD	A,H	
7403 C607	09040	ADD	A,7	
7405 67	09050	LD	H,A	
7406 3A626F	09060	NSSPS	LD	A,(FILL)
7409 A7	09070	AND	A	
740A 280F	09080	JR	Z,NXTSPC	
740C 4F	09090	LD	C,A	
740D E5	09100	SPCLP2	PUSH	
740E 0608	09110	LD	B,8	
7410 7E	09120	SPCLP3	LD	A,(HL)
7411 12	09130	LD	(DE),A	
7412 13	09140	INC	DE	
7413 24	09150	INC	H	
7414 10FA	09160	DJNZ	SPCLP3	
7416 E1	09170	POP	HL	
7417 23	09180	INC	HL	
7418 0D	09190	DEC	C	
7419 20F2	09200	JR	NZ,SPCLP2	
741B E1	09210	NXTSPC	POP	
741C 08	09220	EX	AF,AF'	
741D 3D	09230	DEC	A	
741E 280F	09240	JR	Z,SPCATR	
7420 08	09250	EX	AF,AF'	
7421 0E20	09260	LD	C,32	
7423 ED42	09270	SBC	HL,BC	
7425 CB44	09280	BIT	0,H	
7427 28CF	09290	JR	Z,SPCLP1	
7429 7C	09300	LD	A,H	
742A D607	09310	SUB	7	
742C 67	09320	LD	H,A	
742D 18C9	09330	JR	SPCLP1	
742F 2A636F	09340	SPCATR	LD	HL,(ATTPOS)
7432 3A606F	09350	LD	A,(ROW)	
7435 08	09360	EX	AF,AF'	
7436 E5	09370	SPCAT1	PUSH	
7437 3A616F	09380	LD	A,(SKIP)	
743A 4F	09390	LD	C,A	
743B 09	09400	ADD	HL,BC	
743C 3A626F	09410	LD	A,(FILL)	
743F A7	09420	AND	A	
7440 2803	09430	JR	Z,NXTSPA	

;LOAD 5 BYTES OF INFO

;RESTORE CHAR

;STORE SCREEN FIRST

;NEXT CHAR

;UPD ROW COUNT

;RESTORE POLICE ATTR

;UP ONE LINE

;CROSS SCREEN SECTION?

;ATTRIBUTE START POS



7442	AF	09440	LD	C,A	
7443	EDB0	09450	LDIR		
7445	E1	09460	POP	HL	
7446	08	09470	EX	AF,AF'	
7447	3D	09480	DEC	A	
7448	CB	09490	RET	Z	
7449	08	09500	EX	AF,AF'	
744A	0E20	09510	LD	C,32	
744C	ED42	09520	SBC	HL,BC	
744E	18E6	09530	JR	SPCAT1	
		09540			
		09550			
7450	3A6D6E	09560	RESPC	A,(PCAREXT)	;TEST PC EXIST
7453	A7	09570	AND	A	
7454	CB	09580	RET	Z	
7455	11606F	09590	LD	DE,ROW	
745B	21AF6D	09600	LD	HL,PCSTR+2	
745B	010500	09610	LD	BC,5	
745E	EDB0	09620	LDIR		;RETRIEVE 5 INFO
7460	EB	09630	EX	DE,HL	;DE STORAGE PTR
7461	2AAD6D	09640	LD	HL,(PCSTR)	;LOAD POS
7464	3A606F	09650	LD	A,(ROW)	
7467	08	09660	EX	AF,AF'	
7468	E5	09670	RPCLP1	PUSH	;SAVE POS
7469	3A616F	09680	LD	A,(SKIP)	
746C	4F	09690	LD	C,A	
746D	09	09700	ADD	HL,BC	
746E	CB44	09710	BIT	O,H	
7470	2B04	09720	JR	Z,NSRPS	
7472	3E07	09730	LD	A,7	
7474	84	09740	ADD	A,H	
7475	67	09750	LD	H,A	
7476	3A626F	09760	NSRPS	LD	A,(FILL)
7479	A7	09770	AND	A	
747A	2B0F	09780	JR	Z,NXTRPC	
747C	4F	09790	LD	C,A	
747D	E5	09800	RPCLP2	PUSH	
747E	0608	09810	LD	B,B	
7480	1A	09820	RPCLP3	LD	A,(DE)
7481	77	09830	LD	(HL),A	;RESTORE CHAR
7482	13	09840	INC	DE	
7483	24	09850	INC	H	
7484	10FA	09860	DJNZ	RPCLP3	
7486	E1	09870	POP	HL	
7487	23	09880	INC	HL	
7488	0D	09890	DEC	C	
7489	20F2	09900	JR	NZ,RPCLP2	
748B	E1	09910	NXTRPC	POP	
748C	08	09920	EX	AF,AF'	
748D	3D	09930	DEC	A	;UPD ROW COUNT
748E	2B0F	09940	JR	Z,RPCLP2	;RETORE POLICE CAR
7490	08	09950	EX	AF,AF'	
7491	0E20	09960	LD	C,32	
7493	ED42	09970	SBC	HL,BC	
7495	CB44	09980	BIT	O,H	;MOVE UP ONE LINE
7497	2B0F	09990	JR	Z,RPCLP1	

```

7499 7C      10000      LD      A,H
749A D607    10010      SUB     7 ;CROSS BOUNDARY
749C 67      10020      LD      H,A
749D 18C9    10030      JR      RPCLF1
749F 2A636F  10040  RPCATR LD      HL,(ATTPOS) ;ATTR START LOADING POS
74A2 7A606F  10050      LD      A,(ROW)
74A5 0B      10060      EX      AF,AF'
74A6 E5      10070  RPCAT1 PUSH   HL
74A7 3A616F  10080      LD      A,(SKIP)
74AA 4F      10090      LD      C,A
74AB 09      10100      ADD     HL,BC
74AC 3A626F  10110      LD      A,(FILL)
74AF A7      10120      AND     A
74B0 2B05    10130      JR      Z,NXTRPA
74B2 EB      10140      EX      DE,HL
74B3 4F      10150      LD      C,A
74B4 EDB0    10160      LDIR
74B6 EB      10170      EX      DE,HL
74B7 E1      10180  NXTRPA POP     HL
74B9 0B      10190      EX      AF,AF'
74B9 3D      10200      DEC     A
74BA CB      10210      RET     Z
74BB 0B      10220      EX      AF,AF'
74BC 0E20    10230      LD      C,32
74BE ED42    10240      SEC
74C0 1BE4    10250      JR      RPCAT1
10260 ;
10270 ;
74C2 3A7C6F  10280  FROG LD      A,(CRHFLG) ;CRASH FLAG
74C5 A7      10290      AND     A
74C6 2017    10300      JR      NZ,FROGRH ;FROG CRASH
74C8 32DE6F  10310      LD      (UPDOWN),A ;SET NO SCORE
74CB CDE374  10320      CALL   REGFRG ;REGENERATE FROG
74CE 217A6E  10330      LD      HL,FROCYC ;TEST MOVE
74D1 35      10340      DEC     (HL)
74D2 C0      10350      RET     NZ
74D3 2B      10360      DEC     HL
74D4 7E      10370      LD      A,(HL) ;RESET CYCLE COUNT
74D5 23      10380      INC     HL
74D6 77      10390      LD      (HL),A
74D7 CD1075  10400      CALL   MOVFRG
74DA 3A7C6F  10410      LD      A,(CRHFLG)
74DD A7      10420      AND     A
74DE CB      10430      RET     Z
74DF CD9176  10440  FROGRH CALL   CRASH
74E2 C9      10450      RET
10460 ;
10470 ;***** REGFRG *****
10480 ;
10490 ; Regenerate frog if any left
10500 ; Set GAMFLG to 0 if none left
10510 ;
74E3 3A796E  10520  REGFRG LD      A,(FROEXT)
74E6 A7      10530      AND     A
74E7 C0      10540      RET     NZ ;RETURN IF EXIST
74E8 21B16E  10550      LD      HL,FRODB

```



```

74EB 11796E 10560 LD DE,FRGEXT
74EE 010800 10570 LD BC,B
74F1 EDB0 10580 LDIR
74F3 21846E 10590 LD HL,FRGSTN ;UPDATE FRG STATION
74F6 35 10600 DEC (HL) ;MOVE 3 CHARACTER LEFT
74F7 35 10610 DEC (HL)
74FB 35 10620 DEC (HL)
74F9 2A7C6E 10630 LD HL,(FRGPOS)
74FC 22786F 10640 LD (OLDFRG),HL
74FF 22786F 10650 LD (NEWFRG),HL
7502 21896D 10660 LD HL,FRGSTR ;INIT FRG STR FOR RES
7505 118A6D 10670 LD DE,FRGSTR+1 ;BLANK FRG STORE
7508 012300 10680 LD BC,35
750B 3600 10690 LD (HL),0
750D EDB0 10700 LDIR
750F C9 10710 RET
10720 ;
10730 ;***** MOVFRG *****
10740 ;
10750 ; Move frog, store and restore
10760 ;
7510 AF 10770 MOVFRG XOR A
7511 2120E0 10780 LD HL,0E020H ;H=-32, L=32
7514 4F 10790 LD C,A ;C->ABS MOVEMENT
7515 08 10800 EX AF,AF'
7516 3EDF 10810 LD A,0DFH ;TEST RIGHT
7518 DBFE 10820 IN A,(0FEH)
751A E601 10830 AND 1
751C 2006 10840 JR NZ,LEFT
751E 0C 10850 INC C
751F 11D769 10860 LD DE,FRG2
7522 0601 10870 LD B,1
7524 3EDF 10880 LD A,0DFH ;TEST LEFT
7526 DBFE 10890 IN A,(0FEH)
7528 E604 10900 AND 4
752A 2006 10910 JR NZ,DOWN
752C 0D 10920 DEC C
752D 11176A 10930 LD DE,FRG4
7530 0603 10940 LD B,3 ;TEST DOWN
7532 3EDF 10950 LD A,0DFH
7534 DBFE 10960 IN A,(0FEH)
7536 E601 10970 AND 1
7538 200B 10980 JR NZ,UP
753A 79 10990 LD A,C
753B 85 11000 ADD A,L ;ADD 32
753C 4F 11010 LD C,A
753D 08 11020 EX AF,AF'
753E 3D 11030 DEC A
753F 08 11040 EX AF,AF' ;DEC UP/DWN FLG
7540 11F769 11050 LD DE,FRG3
7543 0602 11060 LD B,2
7545 3EF7 11070 LD A,0F7H ;TEST UP
7547 DBFE 11080 IN A,(0FEH)
7549 E601 11090 AND 1
754B 200B 11100 JR NZ,VALID
754D 79 11110 LD A,C

