**EASYMBLER**

***ÍNDICE:***

**Pag. Capítulo: Observaciones:**

**0 00** [**INTRODUCCIÓN**](#INTRODUCCION) Por PipaGerardo.

**1 01** [**POR QUÉ METERME EN ESTE FREGADO**](#POR_QUE_METERME_EN_ESTE_FREGADO) Justificación de usar CM.

**1 02** [**¿QUÉ ES EL ENSAMBLADOR?**](#QUE_ES_EL_ENSAMBLADOR) Matrix???

**3 03** [**LOS NEMÓNICOS**](#LOS_NEMONICOS) Decodificando Matrix.

**4 04** [**EL SISTEMA BINARIO**](#EL_SISTEMA_BINARIO) Matemáticas 1.

**8 05** [**LAS OPERACIONES LÓGICAS**](#LAS_OPERACIONES_LOGICAS) Matemáticas 2.

**8 06** [**LA NOTACIÓN HEXADECIMAL**](#LA_NOTACION_HEXADECIMAL) Matemáticas 3.

**10 07** [**RESUMEN DE CONCEPTOS**](#RESUMEN_DE_CONCEPTOS) Absorber los capítulos anteriores.

**11 08** [**ESTRUCTURA DEL Z80**](#ESTRUCTURA_DEL_Z80) CPU, RAM, ALU, VPG, PGS, PPI, etc

**12 09** [**LOS REGISTROS DEL Z80**](#LOS_REGISTROS_DEL_Z80) Registros, Banderas, EX y EXX.

**15 10** [**LA PILA**](#LA_PILA) PUSH y POP.

**18 11** [**INSTRUCCIONES DEL Z80**](#INSTRUCCIONES_DEL_Z80) Rollo necesario no agobiarse.

**18 12** [**INSTRUCCIONES DE TRANSFERENCIA DE DATOS**](#INSTRUCCIONES_DE_TRANSFERENCIA_DE_DATOS) LD, LDI, LDIR, LDD, LDR, EX y EXX.

**22 13** [**INSTRUCCIONES ARITMETICAS**](#INSTRUCCIONES_ARITMETICAS) ADD, ADC, SUB y SBC.

**24 14** [**INCREMENTO Y DECREMENTO**](#INCREMENTO_Y_DECREMENTO) INC y DEC.

**24 15** [**OPERACIONES LÓGICAS**](#OPERACIONES_LOGICAS) CPL, NEG, OR, AND y XOR.

**25 16** [**ROTACION Y DESPLAZAMIENTO**](#ROTACION_Y_DESPLAZAMIENTO) SLA, SRA, RLC, RRC, RL, RR, RLD y RRD.

**27 17** [**MANIPULACIÓN DE BITS**](#MANIPULACION_DE_BITS) SET y RES.

**28 18** [**VERIFICACION Y SALTO**](#VERIFICACION_Y_SALTO) JP, JR, CP, CALL, RET y RST.

**29 19** [**ENTRADA/SALIDA**](#ENTRADA_SALIDA) IN[I|IR|D|DR], OUT[I|D|DR] Y OTIR.

**30 20** [**INSTRUCCIONES DE CONTROL**](#INSTRUCCIONES_DE_CONTROL) NOP, HALT, DI, EI y IM 1

**31 21** [**CONCLUSION CITY**](#CONCLUSION_CITY) Primer ejercicio.

**33 22** [**LIGERÍSIMA INTRO. A LA ARQUITECTURA DEL MSX**](#LIGERISIMA_INTRODUCCION_A_LA_ARQUITECTUR) Sólo la memoria básica de 64K.

**34 23** [**INTRO. A LOS ENSAMBLADORES DE VERDAD**](#INTRODUCCION_A_LOS_ENSAMBLADORES_DE_VERD) Directivas de los Ensambladores.

**37 24** [**SEÑORAS Y SEÑORES, PROGRAMEN Y VEAN**](#SEÑORAS_Y_SEÑORES_PROGRAMEN_Y_VEAN) Imprimiendo muchas As.

**39 25** [**INSISTO: ¡DOCUMENTACIÓN, POR FAVOR!**](#INSISTO_DOCUMENTACION_POR_FAVOR) Función de impresión de cadenas.

**41 26** [**INVERSIONATOR**](#INVERSIONATOR) Ejemplo de función sobre texto.

**50 27** [**EN EL PRÓXIMO CAPITULO…**](#EN_EL_PROXIMO_CAPITULO)

**51 28** [**SE ME ACABO LA CUERDA**](#SE_ME_ACABO_LA_CUERDA) Ahora a programar en SjasmPG.

**52 29** [**CAPERUCITA OBSOLETA**](#CAPERUCITA_OBSOLETA)

**30** [**KORE WA DOO TABERU KA**](#KORE_WA_DOO_TABERU_KA)

**31** [**¡HA GANADO UN SUBSLOT!**](#HA_GANADO_UN_SUBSLOT)

**32** [**LA ZONA DE WORKEO**](#LA_ZONA_DE_WORKEO)

**33** [**…ACABOSE**](#ACABOSE)

**34** [**¿COMO ESTAN USTEDEEEES?**](#COMO_ESTAN_USTEDEEEES)

**35** [**MAIN RAM: 128K**](#MAIN_RAM_128K)La memoria mapeada.

**36** [**¿ASI DE FÁCIL?**](#ASI_DE_FACIL)

**37** [**¿CUANTA RAM HAS DICHO?**](#CUANTA_RAM_HAS_DICHO)

**38** [**DOS 2 RULEZ**](#DOS_2_RULEZ)

**39** [**MEMORIZANDO**](#MEMORIZANDO)

**40** [**POSOLOGÍA**](#POSOLOGIA)

**41** [**“DONDE ESTAS LOBATÓN, OIGO TU COMPILAR…”**](#DONDE_ESTAS_LOBATON_OIGO_TU_COMPILAR)

**42** [**EL TRUKI DE LAS TRES TREINTA**](#EL_TRUKI_DE_LAS_TRES_TREINTA)

**43** [**CONCRETIZACIÓN DE LAS CROQUETAS**](#CONCRETIZACION_DE_LAS_CROQUETAS)

**44** [**ACABANDO, QUE YA LLEVAMOS 28K**](#ACABANDO_QUE_YEVAMOS_28K)

**45** [**CONVERSIÓN DE NUMEROS A ASCII**](#CONVERSION_DE_NUMEROS_A_ASCII)

**46** [**OPERACIONES CON ENTEROS**](#OPERACIONES_CON_ENTEROS)

**47** [**OPERACIONES CON PUNTO FIJO**](#OPERACIONES_CON_PUNTO_FIJO)

**48** [**COMPRESIÓN DE DATOS**](#COMPRESION_DE_DATOS)

**49** [**UN JUEGO SENCILLO**](#UN_JUEGO_SENCILLO)

**50** [**TABLAS Z80**](#TABLAS) Completa guía del Z80,

**51** [**VARIABLES BIOS**](#VARIABLES_BIOS)

**52** [**PPI POR BIOS**](#PPI_POR_BIOS)

**53** [**PSG POR BIOS**](#PSG_POR_BIOS)

**54** [**VDP POR BIOS**](#VDP_POR_BIOS)

**55** [**PPI POR PUERTOS**](#PPI_POR_PUERTOS)

**56** [**PSG POR PUERTOS**](#PSG_POR_PUERTOS)

**57** [**VDP POR PUERTOS**](#VDP_POR_PUERTOS)

**58** [**CRÉDITOS**](#CREDITOS)Game Over,

**00** **INTRODUCCIÓN** [**Índice**](#INDICE)

Este curso es una modificación y adaptación del genial curso Easymbler (ensamblador fácil) de Konami Man. El original está en texto plano y no está orientado al ensamblador “SjasmPG”, así pues yo “pipagerardo” mientras me lo estudio lo he adaptado a versión PDF imprimible y con enlaces para su fácil visionado en los ordenadores modernos. También he incluido fotos, tablas y gráficos para amenizar y mejorar el aprendizaje, por supuesto que también he modificado alguna parte siempre en vista de la compatibilidad con “SjasmPG” y con el ensamblaje cruzado. En este curso aunque se hacen referencias al lenguaje de programación **BASIC** no es imprescindible su conocimiento, aunque si tuviste un **MSX** seguro que lo has trasteado y mucho.