```



754E 84	11120	ADD	A,H	;ADD -32
754F 4F	11130	LD	C,A	
7550 08	11140	EX	AF,AF?	
7551 3C	11150	INC	A	
7552 08	11160	EX	AF,AF?	
7553 11B769	11170	LD	DE,FROG1	
7556 0600	11180	LD	B,0	
7558 78	11190	VALID	A,B	;STORE TEMP DIR
7559 327D6F	11200	LD	(TENDIR),A	
755C ED53806F	11210	LD	(TEMSP),DE	;STORE TEMP SHAPE
7560 AF	11220	XOR	A	
7561 89	11230	CP	C	
7562 C9	11240	RET	Z	;IF NO MOVE GO BACK
7563 2A786F	11250	LD	HL,(OLDPRG)	
7566 CB79	11260	BIT	7,C	;TEST -VE
7568 47	11270	LD	B,A	
7569 1E07	11280	LD	E,7	;FOR BOUNDARY ADJ
756B 2803	11290	JR	Z,NETDWN	;NET MOVE RHT,DWN
756D 05	11300	DEC	B	
756E 1EF9	11310	LD	E,-7	
7570 09	11320	NETDWN	ADD	HL,BC
7571 CB44	11330	BIT	O,H	
757X 2803	11340	JR	Z,VALID1	;NO CROSS BOUNDARY
7575 7C	11350	LD	A,H	
7576 83	11360	ADD	A,E	
7577 67	11370	LD	H,A	;ADJ HOB
7578 227E6F	11380	VALID1	LD	(TEMPOS),HL
757B EB	11390	EX	DE,HL	
757C 3E40	11400	LD	A,40H	;TEST UPSCR
757E 84	11410	CP	D	
757F 7B	11420	LD	A,E	
7580 2004	11430	JR	NZ,VALID2	
7582 FE20	11440	CP	20H	
7584 382F	11450	JR	C,NVALID	
7586 E61F	11460	VALID2	AND	1FH
7588 FE1F	11470	CP	1FH	
758A 2829	11480	JR	Z,NVALID	
758C 21BE50	11490	LD	HL,50BEH	;TEST BOT BOUNDARY
758F 47	11500	AND	A	
7590 ED52	11510	SBC	HL,DE	
7592 3821	11520	JR	C,NVALID	
7594 217E50	11530	LD	HL,507EH	;TEST FROG STATION
7597 ED52	11540	SBC	HL,DE	
7599 3011	11550	JR	NC,YVALID	
759B 7B	11560	LD	A,E	;TEST WITHIN BOX
759C E61F	11570	AND	1FH	
759E 67	11580	LD	H,A	
759F 3A846E	11590	LD	A,(FRGSTN)	
75A2 FE00	11600	CP	0A0H	;TEST LAST FROG
75A4 3806	11610	JR	C,YVALID	;NO MORE FROG STATION
75A6 3C	11620	INC	A	;WHEN NO FROG LEFT
75A7 E61F	11630	AND	1FH	
75A9 94	11640	SUB	H	
75AA 3009	11650	JR	NC,NVALID	
75AC ED537A6F	11660	YVALID	LD	(NEWPRG),DE
75B0 08	11670	EX	AF,AF?	;STORE NEW POS



75B1 325E6F	11680	LD	(UPDOWN),A	
75B4 08	11690	EX	AF,AF'	
75B5 2A7B6F	11700	LD	HL,(OLDFRG)	; TEST OLDFRG=NEWFRG
75B8 A7	11710	AND	A	
75B9 ED52	11720	SBC	HL,DE	
75BB 7D	11730	LD	A,L	
75BC B4	11740	DR	H	
75BD CB	11750	RET	Z	; RETURN IF SAME
75BE CDD675	11760	CALL	RESFRG	; RESTORE FROG
75C1 2A7A6F	11770	LD	HL,(NEWFRG)	; UPDATE OLD FROG POS
75C4 227B6F	11780	LD	(OLDFRG),HL	
75C7 217B6F	11790	LD	HL,TEMDIR	
75CA 117B6E	11800	LD	DE,FRGDIR	
75CD 010500	11810	LD	BC,5	
75D0 EDB0	11820	LDIR		
75D2 CD2B76	11830	CALL	STRFRG	
75D5 C9	11840	RET		
	11850 ;			
	11860 ;			
75D6 11B96D	11870	RESFRG	LD	DE,FRGSTR ; STORAGE PTR
75D9 2A7B6F	11880	LD	HL,(OLDFRG)	; RESTORE FROM OLDPOS
75DC E5	11890	PUSH	HL	
75DD 3E02	11900	LD	A,2	; ROW COUNTER
75DF 08	11910	EX	AF,AF'	
75E0 E5	11920	RFRLP1	PUSH	
75E1 0E02	11930	LD	C,2	; COLUMN COUNTER
75E3 E5	11940	RFRLP2	PUSH	
75E4 0608	11950	LD	B,B	
75E6 1A	11960	RFRLP3	LD	A,(DE) ; RESTORE FROM DB
75E7 77	11970	LD	(HL),A	; ONTO SCREEN
75E8 15	11980	INC	DE	
75E9 24	11990	INC	H	; NEXT CHAR BYTE
75EA 10FA	12000	DJNZ	RFRLP3	
75EC E1	12010	POP	HL	
75ED 25	12020	INC	HL	
75EE 0D	12030	DEC	C	; COLUMN COUNT
75EF 20F2	12040	JR	NZ,RFRLP2	
75F1 E1	12050	POP	HL	
75F2 08	12060	EX	AF,AF'	
75F3 3D	12070	DEC	A	; ROW COUNT
75F4 2B10	12080	JR	Z,RFRLP1	
75F6 08	12090	EX	AF,AF'	
75F7 A7	12100	AND	A	
75F8 0E20	12110	LD	C,32	; UP ONE LINE
75FA ED42	12120	SBC	HL,BC	
75FC CB44	12130	BIT	0,H	
75FE 2BE0	12140	JR	Z,RFRLP1	
7600 7C	12150	LD	A,H	
7601 D607	12160	SUB	7	
7603 67	12170	LD	H,A	
7604 1BDA	12180	JR	RFRLP1	
7606 E1	12190	RFRLP1	POP	HL
7607 7C	12200	LD	A,H	
7608 E61B	12210	AND	1BH	
760A CB2F	12220	SRA	A	
760C CB2F	12230	SRA	A	



760E CB2F	12240	SRA	A	
7610 C65B	12250	ADD	A,5BH	
7612 67	12260	LD	H,A	
7613 3E02	12270	LD	A,2	;ROW COUNTER
7615 0B	12280	EX	AF,AF'	
7616 E5	12290 RFRAT1	PUSH	HL	
7617 EB	12300	EX	DE,HL	
7618 0E02	12310	LD	C,2	;RESTORE ATTR
761A EDB0	12320	LDIR		
761C EB	12330	EX	DE,HL	
761D E1	12340	POP	HL	
761E 0B	12350	EX	AF,AF'	
761F 3D	12360	DEC	A	;UPDATE ROW COUNTER
7620 CB	12370	RET	Z	
7621 0B	12380	EX	AF,AF'	
7622 0E20	12390	LD	C,32	
7624 ED42	12400	SBC	HL,BC	
7626 18EE	12410	JR	RFRAT1	
	12420 ;			
	12430 ;			
762B 11896D	12440 STRFRG	LD	DE,FRGSTR	
762B 2A7A6F	12450	LD	HL,(NEWFRG)	;STORE BASE ON NEWPOS
762E D9	12460	EXX		
762F 2A7E6E	12470	LD	HL,(FROGSH)	;LOAD SHAPE AS WELL
7632 D9	12480	EXX		
7633 E5	12490	PUSH	HL	
7634 3E02	12500	LD	A,2	
7636 0B	12510	EX	AF,AF'	
7637 E5	12520 SFRLP1	PUSH	HL	
763B 0E02	12530	LD	C,2	
763B E5	12540 SFRLP2	PUSH	HL	
763B 0A0B	12550	LD	B,8	;STORE AND LOAD A CHAR
763D 7E	12560 SFRLP3	LD	A,(HL)	
763E 12	12570	LD	(DE),A	
763F D9	12580	EXX		
7640 7E	12590	LD	A,(HL)	
7641 23	12600	INC	HL	
7642 D9	12610	EXX		
7643 77	12620	LD	(HL),A	
7644 13	12630	INC	DE	
7645 24	12640	INC	H	
7646 10F5	12650	DJNZ	SFRLP3	
764B E1	12660	POP	HL	
7649 23	12670	INC	HL	;NEXT CHAR
764A 0B	12680	DEC	C	
764B 20ED	12690	JR	NZ,SFRLP2	
764D E1	12700	POP	HL	
764E 0B	12710	EX	AF,AF'	
764F 3D	12720	DEC	A	
7650 2B10	12730	JR	Z,SFRATR	
7652 0B	12740	EX	AF,AF'	
7653 A7	12750	AND	A	
7654 0E20	12760	LD	C,32	
7656 ED42	12770	SBC	HL,BC	;NEXT ROW
765B CB44	12780	BIT	O,H	
765A 2BDB	12790	JR	Z,SFRLP1	



765C 7C	12800	LD	A,H	
765D D607	12810	SUB	7	
765F 67	12820	LD	H,A	
7660 18D5	12830	JR	SFRLP1	
7662 E1	12840	SFRATR	POP	
7663 7C	12850	LD	A,H	;CALCULATE ATTR POS
7664 E61B	12860	AND	1SH	
7666 CB2F	12870	SRA	A	
7668 CB2F	12880	SRA	A	
766A CB2F	12890	SRA	A	
766C C65B	12900	ADD	A,5BH	
766E 67	12910	LD	H,A	
766F 3E02	12920	LD	A,2	
7671 0B	12930	EX	AF,AF'	
7672 0602	12940	SFRAT1	LD	
7674 E5	12950	PUSH	B,2	
7675 7E	12960	SFRATLP	LD	
7676 12	12970	LD	A,(HL)	
7677 3604	12980	LD	(DE),A	
7679 23	12990	LD	(HL),4	;FILL FROG ATTR
767A 13	13000	INC	HL	
767B E607	13010	AND	DE	
767D 2B05	13020	JR	7	;TEST CRASH
767F 3E01	13030	JR	Z,NFROG3	
7681 327C6F	13040	LD	A,1	
7684 10EF	13050	NFROG3	(CRHFLG),A	
7686 E1	13060	POP	SFRATLP	
7687 0B	13070	EX	HL	
7688 3D	13080	DEC	AF,AF'	
7689 CB	13090	RET	A	
768A 0B	13100	EX	Z	
768B 0E20	13110	LD	AF,AF'	
768D ED42	13120	SBC	C,32	
768F 1BE1	13130	JR	HL,BC	
	13140		SFRAT1	
	13150			
7691 AF	13160	CRASH	XOR	
7692 327C6F	13170	LD	A	
7695 3279AE	13180	LD	(CRHFLG),A	;RESET CRASH FLAG
7698 CDA776	13190	CALL	(FRGEXT),A	;SET FROG NONEXIST
769B CDD675	13200	CALL	FRGDIE	;FROG DYING ROUTINE
769E 21B26F	13210	LD	RESFRG	
76A1 35	13220	DEC	HL,NUMFRG	
76A2 C0	13230	RET	(HL)	;DECREASE FROG NUMBER
76A3 32776F	13240	LD	NZ	
76A6 C9	13250	RET	(GAMFLG),A	;ZEROISE GAME FLAG
	13260			;WHEN NO FROG LEFT
	13270			
76A7 2A786F	13280	FRGDIE	LD	
76AA 010240	13290	LD	HL,(OLDFRG)	;OLD POS OF FRG
76AD D9	13300	EXX	BC,4002H	;RED COLOUR
76AE 21396F	13310	LD		
76B1 D9	13320	EXX	HL,DIETON	;SET DIE TONE
76B2 7C	13330	LD	A,H	
76B3 B8	13340	CP	B	;TEST END OF JOURNEY
76B4 2016	13350	JR	NZ,NOTEND	

76B6 7D	13360	LD	A, L	
76B7 B8	13370	CP	B	
76B8 3012	13380	JR	NC, NOTEND	
76BA 11466F	13390	LD	DE, SCORE+3	:100 PTS BONUS
76BD E5	13400	EX	DE, HL	
76BE 34	13410	INC	(HL)	
76BF 21476F	13420	LD	HL, SCORE+4	
76C2 CD4877	13430	CALL	DISSCR	
76C5 0E06	13440	LD	C, 6	:YELLOW
76C7 D9	13450	EXX		
76CB 21156F	13460	LD	HL, HOMTON	
76CB D9	13470	EXX		
76CC 79	13480	NOTEND		
76CD 32656F	13490	LD	A, C	
76D0 2A786F	13500	LD	(ATTR), A	
76D3 ED5B7E6E	13510	LD	HL, (OLDFRG)	
76D7 CD7A70	13520	CALL	DE, (FROGSH)	
76DA 112000	13530	LD	DRWFRG	
76DD 19	13540	ADD	DE, 32	:LINE ADJUST
76DE 08	13550	EX	HL, DE	
76DF 3A656F	13560	LD	AF, AF'	
76E2 08	13570	EX	A, (ATTR)	
76E3 0605	13580	LD	AF, AF'	
76E5 C5	13590	FLASLP	B, 5	
76E6 E5	13600	PUSH	BC	
76E7 AF	13610	PUSH	HL	:ATTRIBUTE PTR
76EB 77	13620	XOR	A	:BLACK INK BLACK PAPER
76E9 23	13630	LD	(HL), A	
76EA 77	13640	INC	HL	
76EB ED52	13650	LD	(HL), A	
76ED 77	13660	SBC	HL, DE	
76EE 2B	13670	LD	(HL), A	
76EF 77	13680	DEC	HL	
76F0 CD0877	13690	LD	(HL), A	
76F3 E1	13700	CALL	FRGTON	:GENERATE FRGS TONE
76F4 E5	13710	POP	HL	
76F5 08	13720	PUSH	HL	
76F6 77	13730	EX	AF, AF'	
76F7 23	13740	LD	(HL), A	:BLACK PAPER, RED OR
76F8 77	13750	INC	HL	:YELLOW INK
76F9 A7	13760	LD	(HL), A	
76FA ED52	13770	AND	A	
76FC 77	13780	SBC	HL, DE	
76FD 2B	13790	LD	(HL), A	
76FE 77	13800	DEC	HL	
76FF 08	13810	LD	(HL), A	
7700 CD0877	13820	EX	AF, AF'	
7703 E1	13830	CALL	FRGTON	
7704 C1	13840	POP	HL	
7705 10DE	13850	POP	BC	
7707 C9	13860	DYNZ	FLASLP	
	13870	RET		
	13880			
770B D9	13890	FRGTON	EXX	
7709 E5	17900	PUSH	HL	
770A CDB577	13910	CALL	TONE1	