**Enlace del curso original de Konami Man:**

Easymbler <http://www.bitbucket.org/konamiman/msx>

El entorno de programación va a ser cruzado, o sea que no vamos a programar directamente en un MSX real, por eso necesitaras los siguientes programas para tu flamante y moderno ordenador (*Guindos*/*Linux*): un editor de texto orientado a lenguajes de programación (SciTE, Geany o similar), un compilador de ensamblador “SjasmPG” (este es obligatorio) y un emulador de MSX (OpenMSX, BlueMSX, etc ).

**Enlaces:**

SjasmPG <https://github.com/pipagerardo/sjasmpg>

OpenMSX <http://openmsx.org/>

BlueMSX <http://www.bluemsx.com/> (Opcional)

Geany <http://www.geany.org/Main/HomePage>

SciTE <http://www.scintilla.org/SciTE.html>

Yo personalmente estoy usando ( Geany + SjasmPG + OpenMSX ) pero para gustos colores.

Más adelante será necesario utilizar el emulador BlueMSX que aunque es más pesado que OpenMSX tiene un magnífico depurador compatible con “SjasmPG” y que nos pueden ayudar si tenemos problemas serios con algún programa nuestro.



Un consejo (consejos vendo y para mí no tengo), no te líes a programar en ensamblador hasta llegar al capítulo 21, puedes trastear y compilar un “hola mundo” pero te faltarán conocimientos y puedes frustrarte y claro sacar el hacha antes de tiempo...

Haz descansos cada media hora, lee y vuelve a leer los capítulos tantas veces como sea necesario u te volverás tarumba o lo que es peor te quedaras como el tipo de la foto...

Sin más dilación en la siguiente página comienza el fascinante curso de programación en ensamblador para nuestro querido MSX, Easymbler de Konami Man…

0

**01 POR QUÉ METERME EN ESTE FREGADO** [**Índice**](#INDICE)

Bienvenido al curso de Easymbler (ensamblador fácil) de Konami Man. Si estás leyendo esto es que por fin te has decidido a aprender a programar en ensamblador. O tal vez no acabas de decidirte, necesitas quizá un empujoncito… ¿Por qué programar en ensamblador? ¡Con lo bonito y sencillo que es el BASIC! Bien, yo te diré por qué.

- Los programas ensamblados son los más veloces. Te lo garantizo.

- Puedes aprovechar al 100% las posibilidades del ordenador. Incluyendo pleno acceso a la memoria (sí, sí, nada de disponer únicamente de 23K de RAM: 128K, 256K, 1024K… según tu máquina).

- Un programa ensamblado ocupa menos que su equivalente BASIC. Pero mucho menos.

- ¡¡El Nemesis está programado en ensamblador!!

Probablemente ya habías oído hablar de todas estas cualidades del CM (Código Máquina) (bueno, la última es cosa mía…). Pero un día cogiste un listado por casualidad y… ¡¡socorro!! ¿Qué es todo esto? ¡No entiendo nada! Vale. Pero recuerda el primer listado BASIC que viste. ¿Lo entendías? No. Ni papa. ¿Lo entiendes ahora? ¡Bueno, que si lo entiendes…! Pues lo mismo pasa con el CM. Y no hay peros que valgan. Simplemente recuerda esto:

\* Programar en ensamblador resulta complejo. Pero no difícil.

En efecto. La particularidad del CM radica en que estás programando el Z80 del modo más directo posible. Una instrucción suelta te permite hacer poco más que un POKE, una suma o enviar un dato a un puerto; por ello son necesarias bastantes instrucciones para hacer aquello que en BASIC se resume, por lo general, en una sola (si es que en BASIC también se puede hacer). Pero con un poco de práctica y un mucho de organización no hay razón para que no podamos desenvolvernos en la programación en ensamblador. Te lo digo por experiencia.

Si has conseguido llegar hasta aquí probablemente ya he logrado, al menos, quitarte ese miedo al CM que todos tenemos cuando no lo conocemos (yo también lo tenía, pero no tuve la suerte de encontrar un texto de iniciación tan genial como éste… vale, sólo bromeaba, no me peguéis…), o incluso puede que te haya convencido de que no se puede vivir sin saber programar en ensamblador (bueno, para mí “vivir” y “programar” son sinónimos). En cualquier caso, sigue leyendo: no te arrepentirás.

**02 ¿QUÉ ES EL ENSAMBLADOR?** [**Índice**](#INDICE)

Paciencia. Antes de ponernos a programar hemos de entrar un poco en situación. Algunos de los conceptos que introduciré ahora no resultan imprescindible para programar, pero el conjunto nos servirá para tener una idea de por dónde nos movemos, y de por qué el CM es distinto a los demás lenguajes de programación.

Como posiblemente ya sabrás, tu MSX contiene diversos circuitos integrados (chips para los que no quieren escribir tanto) imprescindibles para su funcionamiento.

El que ahora nos interesa es el Z80: el procesador central (CPU), el que controla todo el tinglado, el que ejecuta siempre el programa. Al Z80 están conectados el procesador de vídeo (VDP), el generador de sonido (PSG), el Interfaz programable para periféricos (PPI), la memoria y los slots de expansión, entre otros. Esto, de forma muy, pero que muy resumida, es un MSX.

Por ahora nos interesa el Z80. ¿Qué es un Z80? Pues ni más ni menos que un pedazo de piedra capaz de procesar (tratar, manejar, manosear...) datos. No pongas esa cara, y no subestimes los pedazos de piedra (fíjate en las pirámides: también están hechas de pedazos de piedra; pero éstas no procesan datos). Bueno, a lo nuestro. Este pedazo de piedra necesita estar conectado al mundo exterior (los chips nombrados antes) si realmente quiere procesar datos y aspirar a algo más que a decorar (cosa por otra parte lógica: ¿cómo va a procesar datos si no recibe los datos?), y de hecho lo está, por una serie de líneas de conexión, que forman los llamados buses ("caminos" para los datos), conectadas a sus patillas.

1

Simplifiquemos: nuestro Z80 está conectado, por ahora, únicamente a un chip de memoria, y únicamente tiene patillas para mandarle a la memoria una dirección, y para intercambiar datos con ésta; es decir, memoria y Z80 están conectados por medio de sólo dos buses: el de datos y el de direcciones (mentira podrida, claro, pero he simplificado para no asustar a nadie. De hecho no necesitamos más detalles.)

Así pues, tenemos un procesador y un chip de memoria interconectados de forma que pueden intercambiar datos (sí, exactamente igual que cuando haces **PEEK** o **POKE**). Concretemos un poco: un dato, en el caso del Z80, es un número de 8 bits, es decir, un número entero entre 0 y 255. Un número de este tipo se denomina byte. En cuanto a la memoria, un Z80 es capaz de direccionar directamente 64Kbytes, es decir, leer/escribir en una memoria de 65536 bytes. "¡Pero si mi MSX tiene 128K!" Ya hablaremos de eso más adelante (por algo he dicho direccionar "*directamente*"). Estos números tan raros vienen dados por el número de patillas del Z80: 8 para intercambiar datos y 16 para enviar Direcciones. ¿No encuentras la relación? Tranquilo, más adelante explicaré el sistema de numeración binario, y tus dudas serán despejadas.

Vamos progresando: nuestra memoria tiene hasta 65536 datos (0...255) y nuestro Z80 puede leerlos, procesarlos y volverlos a escribir. De hecho, es prácticamente lo único que puede hacer. "¡Venga ya! ¡¡El Nemesis no funciona a base de leer-procesar-escribir!!" Pues sí, tío listo: el Nemesis, el Metal Gear, el SD Snatcher y cualquier otro programa funcionan de esta forma.

Así pues, si queremos que nuestro Z80 funcione (procese un dato), le hemos de indicar:

- Dónde está el dato (o datos, caso de una suma, p. ej.) a procesar.

- Qué queremos que haga con el (los) dato(s) (tipo de proceso).

- Dónde queremos que almacene el resultado del proceso.

En el fondo lo que estamos haciendo es enviarle al Z80 un dato junto a unas instrucciones de uso. "Instrucciones... esto me suena... ¡Claro! Las instrucciones del BASIC." Bingo.

Seguimos avanzando. Una instrucción es una orden que se le da al procesador para que realice un determinado proceso con un dato. ¿Y qué formato tiene una instrucción? Pues el único formato que el Z80 es capaz de recibir: bytes. Ni más ni menos. Así como en BASIC cada instrucción se identifica con una palabra en inglés, en CM las instrucciones son bytes: uno para las más cortas, cinco para las más largas (el conjunto de bytes que forman una instrucción se denomina su código de operación). Y como sabrás del BASIC, detrás de una instrucción siempre viene otra, y otra, y... en resumen: un programa.

"Pues si un programa en ensamblador consiste únicamente en números, apaga y vámonos; esto es imposible de aprender." Tranquilo, no te espantes: está todo previsto, te garantizo que programar en ensamblador no es teclear miles de números. Paciencia.

¿Qué tenemos hasta ahora? No poca cosa: un procesador de datos conectado a una memoria que, casualmente, contiene datos; y una manera de ordenarle al procesador que procese datos: enviándole bytes, que conforman instrucciones que, a su vez, forman un programa una vez puestas en secuencia.

A estas alturas ya estás en condiciones de ver (¡por fin!) un sencillo programa en ensamblador. Atención: ¡Allá va!

**3A** 34 00

**ED 44**

**32** 01 22



¡Ya está! No era tan difícil, ¿verdad? Un momento... ¿qué haces con ese hacha? ¡No lo hagas! ¡Piensa en tu familia, tu Twingo, tu MSX! Ah... ¿que el hacha es para mí? Eh... Je, je... cálmate, hombre, no cometas una estupidez; te dije antes que no programarías a base de números y lo mantengo. Te vuelvo a pedir paciencia.

2

El programa consta de tres instrucciones, que como ves no son más que secuencias de bytes (están expresados en notación hexadecimal; si no la conoces tranquilo, la explico más adelante), tal como te había contado antes. ¿Y qué hace el programa? Simplemente lee el contenido de la dirección de memoria &H0034, le cambia el signo y deposita el resultado en &H2201. (Recuerda “&H” son números en hexadecimal del BASIC)

No te asustes, ahora describo paso por paso el programa; normalmente, a la hora de programar, no nos pondremos a realizar un análisis tan detallado, pero es importante que, al menos una vez, examines un programa con este nivel de detalle.

Lo primero que hace el Z80 al ejecutar el programa es leer el valor **&H3A** de memoria. Este byte contiene, para el Z80, suficiente información como para saber que necesita leer dos bytes más (además de información sobre el proceso a realizar, claro). Así pues, antes de ejecutar la instrucción lee los dos bytes siguientes: &H34 y &H00. Con estos dos bytes compone una dirección de memoria: &H0034 (sí, ya sé que los bytes están guardados al revés; no te preocupes por ahora). A continuación lee el byte contenido en &H0034 y lo deposita en el acumulador (Una posición de memoria especial (un registro) situada en el interior del Z80).

Resúmen: "Lee el contenido de &H0034 y deposítalo en el acumulador."

Seguimos: el Z80 lee **&HED**, y al descodificar este byte sabe que necesita leer otro: **&H44**. La orden es: "Cambia el signo del byte contenido en el acumulador". Más: el Z80 lee **&H32**, y se da cuenta de que necesita dos bytes más. Los lee: &H01 y &H22. La orden es: "Deposita el contenido del acumulador en &H2201".

Eso es lo que hace, y fin del programa. Bueno, no exactamente; el Z80 no puede estarse quieto: lo que ahora haría es leer la siguiente posición de memoria y seguir ejecutando programa. Si no hay programa... peor para ti: el Z80 interpretará los bytes que encuentre en memoria como instrucciones; si son bytes aleatorios lo más probablemente es que el ordenador acabe colgándose. Tranquilo, repito que está todo previsto.

Pues esto es todo lo que hace el Z80: ir leyendo bytes de memoria, traducirlos a instrucciones y ejecutar éstas. Sí, en esto consiste el código máquina; y este proceso de lectura-ejecución no se detiene mientras el Z80 reciba alimentación (eléctrica, se entiende). "¿Y cómo es que al arrancar el MSX, se va al BASIC y no hace nada después del OK?" Los bucles de espera también existen en CM, listillo. Sí: cuando sale el prompt del BASIC, el Z80 se pone a ejecutar algo equivalente a esto:

10 **IF INKEY$**="" **THEN GOTO** 10

por eso parece que no hace nada. Así de simple.

Llegados a este punto te habrás planteado una pregunta que puede dar un poco de miedo a primera vista: "Si el Z80 sólo entiende esos códigos de operación, ¿qué coño es el BASIC?" ¿No te lo imaginas? El intérprete del BASIC (o de cualquier otro lenguaje de alto nivel) no es más que un programa en código máquina que se limita a leer las chorradas que le pasas por el teclado y traducirlas a órdenes (órdenes CM, desde luego) o, como siempre, soltarte un "*Sintáx error, giliposha*". Fuerte, ¿eh? Es como cuando descubres que la tierra no se sostiene sobre tortugas...

**03 LOS NEMÓNICOS** [**Índice**](#INDICE)

Por tu cara deduzco que no te ha hecho gracia lo de los códigos de operación. Además veo que le has cogido el gusto al hacha. Veo también que te acercas hacia mí con cara de "se va a enterar este estafador de mierda". Veo que he de salir corriendo ahora mismo; bueno, por el camino te explicaré algo que tal vez te tranquilice.

Cierto es que es imposible memorizar todos los códigos de operación. Además, depurar un programa escrito de esta manera y calcular las direcciones para los saltos (¡que también hay **GOTO** y **GOSUB** en ensamblador, majos!) se convierte en una tarea mentalmente matadora. Y sin embargo existe el Nemesis. ¿Acaso tiene el "Homo Konamis" un cerebro más desarrollado que el resto de los humanos? No, no se trata de eso. Para que programar en código máquina no resulte una misión imposible para un ser humano, cada vez que se desarrolla un procesador con su correspondiente juego de instrucciones se diseña también un juego de nemónicos.

3

Un nemónico es un nombre simbólico que se le asigna a cada instrucción con el fin de facilitar su memorización y, de esta forma, "humanizar" la redacción de programas. Reescribamos el programa anterior, aquel que leía-negaba-escribía, con nemónicos.

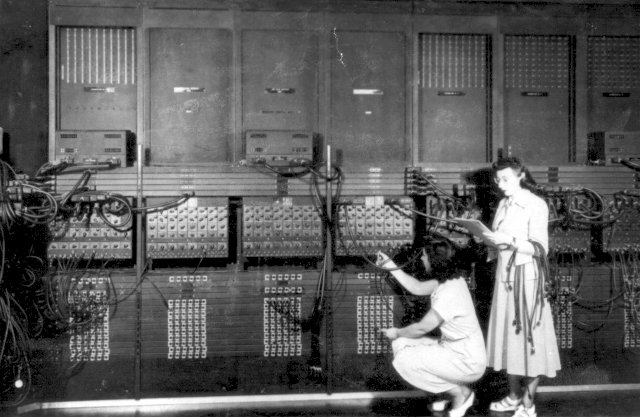
El resultado es algo así:

**LD** A,[$0034] ; En SjasmPG usamos “$” en vez de “&H”

**NEG** ; NEGar

**LD** [$2201], A ; LoaD

¡Esto ya es otra cosa! Simplemente teniendo en cuenta que **LD** es la abreviatura de LoaD (en inglés, "Cargar"), A la de Acumulador, y **NEG** la de NEGar (cambiar el signo), la cosa se vuelve un poco más comprensible, ¿verdad? Pues puedes ir acostumbrándote, este será el aspecto que tendrán tus programas en ensamblador (sólo que más largos. Muuuuucho más largos...)