770D E1	13920	POP	HL	
770E 010400	13930	LD	BC, 4	; MOVE DOWN DATABASE
7711 08	13940	EX	AF, AF'	
7712 FE06	13950	CP	6	
7714 2803	13960	JR	Z, HOME	
7716 01FCFF	13970	LD	BC, -4	; MOVE UP DATABASE
7719 09	13980	HOME	ADD	
771A D9	13990	EXX	HL, BC	
771B 08	14000	EX	AF, AF'	
771C C9	14010	RET		
	14020			
	14030			
771D 3A796E	14040	CALSCR	LD	A, (FRGEXT)
7720 A7	14050	AND	A	; TEST EXISTENCE
7721 C8	14060	RET	Z	; NO UPDATE OF SCORE
7722 3A5E6F	14070	LD	A, (UPDN)	; TEST UP/DOWN MOVEMENT
7725 A7	14080	AND	A	; TEST ANY SCORE
7726 C8	14090	RET	Z	
7727 21476F	14100	LD	HL, SCORE+4	; ADD 10 TO SCORE
772A CB7F	14110	BIT	Z, A	; TEST MOVE DOWN
772C 2003	14120	JR	NZ, DNNSCR	; DOWN SCORE
772E 34	14130	INC	(HL)	
772F 181A	14140	JR	DISSCR	; DIS SCORE
7731 3A796F	14150	DNNSCR	LD	A, (OLDFRG+1)
7734 FE40	14160	CP	40H	; TEST HUB
7736 2009	14170	JR	NZ, TLHWY	; TEST FIRST BLOCK
7738 3A786F	14180	LD	A, (OLDFRG)	; TEST LOW HIGHWAY
773B FE00	14190	CP	0C0H	; NOT EVEN STEP ON HWY
773D DB	14200	RET	C	
773E 34	14210	INC	(HL)	
773F 180A	14220	JR	DISSCR	
7741 FE50	14230	TLHWY	CP	50H
7743 C0	14240	RET	NZ	; TEST IN LOW HWY
7744 3A786F	14250	LD	A, (OLDFRG)	
7747 FE20	14260	CP	20H	
7749 D0	14270	RET	NC	; NO SCORE IF STEP HWY
774A 34	14280	INC	(HL)	
774B 0604	14290	DISSCR	LD	B, 4
774D 7E	14300	ADDOP	LD	A, (HL)
774E FE3A	14310	CRYLOP	CP	3AH
7750 3807	14320	JR	C, UPDDIG	; CARRY LOOP
7752 D60A	14330	SUB	10	; UPDATE DIGIT
7754 2B	14340	DEC	HL	
7755 34	14350	INC	(HL)	; CARRY
7756 23	14360	INC	HL	
7757 18F5	14370	JR	CRYLOP	
7759 77	14380	UPDDIG	LD	(HL), A
775A 2B	14390	DEC	HL	
775B 10F0	14400	DJNZ	ADDOP	
775D 21446F	14410	LD	HL, SCORE+1	; SCORE IMAGE
7760 CD6F77	14420	CALL	SCRIMG	
7763 210640	14430	LD	HL, 4006H	
7766 11596F	14440	LD	DE, IMAGE	
7769 0605	14450	LD	B, 5	
776B CD2873	14460	CALL	DISASC	
776E C9	14470	RET		

```

14480 ;
14490 ;
776F 11596F 14500 SCRIMG LD DE,IMAGE
7772 010500 14510 LD BC,5
7775 EDB0 14520 LDIR
7777 21596F 14530 LD HL,IMAGE
777A 013004 14540 LD BC,0430H
777D 79 14550 PREZER LD A,C
777E BE 14560 CP ;TEST 30H
777F 2005 14570 JR NZ,PREZEX
7781 3620 14580 LD ;SPACE FILL
7783 25 14590 INC HL
7784 10F7 14600 DJNZ PREZER
7786 C9 14610 PREZEX RET
14620 ;
14630 ;
7787 3EBF 14640 SIREN LD A,0BFH
7789 DBFE 14650 IN A,(0FEH)
778B E601 14660 AND I
778D 2009 14670 JR NZ,NSOUND
778F 3A736F 14680 LD A,(SOUNDF) ;RESET SOUND CONDITION
7792 3C 14690 INC A
7793 E601 14700 AND I
7795 32736F 14710 LD (SOUNDF),A
7798 3A736F 14720 NSOUND LD A,(SOUNDF)
779B A7 14730 AND A
779C 2B25 14740 JR Z,DELAY
779E 3A726F 14750 LD A,(CHASE) ;IS POLICE CAR ON
77A1 A7 14760 AND A
77A2 2B1F 14770 JR Z,DELAY
77A4 3A746F 14780 LD A,(TONFLG)
77A7 3C 14790 INC A
77AB E601 14800 AND I
77AA 32746F 14810 LD (TONFLG),A
77AD 210D6F 14820 LD HL,PCTON1
77B0 2B03 14830 JR Z,TONE1
77B2 21116F 14840 LD HL,PCTON2
77B5 5E 14850 TONE1 LD E,(HL) ;DE=DURATION*FREQUENCY
77B6 23 14860 INC HL
77B7 56 14870 LD D,(HL)
77B8 23 14880 INC HL
77B9 4E 14890 LD C,(HL)
77BA 23 14900 INC HL
77BB 46 14910 LD B,(HL)
77BC C5 14920 PUSH BC
77BD E1 14930 POP HL ;HL=437500/FREQ-30.125
77BE CDB503 14940 CALL 03BSH
77C1 F3 14950 DI ;03BSH ENABLE INTERRUPT
77C2 C9 14960 RET
77C3 01001B 14970 DELAY LD BC,6144
77C6 0B 14980 WAIT DEC BC
77C7 7B 14990 LD A,B
77CB B1 15000 OR C
77C9 20FB 15010 JR NZ,WAIT
77CB C9 15020 RET
15030 ;

```



```

15040 :
77CC E5 15050 RANDNO PUSH HL
77CD C5 15060 PUSH BC
77CE 2A756F 15070 LD HL,(RND)
77D1 46 15080 LD B,(HL)
77D2 23 15090 INC HL
77D3 3E3F 15100 LD A,3FH ;BOUND POINTER WITHIN ROM
77D5 A4 15110 AND H
77D6 67 15120 LD H,A
77D7 78 15130 LD A,B
77D8 22756F 15140 LD (RND),HL
77DB C1 15150 POP BC
77DC E1 15160 POP HL
77DD C9 15170 RET
15180 :
15190 :
77DE 21446F 15200 OVER LD HL,SCORE+1 ;HIGH SCORE MANAGE
77E1 11546F 15210 LD DE,HISCR
77E4 0605 15220 LD B,S
77E6 1A 15230 SORTLF LD A,(E)
77E7 BE 15240 CF (HL)
77E8 2803 15250 JR Z,SAMSCR ;TEST 1ST NE DIGIT
77EA D0 15260 RET NC
77EB 1805 15270 JR SCRG1 ;UPDATE HIGH SCORE
77ED 13 15280 SAMSCR INC DE
77EE 23 15290 INC HL
77EF 10F5 15300 DJNZ SORTLF
77F1 C9 15310 RET
77F2 21446F 15320 SCRG1 LD HL,SCORE+1
77F5 11546F 15330 LD DE,HISCR
77F8 010500 15340 LD BC,S
77FB EDB6 15350 LDIR
77FD C9 15360 RET
15370 :
15380 :
77FE 3E38 15390 FINAL LD A,56 ;SET WHITE BORDER
7800 22485C 15400 LD (23624),A
7803 210040 15410 LD HL,4000H ;START OF SCREEN
7806 110140 15420 LD DE,4001H
7809 01FF17 15430 LD BC,6143 ;SIZE OF SCREEN
780C 3600 15440 LD (HL),0
780E EDB0 15450 LDIR
7810 210058 15460 LD HL,5800H ;START OF ATTRIBUTE FILE
7813 110158 15470 LD DE,5801H
7816 01FF02 15480 LD BC,767
7819 2658 15490 LD (HL),56 ;WHITE PAPER BLACK INK
781B EDB0 15500 LDIR
781D C9 15510 RET
15520 :
15530 :
6978 15540 END START
00000 Total errors

```



APENDICE A

TABLA DE ENTRADA DE LAS TECLAS DEL SPECTRUM

Input Value in A	D4	D3	D2	D1	D0
0FE H	V	C	X	Z	CAP SHIFT
0FD H	G	F	D	S	A
0FB H	7	R	E	W	G
0F7 H	5	4	3	2	1
0EF H	6	7	8	9	0
0DF H	Y	U	I	O	P
0BF H	H	J	K	L	ENTER
07F H	B	N	M	SVM SHIFT	BREAK SPACE
X:	16	8	4	2	1

NB: Para atrapar una tecla

- Cargar A con el valor de entrada de la correspondiente fila
LD A, 07FH ; fila inferior
- Recoger del portal de entrada 0EFH.
IN A, (0FEH)
- Comprobar si Dx está puesto a cero para la tecla buscada.
AND I ; atrapa la tecla BREAK/SPACE
- Si es cero, entonces la tecla está pulsada.
JR Z, teclasí ; es estado normal es siempre alto.

Direcciones en HEX
de la zona de memoria

APENDICE B

Direcciones en HEX
de las zonas de memoria

PANTALLA ATRIBUTOS		PANTALLA ATRIBUTOS	
IN HEX	IN HEX	IN HEX	IN HEX
4000	5800	0	401F
4020	5820	1	403F
4040	5840	2	405F
4060	5860	3	407F
4080	5880	4	409F
40A0	58A0	5	40BF
40C0	58C0	6	40DF
40E0	58E0	7	40FF
4800	5900	8	481F
4820	5920	9	483F
4840	5940	10	485F
4860	5960	11	487F
4880	5980	12	489F
48A0	59A0	13	48BF
48C0	59C0	14	48DF
48E0	59E0	15	48FF
5000	5A00	16	501F
5020	5A20	17	503F
5040	5A40	18	505F
5060	5A60	19	507F
5080	5A80	20	509F
50A0	5AA0	21	50BF
50C0	5AC0	22	50DF
50E0	5AE0	23	50FF



APENDICE C

REPERTORIO DE SIMBOLOS DEL SPECTRUM

HEX	HOB	0	1	2	3	4	5	6	7
LOB	BITS	000	001	010	011	100	101	110	111
0	0000	NU	INK ctrl	SPACE	0	@	P	E	p
/	0001	NU	PAPER ctrl	!	1	A	Q	a	q
2	0010	NU	FLASH ctrl	"	2	B	R	b	r
3	0011	NU	BRIGHT ctrl	#	3	C	S	c	s
4	0100	NU	INVERSE ctrl	\$	4	D	T	d	t
5	0101	Nu	OVER ctrl	%	5	E	U	e	u
6	0110	PRINT	AT ctrl	&	6	F	V	f	v
7	0111	EDIT	TAB ctrl	'	7	G	W	g	w
8	1000	cursor left	NU	(8	H	X	h	x
9	1001	cursor right	NU)	9	I	Y	i	y
A	1010	cursor down	NU	*	:	J	Z	j	z
B	1011	cursor up	NU	+	;	K	[k	{
C	1100	DELETE	NU	,	<	L	\	l	
D	1101	ENTER	NU	-	=	M]	m	}
E	1110	number	NU	.	>	N	^	n	~
F	1111	NU	NU	/	?	O	_	o	©

← NO IMPRIMIBLES → IMPRIMIBLES

NB: NU indica "no usado"

APENDICE D
TABLA DE CONVERSION DE HEXADECIMAL A DECIMAL

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	00XX	X'000
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	256	4096
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	512	8192
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	768	12288
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	1024	16384
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	1280	20480
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	1536	24576
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	1792	28672
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	2048	32768
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	2304	36864
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	2560	40960
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	2816	45056
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	3072	49152
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	3328	53248
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	3584	57344
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	3840	61440



APENDICE D

Podemos mostrar cómo se usa esta tabla mediante un ejemplo.