"Pero a ver, ¿no habíamos quedado en que el Z80 sólo entiende bytes?" Cierto: para que un programa escrito con nemónicos funcione, ha de ser traducido a códigos numéricos (es decir, ha de ser ensamblado) por un programa especial llamado, precisamente, ensamblador. "¿Y cómo ha sido programado el ensamblador?" Pues con otro ensamblador más antiguo o con algún lenguaje de alto nivel. "¿Y el primer ensamblador?" Probablemente con otro ordenador ya existente: es lo que se llama ensamblado cruzado. "¿Y el primer ensamblador del primer ordenador?" Hijo, ¿y yo qué sé? El primer ordenador se programaba a base de poner y quitar cables y válvulas. "¡*No jodas*!" Que sí... (Casi me gustaba más hacha en ristre pero con la boquita cerrada)

Una última cosa, a propósito de esa sonrisita maliciosa en tu cara. Sí, podrías diseñarte tu propio juego de nemónicos, al estilo de **TRAEACAPACA** en lugar de **LD**, o **QUENOCOÑO**, **QUENO** en vez de **NEG**. Pero nosotros usaremos los nemónicos que fueron diseñados por los señores de Zilog (los que inventaron el Z80, ¿o acaso creías que la Z era de "ZOI ER MEHO"?) hace ya veinte añitos, que además están estandarizados y son los que todo el mundo usa. Caso de utilizar tus propios nemónicos tendrías que programar tu propio ensamblador, y además nadie sin tu fino sentido del humor entendería tus programas (y no olvides que no eres el único que tiene un hacha...)

Estupendo: sabes cómo funciona un Z80 y sabes que existen los benditos nemónicos. Ya podemos ponernos a empollar instrucciones y empezar a programar. Pues me temo que... ¿has soltado el hacha? Bien, pues ahora nos toca aprender un poco de... matemáticas.

**04 EL SISTEMA BINARIO** [**Índice**](#INDICE)

Vaya, he dicho la palabra maldita. Suelta el hacha, hombre; yo también odio las matemáticas, pero en la programación en ensamblador, tarde o temprano siempre hemos de tratar números binarios. Pero tú tranquilo; en el fondo el sistema binario es bastante facilón.

Observa tus manos. Sí, en estos momentos contienen un hacha; suéltalo. Veeenga, suéltalo. Así. Mucho mejor. Observa ahora tus manos: ¿qué tienen? Pues ni más ni menos que diez dedos. Así pues, es lógico que nuestro sistema de numeración conste de diez dígitos. De esta forma, cuando queremos contar hacemos algo así:



0 1 2 3 4 5 6 7 8 9

¡Guep! ¡¡Nos hemos quedado sin dígitos!! No hay problema: empleamos dos, y ya está:

10 11 12 13 14 15 16 17 18 19 20 21 ... 98 99

¡De nuevo sin dígitos! Pues nada, nada: usaremos tres.

100 101 102 ... 198 199 200 ... 988 999

4

Le vamos cogiendo el truco: cada vez que un dígito pasa de 9, vuelve a 0 y se incrementa el dígito de su izquierda. Cuando es el de más a la izquierda el que pasa de 9, simplemente añadimos un 1 a su izquierda, y vuelta a empezar.

Decimos que el sistema decimal es un sistema posicional: el valor de cada dígito depende de su posición. Así, el número 1974 (que casualmente corresponde al año de nacimiento del gran Konami Man *y del mío también PipaGerardo*) en realidad significa:

4 + 7\*10 + 9\*100 + 1\*1000 = 4 + 70 + 900 + 1000

O, con más formalidad matemática:

4\*10^0 + 7\*10^1 + 9\*10^2 + 1\*10^3

El dígito de más a la derecha se multiplica por diez elevado a cero, el siguiente por diez elevado a uno... y así con todos los dígitos que haya. En resumen, que cualquier número decimal se puede descomponer en potencias de diez.

Pues la he hecho buena: te hago soltar el hacha y ahora me vienes con la camisa de fuerza. Menos mal que me he acostumbrado a correr. Tranquilo, no me he vuelto tarumba: esto sigue siendo un curso de iniciación al ensamblador.

Todo esto ha venido a cuento porque, antes de entrar en un nuevo sistema de numeración, es necesario conocer más a fondo el propio sistema decimal, puesto que todos funcionan igual: sólo cambia el número de dígitos y la base (el número que elevamos a 0, 1, 2...)

"ZZZZZ..." ¡Eh! ¡¡*Despierta*!! Que ya empiezo con el sistema binario.

Decía al principio que nuestras manos tienen diez dedos. Los ordenadores se han diseñado de forma que sus "*manos*", es decir, la corriente eléctrica, sólo tenga dos "*dedos*": valor bajo (no hay corriente, 0) y valor alto (hay corriente, 1). Esto se ha hecho así por motivos tecnológicos que no vienen a cuento ahora. Lo importante es que los ordenadores (y el Z80, *ofcoursemente*) "cuentan" con únicamente dos dígitos: 0 y 1. Un bit no es más que un dígito binario ("Binary digIT").

Imaginemos que nuestro Z80 se pone a contar, tal como lo haríamos nosotros:

0 1

Como no le quedan más dígitos, añade uno:

 10 11

Vaya, otra vez sin dígitos... bueno, vamos añadiendo:

100 101 110 111 ->

-> ¡Otro más!: 1000 1001 1010 1011 1100 1101 1110 1111

Como ves funciona exactamente igual que el sistema decimal al que estamos acostumbrados, pero con sólo dos dígitos. Así, si un número decimal se podía descomponer en potencias de diez, un número binario se descompone en potencias de... ¿lo adivinas?

101001 = 1\*2^0 + 0\*2^1 + 0\*2^2 + 1\*2^3 + 0\*2^4 + 1\*2^5 =

= 1\*1 + 0\*2 + 0\*4 + 1\*8 + 0\*16 + 1\*32 = 41

¡Ya está: esto son los números binarios! No era tan difícil, ¿verdad? Así, para transformar un número binario a decimal, sólo has de fijarte en las posiciones en las que hay un 1, y sumar entonces dos elevado a esa posición (teniendo en cuenta que la posición de más a la derecha es la cero, y que 2^0=1). Tampoco es necesario aprenderse todas las potencias de dos de memoria: para pasar a la siguiente basta multiplicar por dos. Aunque tú tranquilo: a medida que programes en ensamblador acabarás sabiendo de memoria, como mínimo, las diez primeras potencias de dos.

5

Retrocedamos un poco: recuerda que el Z80 puede manejar datos entre 0 y 255 porque sólo tiene 8 líneas para datos. Esto quiere decir que puede tratar números entre 00000000 y 11111111, y 11111111 = 1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255

Observa que, casualmente, 2^8 = 256. Sí: con n bits podemos representar 2^n números, de 0 a 2^n -1.

Recuerda también el número de líneas para direcciones: 16. Esto da un total máximo de 2^16 = 65536 direcciones de memoria accesibles.

Ya sabemos traducir números binarios a decimales, pero, ¿y el proceso inverso? Hay un procedimiento bastante sencillo que consiste en realizar divisiones sucesivas del número, quedándonos con los restos. Lo mejor será ver directamente un ejemplo: ¿cómo se expresa 1974 en binario?

1974 : 2 = 987, R = 0 -> bit 0

987 : 2 = 493, R = 1 -> bit 1

493 : 2 = 246, R = 1 -> bit 2

246 : 2 = 123, R = 0 . . .

123 : 2 = 61, R = 1

61 : 2 = 30, R = 1

30 : 2 = 15, R = 0

15 : 2 = 7, R = 1

7 : 2 = 3, R = 1

3 : 2 = 1, R = 1

1 : 2 = 0, R = 1 -> bit 10

Nos detenemos cuando el resultado de la división sea cero (dividendo menor que divisor, o sea 2), y componemos el número binario de la siguiente forma: empezamos tomando el último resto obtenido, y vamos poniendo los anteriores a su derecha (aprovecho para indicarte que el bit situado más a la izquierda en un número binario se denomina bit más significativo, bit de más peso o, simplemente, MSB, de "Most Significative Bit"; el bit de más a la derecha es el menos significativo, o LSB, de "Less Significative Bit". De esta forma, el último resto obtenido, que es el primero que tomamos, se transforma en el MSB del número binario resultante). Así, 1974 en binario se expresa como 11110110110.

"¿Ya está? ¿Hemos acabado con los números binarios?" No, por desgracia siguen vivos... Nos quedan aún un par de puntos para liquidar el asunto de los números binarios: la suma y los números negativos.

"Pero... ¿es que estos engendros se pueden sumar?" ¡Y tanto! Y además es bien fácil, al igual que ocurre con los números decimales. Simplemente hay que tener en cuenta cuatro reglas: 0+0=0, 0+1=1+0=1, 1+1=0 y me llevo 1, 1+1+1 que me llevaba =1 y me llevo 1.