Hallemos el equivalente en hexadecimal del número decimal 6200. Tenemos que determinar un número binario de 16 bits;

ie.

bbbbbbbb	bbbbbbbb
HOB	LOB

- i.) A partir de la columna de la izquierda y bajo el epígrafe xx00, vemos que 6200 está entre 4096 y 8192. Así que escogemos el valor inferior (4096) y de la correspondiente fila, tomamos los 4 bits más significativos del HOB (alto orden octeto) que será 1, por tanto, 0001

0001bbbb	bbbbbbbb
HOB	LOB

- ii.) El segundo paso, es determinar los 4 bits menos significativos del HOB. Hallamos la diferencia entre 6200 y 4096, que es 2104. Dado que esta diferencia es todavía mayor que 255, consultamos la segunda columna del extremo derecho de la tabla, bajo el epígrafe 00xx, y veremos que 2104 está entre 2048 y 2304. De nuevo, tomamos el valor inferior (2048) y buscamos el valor de la fila que corresponderá a los 4 bits menos significativos del HOB, que es el 8, por tanto, 1000.

0001 1000	bbbbbbbb
HOB	LOB

- iii.) El tercer paso, es determinar el LOB (bajo orden octeto) de ese número. Hallamos la diferencia entre 2104 y 2048 que es 56. En la parte media de la tabla podemos ver que 56 está en la intersección de la fila 3 y la columna 8, así que el LOB es 38H.

Por tanto,

0001 1000	00111000
HOB	LOB

Así que el valor Hexadecimal del número 6200 es 1838H.



APENDICE E

TABLA DE CONVERSION DE "COMPLEMENTO A 2'S" A "DECIMAL "

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	-128	-127	-126	-125	-124	-123	-122	-121	-120	-119	-118	-117	-116	-115	-114	-113
9	-112	-111	-110	-109	-108	-107	-106	-105	-104	-103	-102	-101	-100	-99	-98	-97
A	-96	-95	-94	-93	-92	-91	-90	-89	-88	-87	-86	-85	-84	-83	-82	-81
B	-80	-79	-78	-77	-76	-75	-74	-73	-72	-71	-70	-69	-68	-67	-66	-65
C	-64	-63	-62	-61	-60	-59	-58	-57	-56	-55	-54	-53	-52	-51	-50	-49
D	-48	-47	-46	-45	-44	-43	-42	-41	-40	-39	-38	-37	-36	-35	-34	-33
E	-32	-31	-30	-29	-28	-27	-26	-25	-24	-23	-22	-21	-20	-19	-18	-17
F	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1



APENDICE F

TABLA DE LA ADICION EN HEXADECIMAL

+																																
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

APENDICE G

TABLA RESUMEN DE OPERACIONES CON BANDERINES

INSTRUCCION	C	Z	P/V	S	N	H	COMENTARIOS
ADC HL, SS	#	#	V	#	0	X	Suma de 16 bits con acarreo
ADX s; ADD s	#	#	V	#	0	#	Suma de 8 bits sin o con acarreo
ADD DD, SS	#	—	—	—	0	X	Suma 16 bits
AND s	0	#	P	#	0	1	Operaciones lógicas
BIT b, s	—	#	X	X	0	1	El estado del bit b de la celdilla s es copiado en el banderín Z
CCF	#	—	—	—	0	X	Complementar el acarreo
CPD; CPDR; CPI; CPIR	—	#	#	X	1	X	Instrucción de búsqueda de bloques Z = 1 si A = (HL) si no Z = 0 P/V = 1 si BC ≠ 0, si no P/V = 0
CP s	#	#	V	#	1	#	Comparar acumulador
CPL	—	—	—	—	1	1	Complementar acumulador
DAA	#	#	P	#	—	#	Ajuste decimal acumulador
DEC s	—	#	V	#	1	#	Decrementar 8 bits
IN r, (C)	—	#	P	#	0	0	Entrada registro indirecto
INC s	—	#	V	#	0	#	Incrementar 8 bits
IND; INI	—	#	X	X	1	X	Entrada de bloques Z = 0 si B ≠ 0, si no Z = 1
INDR:INIR	—	1	X	X	1	X	Entrada de bloques Z = 0 si B = 0 si no Z = 1
LD A,I; LD A,R	—	#	IFF	#	0	0	El contenido del biestable de interrupciones se copia en el banderín P/V
LDD; LDI	—	X	#	X	0	0	Instrucciones de transferencia de bloques
LDDR; LDIR	—	X	0	X	0	0	P/V = 1 si BC ≠ 0, si no P/V = 0
NEG	#	#	V	#	1	#	Negar acumulador
OR s	0	#	P	#	0	0	O lógico acumulador
OTDR; OTIR	—	1	X	X	1	X	Salida de bloques Z = 0 si B ≠ 0 si no Z = 1
OUTD; OUTI	—	#	X	X	1	X	Salida de bloques Z = 0 si B ≠ 0 si no Z = 1
RLA; RLCA; RRA; RRCA	#	—	—	—	0	0	Rotar acumulador
RLD; RRD	—	#	P	#	0	/	Rotar dígitos izquierda y derecha
RLS; RLC s; RR s; RRC s SLA s; SRA s; SRL s	#	#	P	#	0	0	Rotar y desplazar celdilla s
SBC HL, SS	#	#	V	#	1	X	Restar 16 bits con acarreo
SCF	1	—	—	—	0	0	Alzar acarreo
SBC s; SUB s	—	—	V	—	1	—	Restar 8 bits con acarreo
XOR x	0	—	P	—	0	0	O exclusivo acumulador



APENDICE G

SÍMBOLO	OPERACION
C	Banderín de acarreo. C=1 si la operación produjo un acarreo desde el bit más significativo del operando o del resultado.
Z	Banderín de cero. Z=1 si el resultado de la operación es cero.
S	Banderín de signo. S=1 si el bit más significativo del resultado es 1; ie. un número negativo.
P/V	Banderín de paridad o rebose. Paridad (P) y rebose (V) comparten el mismo banderín. Las operaciones lógicas afectan a este banderín por la paridad del resultado, mientras las operaciones aritméticas afectan a este banderín con el rebose del resultado. Si P/V indica paridad, P/V=1 si el resultado de la operación es par. P/V=0 si el resultado de la operación es impar. Si P/V indica rebose, P/V=1 si el resultado de la operación produjo un rebose.
H	Banderín de semi-acarreo. H=1 si la operación de suma o resta produjo un acarreo (o un préstamo) desde el bit 4 del acumulador.
N	Banderín de suma/resta. N=1 si la operación previa fue una resta. Los banderines H y N se usan juntamente con la instrucción de ajuste decimal (DAA) para corregir adecuadamente el resultado en el formato BCD después de efectuar una suma o una resta con operandos que estuvieran en ese formato.
#	El banderín queda afectado por el resultado de la operación.
-	El banderín no cambia por la operación.
O	El banderín se baja (=0) por la operación.
I	El banderín se alza (=1) por la operación.
X	Se desconoce el efecto sobre el banderín.
V	El banderín P/V queda afectado según el rebose.
P	El banderín P/V queda afectado según la paridad.
r	Cualquiera de los registros A, B, C, D, E, H, L.
s	Cualquier celdilla de 8 bits en todos los modos de direccionamiento permitidos en esas instrucciones
SS	Cualquier pareja de celdillas (16 bits) en todos los modos de direccionamiento permitidos en esas instrucciones.
R	Registro "refresco" de la memoria.
n	valor con 8 bits (entre 0 y 255).
nn	valor con 16 bits (entre 0 y 65535).



APENDICE I

INSTRUCCIONES DEL MICROPROCESADOR Z80
-por orden alfabético de nemónimos.

HEXADECIMAL	MNEMONIC	HEXADECIMAL	MNEMONIC	HEXADECIMAL	MNEMONIC
00	NOP	4B	LD C,C	92	SUB D
01 XXXX	LD BC,NN	4C	LD C,D	93	SUB E
02	LD (BC),A	4D	LD C,E	94	SUB H
03	INC BC	4E	LD C,H	95	SUB L
04	INC B	4F	LD C,L	96	SUB (HL)
05	DEC B	50	LD C,(HL)	97	SUB A
06XX	LD B,N	51	LD C,A	98	SBC A,B
07	RLCA	52	LD D,B	99	SBC A,C
08	EX AF, AF'	53	LD D,C	9A	SBC A,D
09	ADD HL,BC	54	LD D,D	9B	SBC A,E
0A	LD A, (BC)	55	LD D,E	9C	SBC A,H
0B	DEC BC	56	LD D,H	9D	SBC A,L
0C	INC C	57	LD D,L	9E	SBC A,(HL)
0D	DEC C	58	LD D,(HL)	9F	SBC A,A
0EXX	LD C,N	59	LD D,A	A0	AND B
0F	RRCA	5A	LD D,B	A1	AND C
10XX	DJNZ DIS	5B	LD D,C	A2	AND D
11XXXX	LD DE,NN	5C	LD D,D	A3	AND E
12	LD (DE),A	5D	LD D,E	A4	AND H
13	INC DE	5E	LD D,H	A5	AND L
14	INC D	5F	LD D,L	A6	AND (HL)
15	DEC D	60	LD E,(HL)	A7	AND A
16XX	LD D,N	61	LD E,A	A8	XOR B
17	RLA	62	LD E,B	A9	XOR C
18XX	JR DIS	63	LD E,C	AA	SOR D
19	ADD HL,DE	64	LD E,D	AB	XOR E
1A	LD A,(DE)	65	LD E,E	AC	SOR H
1B	DEC DE	66	LD E,H	AD	SOR L
1C	INC E	67	LD H,L	AE	XOR (HL)
1D	DEC E	68	LD H,(HL)	AF	XOR A
1EXX	LD E,N	69	LD L,B	B0	OR B
1F	RRD	6A	LD L,C	B1	OR C
20XX	JR NZ,DIS	6B	LD L,D	B2	OR D
21XXXX	LD HL,NN	6C	LD L,H	B3	OR E
22XXXX	LD (NN),HL	6D	LD L,L	B4	OR H
23	INC HL	6E	LD L,(HL)	B5	OR L
24	INC H	6F	LD L,A	B6	OR (HL)
25	DEC H	70	LD (HL),B	B7	OR A
26XX	LD H,N	71	LD (HL),C	B8	CP B
27	DAA	72	LD (HL),D	B9	CP C
28XX	JR Z,DIS	73	LD (HL),E	BA	CP D
29	ADD HL,HL	74	LD (HL),H	BB	CP E
2AXXXX	LD HL,(NN)	75	LD (HL),L	BC	CP H
2B	DEC HL	76	HALT	BD	CP L
2C	INC L	77	LD (HL),A	BE	CP (HL)
2D	DEC L	78	LD A,B	BF	CP A
2EXX	LD L,N	79	LD A,C	C0	RET NZ
2F	CPL	7A	LD A,D	C1	POP BC
30XX	JR NC,DIS	7B	LD A,E	C2XXXX	JP NZ,NM
31XXXX	LD SP,NN	7C	LD A,H	C3XXXX	JP NN
32XXXX	LD (NN),A	7D	LD A,(HL)	C4XXXX	CALL NZ,NM
33	INC SP	7E	LD A,A	C5	PUSH BC
34	INC (HL)	7F	LD A,(HL)	C6XX	ADD A,N
35	DEC (HL)	80	ADD A,B	C7	RST 0
3620XX	LD (HL),N	81	ADD A,C	C8	RET Z
37	SCF	82	ADD A,D	C9	RET
38XX	JR C,DIS	83	ADD A,E	CAXXXX	JP Z,NM
39	ADD HL,SP	84	ADD A,H	CBXXXX	CALL Z,NN
3AXXXX	LD A,(NN)	85	ADD A,L	CCXXXX	CALL NN
3B	DEC SP	86	ADD A,(HL)	CDXXXX	ADC A,N
3C	INC A	87	ADD A,A	CEXX	RST B
3D	DEC A	88	ADC A,B	D0	RET NC
3EXXXX	LD A	89	ADC A,C	D1	POP DE
3F	CCF	8A	ADC A,D	D2XXXX	JP NC,NM
40	LD B,B	8B	ADC A,E	D3XX	OUT (N),A
41	LD B,C	8C	ADC A,H	D4XXXX	CALL NC,NN
42	LD B,D	8D	ADC A,L	D5	PUSH DE
43	LD B,E	8E	ADC A,(HL)	D6XX	SUB N
44	LD B,HL	8F	SUB B	D7	RST 10H
45	LD B,L	90	SUB C	D8	RET C
46	LD B,(HL)	91		D9	EXX
47	LD B,A			DAXXXX	JP C,NN
48	LD C,B			DBXX	IN A,(N)

HEXADECIMAL	MNEMONIC	HEXADECIMAL	MNEMONIC	HEXADECIMAL	MNEMONIC
DCXXXX	CALL C,NN	CB28	SRA B	CB79	BIT 7,C
DEXX	SBC A,N	CB29	SRA C	CB7A	BIT 7,D
DF	RST 18H	CB2A	SRA D	CB7B	BIT 7,E
E0	RET P0	CB2B	SRA E	CB7C	BIT 7,H
E1	POP HL	CB2C	SRA H	CB7D	BIT 7,L
E2XXXX	JP PO,NN	CB2D	SRA L	CB7E	BIT 7,(HL)
E3	EX (SP),HL	CB2E	SRA (HL)	CB7F	BIT 7,A
E4XXXX	CALL PO,NN	CB2F	SRA A	CB80	RES 0,B
E5	PUSH HL	CB38	SRL B	CB81	RES 0,C
E6XX	AND N	CB39	SRL C	CB82	RES 0,D
E7	RST 20 H	CB3A	SRL D	CB83	RES 0,E
E8	RET PE	CB3B	SRL E	CB84	RES 0,H
E9	JP (HL)	CB3C	SRL H	CB85	RES 0,L
EAXXXX	JE PE,NN	CB3D	SRL L	CB86	RES 0,(HL)
EB	EX DE,HL	CB3E	SRL (HL)	CB87	RES 0,A
ECXXXX	CALL PE,NN	CB3F	SRL A	CB88	RES 1,B
EDXX	XOR N	CB40	BIT 0,B	CB89	RES 1,C
EF	RST 28H	CB41	BIT 0,C	CB8A	RES 1,D
F0	RET P	CB42	BIT 0,D	CB8B	RES 1,E
F1	POP AF	CB43	BIT 0,E	CB8C	RES 1,H
F2XXXX	JP P,NN	CB44	BIT 0,H	CB8D	RES 1,L
F3	D1	CB45	BIT 0,L	CB8E	RES 1,(HL)
F4XXXX	CALL P,NN	CB46	BIT 0,(HL)	CB8F	RES 1,A
F5	PUSH AF	CB47	BIT 0,A	CB90	RES 2,B
F620XX	OR N	CB48	BIT 1,B	CB91	RES 2,C
F7	RST 30H	CB49	BIT 1,C	CB92	RES 2,D
F8	RET N	CB4A	BIT 1,D	CB93	RES 2,E
F9	LD,SP,HL	CB4B	BIT 1,E	CB94	RES 2,H
FAXXXX	JP N,NN	CB4C	BIT 1,H	CB95	RES 2,L
FB	E1	CB4D	BIT 1,L	CB96	RES 2,(HL)
FCXXXX	CALL M,NN	CB4E	BIT 1,(HL)	CB97	RES 2,A
FE20XX	CP N	CB4F	BIT 1,A	CB98	RES 3,B
FF	RST 38H	CB50	BIT 2,B	CB99	RES 3,C
CB00	RLC 8	CB51	BIT 2,C	CB9A	RES 3,D
CB01	RLC C	CB52	BIT 2,D	CB9B	RES 3,E
CB02	RLC D	CB53	BIT 2,E	CB9C	RES 3,H
CB03	RLC E	CB54	BIT 2,H	CB9D	RES 3,L
CB04	RLC H	CB55	BIT 2,L	CB9E	RES 3,(HL)
CB05	RLC L	CB56	BIT 2,(HL)	CB9F	RES 3,A
CB06	RLC (HL)	CB57	BIT 2,A	CB9A	RES 4,B
CB07	RLC A	CB58	BIT 3,B	CB9B	RES 4,C
CB08	RRC B	CB59	BIT 3,C	CB9C	RES 4,D
CB09	RRC C	CB5A	BIT 3,D	CB9D	RES 4,E
CB0A	RRC D	CB5B	BIT 3,E	CB9E	RES 4,H
CB0B	RRC E	CB5C	BIT 3,H	CB9F	RES 4,L
CB0C	RRC H	CB5D	BIT 3,L	CB9A	RES 4,(HL)
CB0D	RRC L	CB5E	BIT 3,(HL)	CB9B	RES 4,A
CB0E	RRC (HL)	CB5F	BIT 3,A	CB9C	RES 4,B
CB0F	RRC A	CB60	BIT 4,B	CB9D	RES 4,C
CB10	RL B	CB61	BIT 4,C	CB9E	RES 4,D
CB11	RL C	CB62	BIT 4,D	CB9F	RES 4,E
CB12	RL D	CB63	BIT 4,E	CB9A	RES 4,H
CB13	RL E	CB64	BIT 4,H	CB9B	RES 4,L
CB14	RL H	CB65	BIT 4,L	CB9C	RES 4,(HL)
CB15	RL L	CB66	BIT 4,(HL)	CB9D	RES 4,A
CB16	RL (HL)	CB67	BIT 4,A	CB9E	RES 4,B
CB17	RL A	CB68	BIT 5,B	CB9F	RES 4,C
CB18	RR B	CB69	BIT 5,C	CB9A	RES 4,D
CB19	RR C	CB6A	BIT 5,D	CB9B	RES 4,E
CB1A	RR D	CB6B	BIT 5,E	CB9C	RES 4,H
CB1B	RR E	CB6C	BIT 5,H	CB9D	RES 4,L
CB1C	RR H	CB6D	BIT 5,L	CB9E	RES 4,(HL)
CB1D	RR L	CB6E	BIT 5,(HL)	CB9F	RES 4,A
CB1E	RR (HL)	CB6F	BIT 5,A	CB9A	RES 4,B
CB1F	RR A	CB70	BIT 6,B	CB9B	RES 4,C
CB20	SLA B	CB71	BIT 6,C	CB9C	RES 4,D
CB21	SLA C	CB72	BIT 6,D	CB9D	RES 4,E
CB22	SLA D	CB73	BIT 6,E	CB9E	RES 4,H
CB23	SLA E	CB74	BIT 6,H	CB9F	RES 4,L
CB24	SLA H	CB75	BIT 6,L	CB9A	RES 4,(HL)
CB25	SLA L	CB76	BIT 6,(HL)	CB9B	RES 4,A
CB26	SLA (HL)	CB77	BIT 6,A	CB9C	RES 4,B
CB27	SLA A	CB78	BIT 7,B	CB9D	RES 4,C
				CB9E	RES 4,D
				CB9F	RES 4,E
				CB9A	RES 4,H
				CB9B	RES 4,L
				CB9C	RES 4,(HL)
				CB9D	RES 4,A
				CB9E	RES 4,B
				CB9F	RES 4,C
				CB9A	RES 4,D
				CB9B	RES 4,E
				CB9C	RES 4,H
				CB9D	RES 4,L
				CB9E	RES 4,(HL)
				CB9F	RES 4,A
				CB9A	RES 4,B
				CB9B	RES 4,C
				CB9C	RES 4,D
				CB9D	RES 4,E
				CB9E	RES 4,H
				CB9F	RES 4,L
				CB9A	RES 4,(HL)
				CB9B	RES 4,A
				CB9C	RES 4,B
				CB9D	RES 4,C
				CB9E	RES 4,D
				CB9F	RES 4,E
				CB9A	RES 4,H
				CB9B	RES 4,L
				CB9C	RES 4,(HL)
				CB9D	RES 4,A
				CB9E	RES 4,B
				CB9F	RES 4,C
				CB9A	RES 4,D
				CB9B	RES 4,E
				CB9C	RES 4,H
				CB9D	RES 4,L
				CB9E	RES 4,(HL)
				CB9F	RES 4,A
				CB9A	RES 4,B
				CB9B	RES 4,C
				CB9C	RES 4,D
				CB9D	RES 4,E
				CB9E	RES 4,H
				CB9F	RES 4,L
				CB9A	RES 4,(HL)
				CB9B	RES 4,A
				CB9C	RES 4,B
				CB9D	RES 4,C
				CB9E	RES 4,D
				CB9F	RES 4,E
				CB9A	RES 4,H
				CB9B	RES 4,L
				CB9C	RES 4,(HL)
				CB9D	RES 4,A
				CB9E	RES 4,B
				CB9F	RES 4,C
				CB9A	RES 4,D
				CB9B	RES 4,E
				CB9C	RES 4,H
				CB9D	RES 4,L
				CB9E	RES 4,(HL)
				CB9F	RES 4,A
				CB9A	RES 4,B
				CB9B	RES 4,C
				CB9C	RES 4,D
				CB9D	RES 4,E
				CB9E	RES 4,H
				CB9F	RES 4,L
				CB9A	RES 4,(HL)
				CB9B	RES 4,A
				CB9C	RES 4,B
				CB9D	RES 4,C
				CB9E	RES 4,D
				CB9F	RES 4,E
				CB9A	RES 4,H
				CB9B	RES 4,L
				CB9C	RES 4,(HL)
				CB9D	RES 4,A
				CB9E	RES 4,B
				CB9F	RES 4,C
				CB9A	RES 4,D
				CB9B	RES 4,E
				CB9C	RES 4,H
				CB9D	RES 4,L
				CB9E	RES 4,(HL)
				CB9F	RES 4,A
				CB9A	RES 4,B
				CB9B	RES 4,C
				CB9C	RES 4,D
				CB9D	RES 4,E
				CB9E	RES 4,H
				CB9F	RES 4,L
				CB9A	RES 4,(HL)
				CB9B	RES 4,A
				CB9C	RES 4,B
				CB9D	RES 4,C
				CB9E	RES 4,D
				CB9F	RES 4,E
				CB9A	RES 4,H
				CB9B	RES 4,L
				CB9C	RES 4,(HL)
				CB9D	RES 4,A
				CB9E	RES 4,B
				CB9F	RES 4,C
				CB9A	RES 4,D
				CB9B	RES 4,E
				CB9C	RES 4,H
				CB9D	RES 4,L
				CB9E	RES 4,(HL)
				CB9F	RES 4,A
				CB9A	RES 4,B
				CB9B	RES 4,C
				CB9C	RES 4,D
				CB9D	RES 4,E
				CB9E	RES 4,H
				CB9F	RES 4,L
				CB9A	RES 4,(HL)
				CB9B	RES 4,A
				CB9C	RES 4,B
				CB9D	RES 4,C
				CB9E	RES 4,D
				CB9F	RES 4,E
				CB9A	RES 4,H
				CB9B	RES 4,L
				CB9C	RES 4,(HL)
				CB9D	RES 4,A
				CB9E	RES 4,B
				CB9F	RES 4,C
				CB9A	RES 4,D
				CB9B	RES 4,E
				CB9C	RES 4,H
				CB9D	RES 4,L
				CB9E	RES 4,(HL)
				CB9F	RES 4,A
				CB9A	RES 4,B
				CB9B	RES 4,C
				CB9C	RES 4,D
				CB9D	RES 4,E
				CB9E	RES 4,H
				CB9F	RES 4,L
				CB9A	RES 4,(HL)
				CB9B	RES 4,A
				CB9C	RES 4,B
				CB9D	RES 4,C
				CB9E	RES 4,D
				CB9F	RES 4,E
				CB9A	RES 4,H
				CB9B	RES 4,L
				CB9C	RES 4,(HL)
				CB9D	RES 4,A
				CB9E	RES 4,B
				CB9F	RES 4,C
				CB9A	RES 4,D
				CB9B	RES 4,E
				CB9C	RES 4,H
				CB9D	RES 4,L
				CB9E	RES 4,(HL)
				CB9F	RES 4,A
				CB9A	RES 4,B
				CB9B	RES 4,C
				CB9C	RES 4,D
				CB9D	RES 4,E
				CB9E	RES 4,H
				CB9F	RES 4,L
				CB9A	RES 4,(HL)
				CB9B	RES 4,A
				CB9C	RES 4,B
				CB9D	RES 4,C
				CB9E	RES 4,D
				CB9F	RES 4,E
				CB9A	RES 4,H
				CB9B	RES 4,L
				CB9C	RES 4,(HL)
				CB9D	RES 4,A
				CB9E	RES 4,B
				CB9F	RES 4,C
				CB9A	RES 4,D
				CB9B	RES 4,E
				CB9C	RES 4,H
				CB9D	RES 4,L
				CB9E	RES 4,(HL)
				CB9F	RES 4,A
				CB9A	RES 4,B
				CB9B	RES 4,C
				CB9C	RES 4,D
				CB9D	RES 4,E
				CB9E	RES 4,H
				CB9F	RES 4,L
				CB9A	RES 4,(HL)
				CB9B	RES 4,A
				CB9C	RES 4,B
				CB9D	RES 4,C
				CB9E	RES 4,D
				CB9F	RES 4,E
				CB9A	RES 4,H
				CB9B	RES 4,L
				CB9C	RES 4,(HL)
				CB9D	RES 4,A
				CB9E	RES 4,B
				CB9F	RES 4,C
				CB9A	RES 4,D
				CB9B	RES 4,E
				CB9C	RES 4,H
				CB9D	RES 4,L
				CB9E	RES 4,(HL)
				CB9F	RES 4,A
				CB9A	RES 4,B
				CB9B	RES 4,C
				CB9C	RES 4,D
				CB9D	RES 4,E
				CB9E	RES 4,H
				CB9F	RES 4,L
				CB9A	RES 4,(HL)
				CB9B	RES 4,A
				CB9C	RES 4,B
				CB9D	RES 4,C
				CB9E	RES 4,D
				CB9F	RES 4,E
				CB9A	RES 4,H