Ejemplo: 0111 + 0011 (es decir, 7+3) =>

0111 7

+ 0011 +3

---- --

???? ¿?

- Empezamos por los bits de la derecha: 1+1=10, es decir 0 y me llevo 1:

1 1 1

0111 0111 0111 0111 7

+ 0011 + 0011 + 0011 + 0011 +3

---- ---- ---- ---- --

0 10 010 1010 10

El resultado es 1010 = 10, es decir, 7+3, lógicamente. Como ves, de difícil no tiene nada. Sólo hay que tener cuidado con un posible desbordamiento, es decir, el resultado puede necesitar más bits que los dos números iniciales:

1111 15

+ 0010 +2

---- --

10001 17

6

Aquí, 15+2=17, que no cabe en 4 bits. Este fenómeno se denomina desbordamiento, y hay que tenerlo en cuenta a la hora de programar: recuerda que los datos que maneja el Z80 son de 8 bits; por tanto no puedes almacenar, por ejemplo, el resultado de 200+100 en un solo byte. Pero tú tranquilo, el Z80 dispone de mecanismos especiales para tratar desbordamientos, como veremos más adelante.

Queda el asunto de los números negativos. Existen varias formas de expresar un número negativo en notación binaria; nosotros iremos al grano y pasaremos directamente al complemento a dos, que es el método usado en la práctica por ser compatible con las operaciones de suma y resta.

Para expresar un número negativo en notación binaria en complemento a dos, simplemente hay que hacer lo siguiente:

- Expresar el valor absoluto (número sin signo) en binario normal.

- Invertirlo: cambiar los ceros por unos y viceversa.

- Sumarle uno.

Así, para expresar -34 en complemento a dos:

0100010 -> 1011101 -> 1011110

Y ya está: la representación binaria de -34, en complemento a dos, es 1011110. Veamos si este método es consistente: ¿qué pasa si buscamos el complemento a dos de -34, es decir, -(-34)?

1011110 -> 0100001 -> 0100010

Obtenemos de nuevo el 34: así pues, el método del complemento a dos funciona. Sólo has de tener en cuenta que el número de bits necesario para representar un número puede cambiar con respecto al binario puro: en efecto, con n bits podíamos representar números entre 0 y 2^n -1. En complemento a dos aún podemos representar 2^n números distintos, pero ahora la mitad serán positivos y la otra mitad negativos. Por ejemplo, con 8 bits podíamos representar del 0 al 255; en complemento a dos serán números entre **-128** y **127** (128 negativos + 127 positivos + el cero =256). En general, con n bits los límites de representación son -(2^n)/2 y (2^n)/2 -1.

"¿Y cómo puedo distinguir a simple vista si un número expresado en complemento a dos es positivo o negativo?" Fácil: observa el MSB. Si es cero, el número es positivo. Si es uno, negativo. Así de simple. Haz la prueba: con 8 bits el máximo número representable es el **127**, en binario 01111111; el mínimo es **-128**, es decir 10000000. Sabemos sumar. Sabemos expresar números negativos. ¿Sabemos restar...? ¡Claro! ¿Qué es restar, sino sumar un número positivo y uno negativo? En efecto: A-B no es más que A+(-B).

Veamos un ejemplo: **5** **-2**, usando 8 bits. Si hacemos una suma normal y corriente obtendremos:

00000101 5

+ 11111110 -2

--------- --

100000011 ?

¡El resultado tiene un bit más! ¿Se ha generado desbordamiento? Imposible, el resultado es 3 y cabe holgadamente en 8 bits. Este bit sobrante se llama acarreo y debe ignorarse. En efecto, si así lo hacemos el resultado de la resta es correcto: 00000011 = 3.

Retrocedamos ahora un poco y veamos una simple suma: **126** + **3**.

01111110

+ 00000011

---------

10000001

Ateniéndonos a las reglas del complemento a dos, vemos que el resultado es... -127. ¿Cómo es posible? Ahora sí ha habido desbordamiento, a pesar de que no sobre ningún bit. En efecto, el resultado correcto es 129, que no cabe en ocho bits; por eso hemos obtenido un resultado absurdo.

7

"¡Pues sí que estamos listos! Ahora resulta que, con esto del complemento a dos, puedes tener desbordamiento sin bits sobrantes, y puedes tener bits sobrantes sin desbordamiento. A que cojo el hacha otra vez..." Estas un poco nerviosillo, ¿verdad? Tranquilo, hay una regla sencilla para averiguar si el resultado de una suma ha producido desbordamiento.

Observa el MSB de los dos sumandos. Si uno es cero y el otro uno, estás sumando un número positivo y uno negativo: seguro que no hay desbordamiento, así que si se produce un acarreo ignóralo y quédate con el resultado, que seguro que es correcto. En cambio, si los dos MSB son iguales es que los dos números son del mismo signo: el resultado también ha de tener ese signo (es decir, el mismo MSB); en caso contrario ha habido desbordamiento y el resultado no es correcto.

Pues ya hemos visto todo lo que necesitábamos sobre números binarios. "¡Oye, que no me gusta este tipo de bromas! ¡A que cojo el hacha...!" Te juro por *San Seed* que hemos acabado la sección sobre notación binaria. Que sí, hombre, si al final todo llega...

**05 LAS OPERACIONES LÓGICAS** [**Índice**](#INDICE)

Seguramente ya las habrás usado en tus andares BASICeros, o al menos habrás oído hablar de ellas: las famosas **NOT**, **AND**, **OR** y **XOR**. En esta ocasión se trata de operaciones entre dos bits que dan como resultado otro bit, sin más historias. Como no quiero aburrirte, y veo que te encaminas ansiosamente hacia el hacha, paso sin más a describir estas operaciones; verás cómo acabamos enseguida (o este tío acaba conmigo enseguida...)

\* **NOT**: negación. NOT 0 = 1, NOT 1 = 0, es la única operación de un solo bit.

\* **AND**: producto lógico, vale uno sólo si los dos bits son uno:

0 AND 0 = 0

0 AND 1 = 0

1 AND 0 = 0

1 AND 1 = 1

\* **OR**: suma lógica, vale uno si al menos uno de los dos bits vale uno.

0 OR 0 = 0

0 OR 1 = 1

1 OR 0 = 1

1 OR 1 = 1

\* **XOR**: OR exclusiva. El resultado es uno si sólo uno de los bits es uno.

0 XOR 0 = 0

0 XOR 1 = 1

1 XOR 0 = 1

1 XOR 1 = 0

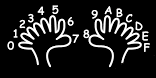
Estas operaciones lógicas (hay más, pero me las he saltado porque no las necesitaremos) son importantísimas en la programación en ensamblador, se usan incluso más que la suma. Ya lo irás descubriendo a medida que programes, que será dentro de poco (suelta el hacha, rey, anda, *porfa*...)

**06 LA NOTACIÓN HEXADECIMAL** [**Índice**](#INDICE)

Juro por San Burton que esta es la última sección matemático-numérica. Piensa que el *Konamimanicidio* está penado por la ley (supongo...), relájate y disfruta de esta genial sección (por decir algo).

Volvemos al símil de los dedos. Imagina un *snatcher* que ha salido defectuoso y tiene 16 dedos en las manos; así, su sistema de numeración tiene 16 dígitos: es el llamado sistema hexadecimal. Como nuestro pobre *snatcher* sólo conoce diez dígitos (de hecho no existen más), emplea las letras de la A a la F para representar los cinco dígitos que hay después del 9. Así, si se pone a contar...

8



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

Ya conoces la historia: ahora toca añadir otro dígito...

10 11 12 ... 19 1A 1B 1C 1D 1E 1F 20 ...

Atención que esto puede llegar a ser hasta divertido: ¿qué número viene tras el 99...? ¡El 9A, por supuesto! ¿Y tras el 9F? ¡¡El A0!! Tras el A9 el AA, tras el AF el B0... y así hasta el FF, que viene seguido por el... ¡¡¡100!!! ¡Yupi! ¡Viva! ¡Fiesta! (a ver si convenzo al menda de que suelte el hacha, y mientras va a buscar la camisa de fuerza huyo...)

Así pues, la historia se repite: cualquier número hexadecimal se puede descomponer en potencias de 16...

7B6 = 6\*16^0 + 11\*16^1 + 7\*16^2 = 6\*1 + 11\*16 + 7\*256 =

= 1974 (caramba...)