HEXADECIMAL	MNEMONIC	HEXADECIMAL	MNEMONIC	HEXADECIMAL	MNEMONIC
CBC3	SET 0,E	DD4EXX	LD C,(IX+)	ED56	IN 1
CBC4	SET 0,H	DD5EXX	LD D,(IX+)	ED57	LD A,1
CBC5	SET 0,L	DD6EXX	LD E,(IX+)	ED58	IN E,(C)
CBC6	SET 0,(HL)	DD7EXX	LD H,(IX+)	ED59	OUT (C),E
CBC7	SET 0,A	DD8EXX	LD L,(IX+)	ED5A	ADC HL,DE
CBC8	SET 1,B	DD70XX	LD (IX+),B	ED5BXXXX	LD DE,(NN)
CBC9	SET 1,C	DD71XX	LD (IC+),C	ED5C	IM 2
CBCA	SET 1,D	DD72XX	LD (IX+),D	ED60	IN H,(C)
CBCB	SET 1,E	DD73XX	LD (IX+),E	ED61	OUT (C),H
CBCD	SET 1,H	DD74XX	LD (IX+),H	ED62	SBC HL,HL
CBCD	SET 1,L	DD75XX	LD (IX+),L	ED67	RRO
CBCF	SET 1,(HL)	DD77XX	LD (IX+),A	ED68	IN L,(C)
CBCE	SET 1,A	DD7EXX	LD A,(IX+)	ED69	OUT (C),L
CBDD	SET 2,B	DD8EXX	ADD A,(IX+)	ED6A	ADC HL,HL
CBDD	SET 2,C	DD8EXX	ADC A,(IX+)	ED6F	RLO
CBDD	SET 2,D	DD8EXX	SUB (IX+)	ED72	SBC HL,SP
CBDD	SET 2,E	DD8EXX	SBC A,(IX+)	ED73XXXX	LD (NN),SP
CBDD	SET 2,H	DD8EXX	AND (IX+)	ED78	IN A,(C)
CBDD	SET 2,L	DD8EXX	XOR (IX+)	ED79	OUT (C),A
CBDD	SET 2,(HL)	DD8EXX	OR (IX+)	ED7A	ADC HL,SP
CBDD	SET 2,A	DD8EXX	CPI (IX+)	ED7BXXXX	LD SP,(NN)
CBDD	SET 3,B	DD8EXX	POP IX	ED80	LDI
CBDD	SET 3,C	DD8EXX	EX (SP),IX	EDA1	CPI
CBDD	SET 3,D	DD8EXX	PUSH IX	EDA2	INI
CBDD	SET 3,E	DD8EXX	JP (IX)	EDA3	OUTI
CBDD	SET 3,H	DD8EXX	LD SP,IX	EDA8	LDD
CBDD	SET 3,L	DD8EXX06	RLC (IX+)	EDA9	CP0
CBDD	SET 3,(HL)	DD8EXX0E	RRC (IX+)	EDA9	IND
CBDD	SET 3,A	DD8EXX16	RL (IX+)	EDAB	OUTD
CBDD	SET 4,B	DD8EXX1E	RR (IX+)	EDB0	LDIR
CBDD	SET 4,C	DD8EXX26	SLA (IX+)	EDB1	CPIR
CBDD	SET 4,D	DD8EXX2E	SRA (IX+)	EDB2	INIR
CBDD	SET 4,E	DD8EXX3E	SRL (IX+)	EDB3	OTIR
CBDD	SET 4,H	DD8EXX46	BIT 0,(IX+)	EDB8	LDOR
CBDD	SET 4,L	DD8EXX4E	BIT 1,(IX+)	EDB9	CPDR
CBDD	SET 4,(HL)	DD8EXX56	BIT 2,(IX+)	EDBA	INDR
CBDD	SET 4,A	DD8EXX5E	BIT 3,(IX+)	EDB9	OTDR
CBDD	SET 5,B	DD8EXX66	BIT 4,(IX+)	ED09	ADD IV,BC
CBDD	SET 5,C	DD8EXX6E	BIT 5,(IX+)	ED19	ADD IV,DC
CBDD	SET 5,D	DD8EXX76	BIT 6,(IX+)	ED21XXXX	LD IV,NN
CBDD	SET 5,E	DD8EXX7E	BIT 7,(IX+)	ED22XXXX	LD (NN),IV
CBDD	SET 5,H	DD8EXX86	RES 0,(IX+)	FD23	INC IV
CBDD	SET 5,L	DD8EXX8E	RES 1,(IX+)	FD29	ADD IV,IY
CBDD	SET 5,(HL)	DD8EXX96	RES 2,(IX+)	FD2AXXXX	LD IV,(NN)
CBDD	SET 5,A	DD8EXX9E	RES 3,(IX+)	FD2B	DEC IV
CBDD	SET 6,B	DD8EXXA6	RES 4,(IX+)	FD34XX	INC (IY+)
CBDD	SET 6,C	DD8EXXAE	RES 5,(IX+)	FD35XX	DEC (IY+)
CBDD	SET 6,D	DD8EXXB6	RES 6,(IX+)	FD36XX20	LD (IY+),N
CBDD	SET 6,E	DD8EXXB6	RES 7,(IX+)	FD38	ADD IV,SP
CBDD	SET 6,H	DD8EXXC6	SET 0,(IX+)	FD46XX	LD B,(IY+)
CBDD	SET 6,(HL)	DD8EXXCE	SET 1,(IX+)	FD3EXX	LD C,(IY+)
CBDD	SET 6,A	DD8EXXD6	SET 2,(IX+)	FD56XX	LD D,(IY+)
CBDD	SET 7,B	DD8EXXE6	SET 3,(IX+)	FD5EXX	LD E,(IY+)
CBDD	SET 7,C	DD8EXXE6	SET 4,(IX+)	FD66XX	LD H,(IY+)
CBDD	SET 7,D	DD8EXXE6	SET 5,(IX+)	FD6EXX	LD L,(IY+)
CBDD	SET 7,E	DD8EXXFE	SET 6,(IX+)	FD70XX	LD (IY+),B
CBDD	SET 7,H	DD8EXXFE	SET 7,(IX+)	FD71XX	LD (IY+),C
CBDD	SET 7,L	ED41	IN B,(C)	FD72XX	LD (IY+),D
CBDD	SET 7,(HL)	ED42	SBC HL,BC	FD73XX	LD (IY+),E
CBDD	SET 7,A	ED43XXXX	LD (NN),BC	FD74XX	LD (IY+),H
DD09	ADD IX,BC	ED44	NEG	FD75XX	LD (IY+),L
DD19	ADD IX,DE	ED45	RETN	FD77XX	LD (IY+),A
DD21XXXX	LD IX,NN	ED46	IM 0	FD7EXX	LD A,(IY+)
DD22XXXX	LD (NN),IX	ED47	LD 1,A	FD86XX	ADD A,(IY+)
DD23	INC IX	ED48	IN C,(C)	FD8EXX	ADC A,(IY+)
DD29	ADD IX,IX	ED49	OUT (C),C	FD96XX	SUB (IY+)
DD2AXXXX	LD IX,(NN)	ED4A	ADC HL,BC	FD9EXX	SBC A,(IY+)
DD2B	DEC IX	ED4BXXXX	LD BC,(NN)	FDAXXX	AND (IY+)
DD34XX	INC (IX+)	ED4D	RET	FD86XX	XOR (IY+)
DD35XX	DEC (IX+)	ED50	IN D,(C)	FD8EXX	OP (IY+)
DD36XX20	LD (IX+),N	ED51	OUT (C),D	FDE1	POP IY
DD39	ADD IX,SP	ED52	SBC HL,DE	FDE3	EX (SP), IY
DD46XX	LD B,(IX+)	ED53XXXX	LD (NN),DE		



HEXADECIMAL	MNEMONIC	HEXADECIMAL	MNEMONIC	HEXADECIMAL	MNEMONIC
FDE5	PUSH IY				
FDE9	JP (IY)				
FD99	L ¹ SP, IY				
FDCBXX06	HLC(IY+di)				
FDCBXX0E	RRC(IY+di)				
FDCBXX16	RL(IY+di)				
FDCBXX1E	RR(IY+di)				
FDCBXX26	S ¹ A(IY+di)				
FDCBXX2E	SRA(IY+di)				
FDCBXX3E	SRL(IY+di)				
FDCBXX46	BIT 0, (IY+di)				
FDCBXX4E	BIT 1, (IY+di)				
FDCBXX56	BIT 2, (IY+di)				
FDCBXX5E	BIT 3, (IY+di)				
FDCBXX66	BIT 4, (IY+di)				
FDCBXX6E	BIT 5, (IY+di)				
FDCBXX76	BIT 6, (IY+di)				
FDCBXX7E	BIT 7, (IY+di)				
FDCBXX86	RES 0, (IY+di)				
FDCBXX8E	RES 1, (IY+di)				
FDCBXX96	RES 2, (IY+di)				
FDCBXX9E	RES 3, (IY+di)				
FDCBXXA6	RES 4, (IY+di)				
FDCBXXAE	RES 5, (IY+di)				
FDCBXXB6	RES 6, (IY+di)				
FDCBXXBE	RES 7, (IY+di)				
FDCBXXC6	SET 0, (IY+di)				
FDCBXXCE	SET 1, (IY+di)				
FDCBXXD6	SET 2, (IY+di)				
FDCBXXDE	SET 3, (IY+di)				
FDCBXXE6	SET 4, (IY+di)				
FDCBXXEE	SET 5, (IY+di)				
FDCBXXF6	SET 6, (IY+di)				
FDCBXXFE	SET 7, (IY+di)				



APENDICE H

INSTRUCCIONES DEL MICROPROCESADOR Z80

-ordenadas por código de operación.