... y para pasar un número de decimal a hexadecimal sigue siendo válido el método de las divisiones sucesivas:

1974 : 16 = 123, R = 6

123 : 16 = 7, R = 11 -> B

7 : 16 = 0, R = 7

Resultado: 1974 en hexadecimal es 7B6, por si aún no lo habías notado. Y por último, indicarte que con n dígitos puedes representar 16^n números.

"¡Cuando por fin habíamos acabado con los binarios e íbamos a ponernos por fin a programar, se saca de la manga un sistema de base 16! Primero le pongo la camisa de fuerza y luego le doy con el hacha." *Tranquiiiilo*, caaaaalma, hombre, ahora te explico la utilidad del sistema hexadecimal.

A ver, ¿cuántos números puedes representar con dos dígitos hexadecimales? Pues 16^2 = 256. ¡Caramba, qué casualidad, si es la cantidad de números representables con un byte! Sí: hexadecimalmente hablando, con un byte puedes representar 100 números, de 0 al FF... ¡*son números redondos*! ¿Y con cuatro dígitos? Pues 16^4 = 65536... ¡¡64Kbytes!! Exactamente 10000 posiciones de memoria, de 0 a FFFF, son las que puede direccionar el Z80.

Veamos ahora el pseudo-más difícil todavía: ¡¡pasar de binario a hexadecimal!! "¡Ya está! ¡Le doy con el hacha...!" Calma, hombre, que me estoy cansando de correr. He dicho pseudo-más difícil todavía porque en realidad nos encontramos ante el más fácil imposible. Para pasar un número de binario a hexadecimal basta dividirlo en grupos de 4 bits, y convertir cada grupo, que representará un número entre 0 y 15, a su dígito hexadecimal correspondiente.

Ejemplo: ¿recuerdas el binario de 1974? Sí hombre: 11110110110. Lo dividimos en grupos de cuatro bits (un grupo de cuatro bits recibe el nombre de nibble), empezando por la derecha; si el último grupo (el situado más a la izquierda) no tiene cuatro bits simplemente añadimos ceros a la izquierda:

0111 1011 0110

Tenemos que 0111 = 7, 1011 = 11 = B, y 0110 = 6, luego 1974 en hexadecimal es 7B6. ¡¿A que no te lo esperabas?! Por eso es tan importante la notación hexadecimal: permite expresar bytes con sólo dos dígitos, facilitando además un método muy sencillo de conversión a notación binaria.

Retrocedamos un poco (bueno... bastante), hasta aquel primer programa de ejemplo. El código de operación de **NEG**, expresado en hexadecimal, era **ED 44**. Como E = 14 = 1110, D = 13 = 1101 y 4 = 0100, tenemos que lo que en realidad el Z80 está recibiendo por sus patillas es 11101101, seguido de 01000100; un uno significa que llega corriente a la patilla correspondiente, un cero que no llega. Así es como realmente recibe las instrucciones el Z80; ¿no es más cómodo expresarlas en hexadecimal...? (que sí, que nosotros usaremos nemónicos, tranquilo)

9

Pues ya hemos terminado con las matemáticas. De regalo, y como premio por haber soportado tan pacientemente tan pesada sección, he aquí una tabla de conversión/referencia rápida de decimal a hexadecimal y a binario para nibbles:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Dec** | **Hex** | **Bin** |  | **Dec** | **Hex** | **Bin** |
| 0 | 0 | 0000 |  | 8 | 8 | 1000 |
| 1 | 1 | 0001 |  | 9 | 9 | 1001 |
| 2 | 2 | 0010 |  | 10 | A | 1010 |
| 3 | 3 | 0011 |  | 11 | B | 1011 |
| 4 | 4 | 0100 |  | 12 | C | 1100 |
| 5 | 5 | 0101 |  | 13 | D | 1101 |
| 6 | 6 | 0110 |  | 14 | E | 1110 |
| 7 | 7 | 0111 |  | 15 | F | 1111 |

**07 RESUMEN DE CONCEPTOS** [**Índice**](#INDICE)

Este apartado es un resumen de los conceptos más importantes que hemos visto hasta ahora; está pensado para fijar ideas, aunque también te puede servir como referencia rápida en el futuro. Ahí va:

- El Z80 es el procesador central del MSX. Es capaz de procesar datos en función de una serie de órdenes (instrucciones). Tanto los datos como las instrucciones tienen forma de bytes.

- La memoria conectada al Z80 es la encargada de almacenar los datos y las instrucciones que forman el programa. Memoria y Z80 se comunican a través del bus de datos (8 líneas de conexión) y del bus de direcciones (16 líneas). También hay un bus de control, claro, pero a nivel de programación no nos interesan. El Z80 es capaz de direccionar (o gestionar) directamente 64Kbytes de memoria, es decir, 65536 bytes (1K=1024).

- Una instrucción es una orden dada al Z80 para que realice una determinada acción (normalmente procesar un dato, aunque también hay otro tipo de instrucciones, p.ej. para generar saltos). Está formada por uno o más bytes; en este caso, tras leer el primero y descodificarlo el Z80 ya sabe cuántos bytes forman la instrucción, y los lee de memoria antes de realizar cualquier otra acción.

- El código de operación de una instrucción es, precisamente, el listado ordenado de los bytes que la conforman. Por ejemplo, el código de operación de **NEG** es **ED 44**, que no es lo mismo que 44 ED. Aunque el Z80 los trata en formato binario, los pobres mortales solemos manejarlos transformados a hexadecimal.

- Los nemónicos son nombres simbólicos asociados a cada código de operación, con el fin de facilitar su memorización y tratamiento por parte de los humanos. Por ejemplo, **NEG** para **ED 44**. Para que el Z80 los entienda han de ser transformados a códigos de operación (ensamblados) y guardados en memoria, bien a mano (¡*socorro*!) bien por un programa ensamblador.

- Ensamblar un programa no es más que traducir los nemónicos a códigos de operación. La palabra "ensamblador" se refiere tanto al programa encargado de realizar esta pesada acción, como al lenguaje de programación basado en los nemónicos.

- El sistema de numeración binario se basa en la descomposición de los números en potencias de dos; un número binario está compuesto por bits. Es el sistema usado de forma natural por los ordenadores y sus componentes en general, y por el Z80 en particular.

- Un bit es un dígito binario. Sólo puede tomar dos valores posibles: 0 o 1.

- Un byte es un número de 8 bits, entre 0 y 255.

- Un nibble es un número de 4 bits, entre 0 y 15. También, por extensión, un dígito hexadecimal.

- El último bit a la izquierda de un número binario es el que tiene más peso en el número. Por ello es llamado bit de más peso, bit más significativo o MSB.

10

- Análogamente, el primer bit por la derecha es el bit de menos peso, bit menos significativo o LSB.

- Complemento a dos es el nombre del método usado para expresar números negativos en notación binaria. Consiste en invertir el valor absoluto del número (cambiar los ceros por unos y viceversa) y posteriormente sumar uno al resultado. El MSB de un número expresado en complemento a dos es siempre cero si el número es positivo, y uno si es negativo.

- Cuando el resultado de una suma o resta en binario consta de más bits que los sumandos decimos que hay desbordamiento. En la teoría no representa problema, pero no debemos olvidar que en la práctica el número de bits por dato está limitado (a 8 en el caso del Z80). Por este motivo los procesadores disponen de mecanismos especiales para tratar desbordamientos (y acarreos).

- Acarreo es un bit extra que puede generarse a la hora de realizar operaciones de suma y resta en complemento a dos. Se debe ignorar y tomar el resto de bits como resultado correcto. No confundir acarreo (es un bit) con desbordamiento (es una situación): puede haber uno sin que se genere el otro.

- Las operaciones lógicas (**NOT**, **OR**, **AND**, **XOR**) se efectúan entre dos bits y dan como resultado otro bit. Son muy importantes en la programación en ensamblador.

- El sistema de numeración hexadecimal se basa en la descomposición de los números en potencias de 16. Los dígitos hexadecimales son los números del 0 al 9 y las letras A a F. Existe un procedimiento muy sencillo para pasar un número binario a hexadecimal y viceversa, mediante la descomposición directa en nibbles.