MNEMONIC	HEXADecimal	MNEMONIC	HEXADecimal	MNEMONIC	HEXADecimal
ADC A, (HL)	8E	BIT 2,B	CB 50	CP n	FE XX
ADC A, (IX+dis)	DD 8E XX	BIT 2,C	CB 51	CP E	85
ADC A, (IY+dis)	FD 8E XX	BIT 2,D	CB 52	CP H	BC
ADC A,A	8F	BIT 2,E	CB 53	CP L	BD
ADC A,B	88	BIT 2,H	CB 54	CPD	ED A9
ADC A,C	89	BIT 2,L	CB 55	CPDR	ED 89
ADC A,D	8A	BIT 3,(HL)	CB 5E	CPI	ED A1
ADC A,n	CE XX	BIT 3,(IX+dis)	DD CB XX 5E	CPIR	ED B1
ADC A,E	8B	BIT 3,(IY+dis)	FD CB XX 5E	CPL	2F
ADC A,H	8C	BIT 3,A	CB 5F	DAA	27
ADC A,L	8D	BIT 3,B	CB 58	DEC (HL)	35
ADC HL,BC	ED 4A	BIT 3,C	CB 59	DEC (IX+dis)	DD 35 XX
ADC HL,DE	ED 5A	BIT 3,D	CB 5A	DEC (IY+dis)	FD 35 XX
ADC HL,HL	ED 6A	BIT 3,E	CB 5B	DEC A	3D
ADC HL,SP	ED 7A	BIT 3,H	CB 5C	DEC B	05
ADD A, (HL)	86	BIT 3,L	CB 5D	DEC BC	0B
ADD A, (IX+dis)	DD 86 XX	BIT 4,(HL)	CB 66	DEC C	0D
ADD A, (IY+dis)	FD 86 XX	BIT 4,(IX+dis)	DD CB XX 66	DEC D	15
ADD A,A	87	BIT 4,(IY+dis)	FD CB XX 66	DEC DE	1B
ADD A,B	80	BIT 4,A	CB 67	DEC E	1D
ADD A,C	81	BIT 4,B	CB 68	DEC H	25
ADD A,D	82	BIT 4,C	CB 69	DEC HL	2B
ADD A,n	CE XX	BIT 4,D	CB 6A	DEC IX	DD 2B
ADD A,E	83	BIT 4,E	CB 6B	DEC IY	FD 2B
ADD A,H	84	BIT 4,H	CB 6C	DEC L	2D
ADD A,L	85	BIT 4,L	CB 6D	DEC SP	3B
ADD HL,BC	09	BIT 5,(HL)	CB 6E	DI	76
ADD HL,DE	19	BIT 5,(IX+dis)	DD CB XX 6E	DIJ,dis	10 XX
ADD HL,HL	29	BIT 5,(IY+dis)	FD CB XX 6E	EI	FB
ADD HL,SP	39	BIT 5,A	CB 6F	EX (SP),HL	E3
ADD IX,BC	DD 09	BIT 5,B	CB 68	EX (SP),IX	DD E3
ADD IX,DE	DD 19	BIT 5,C	CB 69	EX (SP),IY	FD E3
ADD IX,HL	DD 29	BIT 5,D	CB 6A	EX AF,AF'	0B
ADD IX,SP	DD 39	BIT 5,E	CB 6B	EX DE,HL	E8
ADD IY,BC	FD 09	BIT 5,H	CB 6C	EXX	D9
ADD IY,DE	FD 19	BIT 5,L	CB 6D	HALT	76
ADD IY,IY	FD 29	BIT 6,(HL)	CB 76	IM 0	ED 46
ADD IY,SP	FD 39	BIT 6,(IX+dis)	DD CB XX 76	IM 1	ED 56
AND (HL)	A6	BIT 6,(IY+dis)	FD CB XX 76	IM 2	ED 5E
AND (IX+dis)	DD A6 XX	BIT 6,A	CB 77	IN A, (C)	ED 7B
AND (IY+dis)	FD A6 XX	BIT 6,B	CB 78	IN A, port	DB XX
AND A	A7	BIT 6,C	CB 79	IN B, (C)	ED 40
AND B	A0	BIT 6,D	CB 7A	IN C, (C)	ED 48
AND C	A1	BIT 6,E	CB 7B	IN D, (C)	ED 50
AND D	A2	BIT 6,H	CB 7C	IN E, (C)	ED 58
AND n	E6 XX	BIT 6,L	CB 7D	IN H, (C)	ED 60
AND E	A3	BIT 7,(HL)	CB 7E	IN L, (C)	ED 68
AND H	A4	BIT 7,(IX+dis)	DD CB XX 7E	INC (HL)	34
AND L	A5	BIT 7,(IY+dis)	FD CB XX 7E	INC (IX+dis)	DD 34 XX
BIT 0,(HL)	CB 46	BIT 7,A	CB 7F	INC (IY+dis)	FD 34 XX
BIT 0,(IX+dis)	DD CB XX 46	BIT 7,B	CB 78	INC A	3C
BIT 0,(IY+dis)	FD CB XX 46	BIT 7,C	CB 79	INC B	04
BIT 0,A	CB 47	BIT 7,D	CB 7A	INC BC	03
BIT 0,B	CB 40	BIT 7,E	CB 7B	INC C	0C
BIT 0,C	CB 41	BIT 7,H	CB 7C	INC D	14
BIT 0,D	CB 42	BIT 7,L	CB 7D	INC DE	1C
BIT 0,E	CB 43	CALL ADDR	DC XX XX	INC E	13
BIT 0,H	CB 44	CALL C,ADDR	DC XX XX	INC H	24
BIT 0,L	CB 45	CALL M,ADDR	FC XX XX	INC HL	23
BIT 1,(HL)	CB 4E	CALL NC,ADDR	D4 XX XX	INC IX	DD 23
BIT 1,(IX+dis)	DD CB XX 4E	CALL NZ,ADDR	C4 XX XX	INC IY	FD 23
BIT 1,(IY+dis)	FD CB XX 4E	CALL P,ADDR	F4 XX XX	INC L	2C
BIT 1,A	CB 4F	CALL PE,ADDR	EC XX XX	INC SP	33
BIT 1,B	CB 48	CALL PO,ADDR	E4 XX XX	IND	ED AA
BIT 1,C	CB 49	CALL Z,ADDR	CC XX XX	INDR	ED BA
BIT 1,D	CB 4A	CCF	3F	INI	ED A2
BIT 1,E	CB 4B	CP (HL)	8E	INIR	ED B2
BIT 1,H	CB 4C	CP (IX+dis)	DD BE XX	JP (HL)	E9
BIT 1,L	CB 4D	CP (IY+dis)	FD BE XX	JP (IX)	DD E9
BIT 2,(HL)	CB 56	CP A	BF	JP (IY)	FD E9
BIT 2,(IX+dis)	DD CB XX 56	CP B	88	JP ADDR	C3 XX XX
BIT 2,(IY+dis)	FD CB XX 56	CP C	89	JP C,ADDR	DA XX XX
BIT 2,A	CB 57	CP D	BA	JP M,ADDR	FA XX XX

MNEMONIC	HEXADECIMAL	MNEMONIC	HEXADECIMAL	MNEMONIC	HEXADECIMAL
JP NC, ADDR	D2 XX XX	LD BC, nn	01 XX XX	LDDR	ED B8
JP NZ, ADDR	C2 XX XX	LD C, (HL)	4E	LDI	ED A0
JP P, ADDR	F2 XX XX	LD C, (IX+dis)	DD 4E xx	LDIR	ED B0
JP PE, ADDR	5A XX XX	LD C, (IY+dis)	FD 4E XX	NEG	ED 44
JP PO, ADDR	E2 XX XX	LD C, A	4F	NOP	D0
JP Z, ADDR	CA XX XX	LD C, B	48	OR (HL)	B6
JR C, dis	38 XX	LD C, C	49	OR (IX+dis)	DD B6 XX
JR dis	18 XX	LD C, D	4A	OR (IY+dis)	FD B6 xx
JR NC, dis	30 XX	LD C, n	0E XX	OR B	B0
JR NZ, dis	20 XX	LD C, E	4B	OR C	B1
JR Z, dis	28 XX	LD C, H	4C	OR D	B2
LD (ADDR), A	32 XX XX	LD C, A	4D	OR n	F6 XX
LD (ADDR), BC	ED 43 XX XX	LD D, (HL)	56	OR H	B4
LD (ADDR), DE	ED 53 XX XX	LD D, (IX+dis)	DD 56 XX	OTDR	ED B8
LD (ADDR), HL	ED 63 XX XX	LD D, (IY+dis)	FD 56 XX	OTIR	ED B3
LD (ADDR), HL	22 XX XX	LD D, B	57	OUT (C), A	ED 79
LD (ADDR), IX	DD 22 XX XX	LD D, C	51	OUT (C), B	ED 41
LD (ADDR), IY	FD 22 XX XX	LD D, D	52	OUT (C), C	ED 49
LD (ADDR), SP	ED 73 XX XX	LD D, E	53	OUT (C), D	ED 51
LD (BC), A	02	LD D, H	54	OUT (C), E	ED 59
LD (DE), A	12	LD D, L	55	OUT (C), H	ED 61
LD (HL), A	77	LD DE, nn	11 XX XX	OUT (C), L	ED 69
LD (HL), B	70	LD DE, (ADDR)	ED 5B XX XX	OUTD	D3 port
LD (HL), C	71	LD E, (HL)	5E	OUTD part, A	ED A3
LD (HL), D	72	LD E, (IX+dis)	DD 5E XX	POP AF	F1
LD (HL), n	36 XX	LD E, (IY+dis)	FD 5E XX	POP BC	C1
LD (HL), E	73	LD E, A	5F	POP DE	D1
LD (HL), H	74	LD E, B	58	POP HL	E1
LD (IX+dis), A	DD 77 XX	LD E, C	59	POP IX	DD E1
LD (IX+dis), B	DD 70 XX	LD E, D	5A	POP IY	FD E1
LD (IX+dis), C	DD 71 XX	LD E, n	1E XX	PUSH AF	F5
LD (IX+dis), D	DD 72 XX	LD E, E	5B	PUSH BC	C5
LD (IX+dis), n	DD 36 XX XX	LD E, H	5C	PUSH DE	D5
LD (IX+dis), E	DD 73 XX	LD E, L	5D	PUSH HL	E5
LD (IX+dis), H	DD 74 XX	LD H, (HL)	66	PUSH IX	DD E5
LD (IX+dis), L	DD 75 XX	LD H, (IX+dis)	DD 66 XX	PUSH IY	FD E5
LD (IY+dis), A	FD 77 XX	LD H, A	67	RES 0, (HL)	DD CB XX 86
LD (IY+dis), B	FD 70 XX	LD H, B	60	RES 0, (IY+dis)	FD CB XX 86
LD (IY+dis), C	FD 71 XX	LD H, C	61	RES 0, A	C8 87
LD (IY+dis), D	FD 72 XX	LD H, D	62	RES 0, B	C8 88
LD (IY+dis), n	FD 36 XX XX	LD H, n	26 XX	RES 0, C	C8 81
LD (IY+dis), E	FD 73 XX	LD H, E	63	RES 0, D	C8 82
LD (IY+dis), H	FD 74 XX	LD H, H	64	RES 0, E	C8 83
LD (IY+dis), L	FD 75 XX	LD H, L	65	RES 0, H	C8 84
LD A, (ADDR)	3A XX XX	LD HL, (ADDR)	ED 68 XX XX	RES 0, L	C8 85
LD A, (BC)	0A	LD HL, nn	21 XX XX	RES 1, (HL)	C8 8E
LD A, (DE)	1A	LD IX, (ADDR)	DD 2A XX XX	RES 1, (IX+dis)	DD CB XX 8E
LD A, (HL)	7E	LD IX, nn	DD 21 XX XX	RES 1, (IY+dis)	FD CB XX 8E
LD A, (IX+dis)	DD 7E XX	LD IY, (ADDR)	FD 2A XX XX	RES 1, A	C8 8F
LD A, (IY+dis)	FD 7E XX	LD IY, nn	FD 21 XX XX	RES 1, B	C8 88
LD A, A	7F	LD A, A	6F	RES 1, C	C8 89
LD A, B	78	LD A, B	68	RES 1, D	C8 8A
LD A, C	79	LD A, C	69	RES 1, E	C8 8B
LD A, D	7A	LD A, D	6A	RES 1, H	C8 8C
LD A, n	3E XX	LD A, E	2E XX	RES 1, L	C8 8D
LD A, E	7B	LD A, H	6B	RES 2, (HL)	C8 86
LD A, H	7C	LD A, L	6C	RES 2, (IX+dis)	DD CB XX 96
LD A, I	ED 57	LD A, n	ED 4F	RES 2, (IY+dis)	FD CB XX 96
LD A, L	7D	LD SP, (ADDR)	ED 7B XX XX	RES 2, A	C8 87
LD A, R	ED 5F	LD SP, nn	31 XX XX	RES 2, B	C8 88
LD B, (HL)	46	LD SP, IX	F9	RES 2, C	C8 89
LD B, (IX+dis)	DD 46 XX	LD SP, IY	FD F9	RES 2, D	C8 8A
LD B, (IY+dis)	FD 46 XX	LD B, (HL)	6E	RES 2, E	C8 83
LD B, A	47	LD B, (IX+dis)	DD 6E XX	RES 2, H	C8 84
LD B, B	40	LD B, (IY+dis)	FD 6E XX	RES 2, L	C8 85
LD B, C	41	LD B, L	6D	RES 3, (HL)	C8 86
LD B, D	42	LD B, R	ED 4F	RES 3, (IX+dis)	DD CB XX 9E
LD B, n	06 XX	LD SP, (ADDR)	ED 7B XX XX	RES 3, (IY+dis)	FD CB XX 9E
LD B, E	43	LD SP, nn	31 XX XX	RES 3, A	C8 8F
LD B, H	44	LD SP, IX	F9		
LD B, L	45	LD SP, IY	FD F9		
LD BC, (ADDR)	ED 4B XX XX	LDD	ED A8		

MNEMONIC	HEXADecimal	MNEMONIC	HEXADecimal	MNEMONIC	HEXADecimal
RES 3,B	CB 98	RLC C	CB 01	SET 1,L	CB CD
RES 3,C	CB 99	RLC D	CB 02	SET 2,(HL)	CB D6
RES 3,D	CB 9A	RLC E	CB 03	SET 2,(IX+dis)	DD CB XX D6
RES 3,E	CB 9B	RLC H	CB 04	SET 2,(IY+dis)	FD CB XX D6
RES 3,H	CB 9C	RLC L	CB 05	SET 2,A	CB D7
RES 3,L	CB 9D	RLCA	07	SET 2,B	CB D8
RES 4,(HL)	CB A6	RLD	ED 6F	SET 2,C	CB D1
RES 4,(IX+dis)	DD CB XX A6	RR (HL)	CB 1E	SET 2,D	CB D2
RES 4,(IY+dis)	FD CB XX A6	RR (IX+dis)	DD CB XX 1E	SET 2,E	CB D3
RES 4,A	CB A7	RR (IY+dis)	FD CB XX 1E	SET 2,H	CB D4
RES 4,B	CB A0	RR A	CB 1F	SET 2,L	CB D5
RES 4,C	CB A1	RR B	CB 18	SET 3,(HL)	CB D6
RES 4,D	CB A2	RR C	CB 19	SET 3,(IX+dis)	DD CB XX DE
RES 4,E	CB A3	RR D	CB 1A	SET 3,(IY+dis)	FD CB XX DE
RES 4,H	CB A4	RR E	CB 1B	SET 3,A	CB D7
RES 4,L	CB A5	RR H	CB 1C	SET 3,B	CB D8
RES 5,(HL)	CB AE	RR L	CB 1D	SET 3,C	CB D9
RES 5,(IX+dis)	DD CB XX AE	RR A	1F	SET 3,D	CB DA
RES 5,(IY+dis)	FD CB XX AE	RR (HL)	CB 0E	SET 3,E	CB DB
RES 5,A	CB AF	RR (IX+dis)	DD CB XX 0E	SET 3,H	CB DC
RES 5,B	CB A8	RR (IY+dis)	FD CB XX 0E	SET 3,L	CB DD
RES 5,C	CB A9	RR C	CB 0F	SET 4,(HL)	CB DE
RES 5,D	CB AA	RR B	CB 08	SET 4,(IX+dis)	DD CB XX E6
RES 5,E	CB AB	RR C	CB 09	SET 4,(IY+dis)	FD CB XX E6
RES 5,H	CB AC	RR D	CB 0A	SET 4,A	CB E7
RES 5,L	CB AD	RR E	CB 0B	SET 4,B	CB E8
RES 6,(HL)	CB B6	RR H	CB 0C	SET 4,C	CB E1
RES 6,(IX+dis)	DD CB XX B6	RR L	CB 0D	SET 4,D	CB E2
RES 6,(IY+dis)	FD CB XX B6	RRCA	0F	SET 4,E	CB E3
RES 6,A	CB B7	RRD	ED 67	SET 4,H	CB E4
RES 6,B	CB B0	RST 00	C7	SET 4,L	CB E5
RES 6,C	CB B1	RST 08	0F	SET 5,(HL)	CB E6
RES 6,D	CB B2	RST 10	D7	SET 5,(IX+dis)	DD CB XX EE
RES 6,E	CB B3	RST 18	DF	SET 5,(IY+dis)	FD CB XX EE
RES 6,H	CB B4	RST 20	E7	SET 5,A	CB EF
RES 6,L	CB B5	RST 28	EF	SET 5,B	CB E8
RES 7,(HL)	CB BE	RST 30	F7	SET 5,C	CB E9
RES 7,(IX+dis)	DD CB XX BE	RST 38	FF	SET 5,D	CB EA
RES 7,(IY+dis)	FD CB XX BE	SBC A,(HL)	9E	SET 5,E	CB EB
RES 7,A	CB BF	SBC A,(IX+dis)	DD 9E XX	SET 5,H	CB EC
RES 7,B	CB B8	SBC A,(IY+dis)	FD 9E XX	SET 5,L	CB ED
RES 7,C	CB B9	SBC A,A	9F	SET 6,(HL)	CB FE
RES 7,D	CB BA	SBC A,B	98	SET 6,(IX+dis)	DD CB XX F6
RES 7,E	CB BB	SBC A,C	99	SET 6,(IY+dis)	FD CB XX F6
RES 7,H	CB BC	SBC A,D	9A	SET 6,A	CB F7
RES 7,L	CB BD	SBC A,n	DE XX	SET 6,B	CB F8
RET	C9	SBC A,E	9B	SET 6,C	CB F1
RET C	D8	SBC A,H	9C	SET 6,D	CB F2
RET M	F8	SBC A,L	9D	SET 6,E	CB F3
RET NC	D0	SBC HL,BC	ED 42	SET 6,H	CB F4
RET NZ	C0	SBC HL,DE	ED 52	SET 6,L	CB F5
RET P	F0	SBC HL,HL	ED 62	SET 7,(HL)	CB FE
RET PE	E8	SBC HL,SP	ED 72	SET 7,(IX+dis)	DD CB XX FE
RET PO	E0	SCF	37	SET 7,(IY+dis)	FD CB XX FE
RET Z	C8	SET 0,(HL)	CB C6	SET 7,A	CB FF
RET I	ED 4D	SET 0,(IX+dis)	DD CB XX C6	SET 7,B	CB F8
RET N	ED 45	SET 0,(IY+dis)	FD CB XX C6	SET 7,C	CB F9
RL (HL)	CB 16	SET 0,A	CB C7	SET 7,D	CB FA
RL (IX+dis)	DD CB XX 16	SET 0,B	CB C0	SET 7,E	CB FB
RL (IY+dis)	FD CB XX 16	SET 0,C	CB C1	SET 7,H	CB FC
RL A	CB 17	SET 0,D	CB C2	SET 7,L	CB FD
RL B	CB 10	SET 0,E	CB C3	SLA (HL)	CB 26
RL C	CB 11	SET 0,H	CB C4	SLA (IX+dis)	DD CB XX 26
RL D	CB 12	SET 0,L	CB C5	SLA (IY+dis)	FD CB XX 26
RL E	CB 13	SET 1,(HL)	CB CE	SLA A	CB 27
RL H	CB 14	SET 1,(IX+dis)	DD CB XX CE	SLA B	CB 20
RL L	CB 15	SET 1,(IY+dis)	FD CB XX CE	SLA C	CB 21
RLA	17	SET 1,A	CB CF	SLA D	CB 22
RLC (HL)	CB 06	SET 1,B	CB CB	SLA E	CB 23
RLC (IX+dis)	DD CB XX 06	SET 1,C	CB CA	SLA H	CB 24
RLC (IY+dis)	FD CB XX 06	SET 1,D	CB C9	SLA L	CB 25
RLC A	CB 07	SET 1,E	CB CB	SRA (HL)	CB 2E
RLC B	CB 00	SET 1,H	CB CC	SRA (IX+dis)	DD CB XX 2E

MNEMONIC	HEXADECIMAL	MNEMONIC	HEXADECIMAL	MNEMONIC	HEXADECIMAL
SRA (IY+dis)	FD CB XX 2E				
SRA A	CB 2F				
SRA B	CB 28				
SRA C	CB 29				
SRA D	CB 2A				
SRA E	CB 2B				
SRA H	CB 2C				
SRA L	CB 2D				
SRL (HL)	CB 3E				
SRL (IX+dis)	DD CB XX 3E				
SRL (IY+dis)	FD CB XX 3E				
SRL A	CB 3F				
SRL B	CB 38				
SRL C	CB 39				
SRL D	CB 3A				
SRL E	CB 3B				
SRL H	CB 3C				
SRL L	CB 3D				
SUB (HL)	96				
SUB (IX+dis)	DD 96 XX				
SUB (IY+dis)	FD 96 XX				
SUB A	97				
SUB B	90				
SUB C	91				
SUB D	92				
SUB E	D6 XX				
SUB n	93				
SUB H	94				
SUB L	95				
XOR (HL)	AE				
XOR (IX+dis)	DD AE XX				
XOR (IY+dis)	FD AE XX				
XOR A	AF				
XOR B	A9				
XOR C	A9				
XOR D	AA				
XOR n	EE XX				
XOR E	A8				
XOR H	AC				
XOR L	AD				



INDICE

Acumulador	34
Atributos en Spectrum	142
BCD	126
Banderines	58
-de acarreo	60
-de cero	59
-de negación	62
-de paridad/rebose	62
-de resta	62
-de semiacarreo	62
de signo	59
BASIC	5, 10, 39
BIT, SET/RESET	117
CALL	105
Complemento a dos	29
Contar	18
Contador de programa	31
Cotejo	162
DAA	126
Datos numéricos	25
Datos alfanuméricos	28
Desplazamientos	119
DI	127
Direccionamiento	49
EI	127
Ensamblador	8
Hexadecimal	19
conversión	23
cargador	155
IN	122
Interrupciones	127
Lazos	99
Lazos de espera	104
Limpiar	133
Modos de direccionamiento	49
Monitor de rutinas	145
Operaciones Booleanas	78
Operaciones con bloques	109
OUT	122
Nemónimos	9
Números con signo	26

Pantalla del Spectrum	138
Planteando tu programa	130
Pila	14
Puntero de la pila	36
Pastilla	5
Refrescar	37
Registro F	34
Registro I	37
Registro R	37
Registros .	38
alternativos	35, 115
indiciales	33
de instrucciones	31
Rotaciones	119
ROM	10
RST	128
Salto	99
absoluto	101
condicional	61
Sonido en el Spectrum	124
Teclado del Spectrum	135
Unidad Aritmeticológica	31
Unidad de control	31
Variables	14