- En un sistema de numeración basado en potencias de B (lo llamamos el sistema de base B), con N dígitos pueden expresarse B^N números distintos, de 0 a B^N -1. En el caso de usar complemento a dos (pasando el número a binario, realizando el complemento y volviendo luego a base B), el rango de representación va de -(B^N)/2 a (B^N)/2 -1. Tranquilo: nosotros sólo usaremos B=2 y B=16 (y B=10, *ofcoursemente*). En el pasado también se usaba B=8 (sistema octal, ya en desuso).

No está mal, ¿verdad? Ya sabemos bastantes cosas. Sí, ya sé que aún no hemos programado nada, pero era necesario: todos estos conceptos son básicos si quieres programar en ensamblador. De todas formas no intentes retenerlos: basta que te suenen, y para cuestiones concretas no dudes en consultar este resumen, o algún libro de programación, 34 veces por minuto si es necesario. Te garantizo que con la práctica todo este maremágnum de información te resultará tan familiar como la tabla de multiplicar. ¿Que no te lo crees? Todo lo que llevas leído lo he escrito sin consultar texto alguno. Sí: de memoria... y si yo puedo, ¿por qué no tú, cuando tengas práctica?

**08 ESTRUCTURA DEL Z80** [**Índice**](#INDICE)

Veo una lágrima correr por tu mejilla, al tiempo que sueltas el hacha con cara de felicidad. El sentimiento es mutuo (uf...), yo también estaba empezando a hartarme de matemáticas (y de hacha, claro).

Poca cosa nos queda por decir del Z80 en lo que respecta al exterior. Ya sabemos que está conectado a diversos circuitos, de los cuales sólo hemos hablado de la memoria (de hecho, cualquier chip o periférico con el que el Z80 pueda comunicarse puede contemplarse como si fuera memoria). Esta conexión se realiza mediante el bus de datos, el bus de direcciones y el bus de control. De este último no hemos hablado, ya que no es necesario conocer su funcionamiento para programar. De todas formas no está de más tener una idea general sobre las funciones de este bus. Formando el bus de control nos encontramos líneas de petición y reconocimiento de interrupciones (un periférico puede interrumpir temporalmente el desarrollo normal de un programa para ejecutar su propia subrutina: esto es una interrupción. Ya retomaremos el tema). También, por supuesto, líneas de control de la memoria; así, cada vez que el Z80 quiere acceder a memoria ha de advertírselo, e indicar si quiere leer o escribir; la memoria, por su parte, ha de avisar cuando el dato está en el bus de ídems listo para ser recogido.

11

Por último os mencionaré que hay una línea especial para provocar una reinicialización (reset) del procesador (que comienza entonces a ejecutar programa a partir de la posición 0000), así como, por supuesto, una línea conectada a un reloj que proporciona 3,5 millones de pulsos por segundo, proporcionando al procesador una referencia temporal imprescindible para su funcionamiento. Pasemos al interior del Z80. Creo haber dicho antes que el Z80 no es más que un pedazo de piedra... pues eso es lo que encontramos en su interior: capas de piedra (básicamente silicio) formando transistores, condensadores y demás parafernalia electrónica. Pero tranquilo: al igual que a la hora de programar usamos nemónicos en lugar de bytes, también a la hora de describir el interior del Z80 emplearemos un modelo comprensible, basado en registros.

Así pues, ateniéndonos a nuestro modelo, en el interior del Z80 podemos encontrar los siguientes componentes:

- Registros de propósito general. Un registro no es más que una memoria para un sólo dato; el Z80 tiene registros de 8 bits y registros de 16 bits. Los de este grupo se denominan "de propósito general" porque no tienen una función específica determinada de antemano: el cometido de estos registros dependerá del programa, están ahí para que el programador (¡tú!) los use como mejor le convenga. Podríamos compararlos a las variables del BASIC. La ventaja de los registros con respecto a la memoria convencional es clara: al estar situados en el interior del Z80, el acceso a ellos resulta mucho más rápido. Además, muchas operaciones de proceso de datos (de hecho la mayoría) sólo pueden realizarse con estos registros, y no con la memoria directamente.

- Registros de uso específico. Cada uno de estos registros tiene una razón de ser, una función específica: el Z80 los usa al controlar el desarrollo del programa. Aunque son accesibles al programador, nunca los usaremos para almacenar nuestros propios datos, y no los modificaremos a no ser que estemos muy seguros de lo que hacemos, pues nos arriesgamos a un cuelgue de los que hacen época.

- Unidad Aritmético Lógica (ALU). Tiene dos entradas y una salida, y es la encargada de realizar las operaciones aritméticas (suma, resta, ajuste decimal), las operaciones lógicas y las rotaciones y desplazamientos (¡Guep! ¡De estas no habíamos hablado! Tranquilo: ver·las, *veremoslas*).

- Unidad de control. Es la encargada de descodificar las instrucciones y controlar, en general, el desarrollo de las operaciones a realizar y la coordinación del sistema. Recuerda que "el Z80 sabe, tras leer el primer byte de una instrucción, si ha de leer más"; pues bien, es aquí donde se toma tal decisión, tras la descodificación del primer byte.

- Por supuesto, los buses: el de datos (conectado a todos los registros, la ALU y la unidad de control), el de direcciones (conectado a ciertos registros), el de control, y buses internos encargados de la interconexión de los diferentes elementos del procesador.

¡Y se acabó! Esto es, en suma, un Z80. "¿De verdad? ¡¿No hay más?!" En lo que a programación respecta, no. De hecho, yo no conozco muchos más detalles del Z80, y en el libro de programación que tengo tampoco se especifica gran cosa más (está bien, me has pillado: ahora no escribo tan de memoria...) Bueno, podemos pasar a examinar con detalle la parte que nos interesa: los registros.

**09 LOS REGISTROS DEL Z80** [**Índice**](#INDICE)

De todas las partes del Z80 explicadas anteriormente, únicamente necesitamos conocer a fondo los registros a la hora de programar; y, como veremos, no todos los registros tienen la misma importancia... Empezaremos esta andadura *registrera* con los de propósito general, los "*nuestros*". Los que tiene el Z80 son:

\* El acumulador, o registro A, de 8 bits. Es el registro más versátil del Z80; muchas operaciones sólo pueden realizarse si el dato ha sido cargado previamente en el acumulador. Recuerda **NEG**, la instrucción que cambiaba el signo de un byte; esta instrucción actúa siempre, única y exclusivamente sobre el dato contenido en el acumulador.

12

\* El registro de estado o de **banderas**, o registro F (de "**flags**"). Este registro no es exactamente de propósito general; lo he puesto aquí porque está unido físicamente al acumulador, y bajo determinadas circunstancias puede verse el par AF como un registro de 16 bits.

Aunque físicamente se trata de un registro de 8 bits, en realidad el registro de estado está compuesto por seis banderas de un bit (dos bits quedan sin uso). Estas "banderas" se activan o desactivan en función del resultado de las operaciones realizadas por las diferentes instrucciones. Así, en la descripción detallada de cada instrucción no puede faltar el efecto causado sobre las banderas.

La situación física de las banderas sobre el registro F es la siguiente:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **bit 7** | **bit 6** | **bit 5** | **bit 4** | **bit 3** | **bit 2** | **bit 1** | **bit 0** |
| **S** | **Z** | **-** | **H** | **-** | **P/V** | **N** | **C** |
| signo | cero |  | acarreo  mitad |  | desbord. | resta | acarreo |

Como ya he dicho, cada bandera se activa o desactiva (se pone a 1 o a 0) en función de la instrucción ejecutada, pero se puede hacer (y de hecho voy a hacer) una descripción general de la función de cada una de ellas:

- **S**: bandera de signo. Esta bandera refleja el signo (el MSB) del resultado de determinadas operaciones aritméticas, lógicas, de rotación o desplazamiento, y de algunas operaciones de transferencia de datos entre registros.

- **Z**: bandera cero. Se activa si el resultado de la operación realizada por la instrucción es cero. Ojo, no te confundas: el valor de la bandera es UNO cuando el resultado de la operación es CERO.

- **H**: bandera de acarreo mitad. Se activa cuando hay acarreo del bit 3 al 4 en una operación aritmética. Es usada por el Z80 y raramente resultará útil para el programador (sí: ¡¡tú!!).

- **P/V**: paridad/desbordamiento. Esta bandera tiene dos funciones distintas, dependiendo de la instrucción. Tras una operación lógica o de rotación / desplazamiento, la bandera se activa si el número de bits puestos a 1 del resultado es par (bueno, yo nunca le he encontrado la utilidad a esto, pero supongo que la tendrá...) Tras una operación aritmética se activa si se ha producido desbordamiento. "¿Desbordamiento? Me suena..." ¡Sí, hombre! Es el mismo desbordamiento del que hablábamos en la sección de notación binaria: el resultado de la operación no cabe en 8 bits. He aquí un "mecanismo especial" de los que te mencioné.

- **N**: resta. Se activa tras una resta, se desactiva tras una suma. Al igual que H, normalmente sólo es útil para el propio Z80.

- **C**: acarreo. ¡Sí!, aquel bit inútil que se generaba al restar. *Helo aquí*. Entonces dije que había que despreciarlo, pero más adelante veremos que en las instrucciones de rotación y desplazamiento, así como en las sumas y restas de 16 bits, la bandera de acarreo juega un papel vital.

Un par de cosillas sobre las banderas antes de seguir. Algunas instrucciones del Z80, una vez ejecutadas, activarán o desactivarán algunas banderas, dejando las otras tal como estaban. Otras instrucciones no tocarán ninguna bandera. Esto es importante a la hora de programar, ya que las instrucciones de salto condicionadas (equivalentes a **IF**...**GOTO**) se basan en el estado de las banderas; es importante comprobar que antes de un salto de este tipo no haya instrucciones que nos las machaquen. Como ya he dicho antes, en todo texto que detalle las características de las instrucciones del Z80 ha de constar el efecto sobre las banderas.

Ufff... menuda paliza, ¿eh? Recuerda: no intentes retener. Simplemente lee con calma. Cuando lo hayas leído 34 veces ya empezará a sonarte... (no, el hacha no, sólo bromeaba...) Bueno, quedan más registros por ver. ¡Animo!

13

\* Los pares BC, DE y HL. Cada uno de estos pares puede usarse como un registro de 16 bits, o como dos registros de 8 bits (entonces se transforman en B, C, D, E, H y L). Están conectados al bus de direcciones, por tanto pueden usarse como punteros a memoria cuando funcionan a pares (p.ej., existe la instrucción **LD** A, [HL]: "carga en el acumulador el contenido de la posición de memoria apuntada por HL". Parece mentira la capacidad de resumen que tienen los nemónicos, ¿verdad?)

\* Los registros índice: IX e IY. Se trata de dos registros de 16 bits. Esta es la versión oficial, la suministrada por Zilog; la realidad es que también pueden usarse como cuatro registros de 8 bits (IXh, IXl, IYh, IYl) mediante instrucciones ocultas, no documentadas por Zilog. ¿*A qué jugaban*? Vete a saber. Igual estaban borrachos, celebrando mi segundo aniversario... (¡pero hombre, no ves que es coña! ¡¡A ver si empeñas el hacha de una puta vez!!)

Estos dos registros están pensados para ser usados como punteros a memoria, pero tienen una característica de la que no disponen los pares BC, DE y HL: la **indexación**. *Explicome*: ya has visto que, con la instrucción adecuada, puedes cargar en el acumulador el contenido de la posición de memoria apuntada por HL, DE o BC: **LD** A,[HL], o **LD** A,[DE], o **LD** A,[BC]. Pues bien, existe la instrucción **LD** A,[IX+nn], donde nn es un desplazamiento (índice) comprendido entre **-128** y **127**; de esta forma puedes acceder fácilmente a una tabla de datos, simplemente cargando IX con la dirección de la base. Ejemplo:

**LD** IX, $1000 ; $ es el indicador de número hexadecimal.

; El punto y coma sirve para insertar comentarios en el listado ; (como el REM del BASIC)

**LD** A,[IX] ; Cargamos A con el contenido de $1000

**LD** B,[IX+2] ; Cargamos B con el contenido de $1002

**LD** H,[IX-1] ; Cargamos H con el contenido de $0FFF

**ADD** B ; Con esta instrucción y la siguiente realizamos la operación:

**SUB** H ; A = A + B - H

**LD** [IX+12], A ; Guardamos el resultado de la operación en $100C

Ni te has dado cuenta y has visto (y seguro que comprendido perfectamente) otro programa en ensamblador, esta vez algo más largo... no es difícil, ¿verdad? De paso hemos visto un par de instrucciones nuevas: **ADD** B, que suma (ADDition) A y B, dejando el resultado en A; y **SUB** H, que resta (SUBstraction) H de A, dejando también el resultado en A. "Pero... ¿y las banderas?" Dado que se trata de un ejemplo muy sencillo hemos pasado olímpicamente de ellas. "¿Y si ha habido desbordamiento?" Peor para nosotros; simplemente el resultado será incorrecto, pero el Z80 no explotará, ni nada parecido.

...Por supuesto, todo lo dicho para IX es igualmente válido para IY.

\* Los registros alternativos: cuentan que el ziloguero que diseñó el Z80 estaba borracho ese día. El caso es que el Z80 dispone de dos pares de registros AF, BC, DE y HL; se ve que al llegar a los indexados ya había dormido la mona. Para intentar disimular un poco el desaguisado organizó el Z80 de forma que sólo un juego de registros es visible en un mismo instante; para realizar el intercambio con el juego alternativo existen las instrucciones **EX** AF, AF' (que oculta AF y hace visible AF', su alter-ego) y **EXX** (que oculta BC, DE, HL y desenvaina los alternativos BC', DE', HL'). Por supuesto, volviendo a ejecutar **EX** AF, AF' y **EXX** recuperamos los registros originales.

Estos son to... estos son to... estos son todos los registros de propósito general, amigos. Veamos ahora los de uso específico.

\* EL PC. "¡Hereje!" ¡Tranquilo! PC significa Program Counter (contador de programa). Este registro contiene la dirección en memoria de la siguiente instrucción a ejecutar; cada vez que el Z80 lee uno de los bytes que forman la instrucción en curso incrementa este registro automáticamente. "¿Y si pasa de $FFFF...?" Pues mira por donde, tampoco en este caso el Z80 explota. Simplemente, vuelve a empezar por $0000.

\* El SP, o Stack Pointer (puntero de pila). La pila es una estructura de datos extremadamente útil a la hora de programar, e imprescindible en el proceso de ejecución de subrutinas. Trataremos la pila con más detalle en la siguiente sección.

14

\* El registro I, o registro de interrupción. Se utiliza únicamente con un tipo de interrupción que el MSX no usa, por lo que no volveremos a hablar de él.

\* Registro R o de refresco de memoria. La memoria principal del MSX es de tipo dinámico, es decir, necesita ser refrescada (leída y reescrita) continuamente para mantener la información. El Z80 se ayuda de este registro para realizar este proceso. En una demo de Anma salía un "starfield" (entramado de estrellas que se acercan) muy majo, y en el texto que simultáneamente aparecía explicaban que dicho efecto había sido programado usando el registro de refresco... me lo expliquen...

\* El registro de instrucción (IR, Instruction Register). Situado dentro de la unidad de control, este registro almacena el último byte de la instrucción actual leído, listo así para su descodificación. No nos interesa a efectos de programación.

¡Uf! Muchos registros, ¿verdad? La verdad es que no son tantos. Para que lo veas más claro he preparado un esquema en el que se agrupan los registros en forma de tabla, de forma que tengas una referencia con un golpe de vista (¡Ay! ¡Mis ojos...!)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | F | <- **EX** AF,AF' -> | A' | F' | Acumulador y Flags (banderas) |
| B | C |  | B' | C' |  |
| D | E | <- **EXX** -> | D' | E' |  |
| H | L |  | H' | L' |  |
| I | R | Interrupción y Refresco de Memoria |  |  |  |
| IX | | <- Con instrucciones -> | IXh | IXl | Registros de Indexación de 16 bits Ejemp: **LD** H,(IX+n) |
| IY | |  | IYh | IYl |
| SP | | Stack Pointer (puntero de pila) |  |  |  |
| PC | | Program Counter (contador de programa) |  |  |  |

Ahora está un poco más claro, ¿verdad? Una vez más puedo garantizarte y te garantizo que cuando lleves un mínimo tiempo programando te sabrás de memoria la tabla de registros. Y ahora, señoras y señores, la estrella es... ¡la pila!

**10 LA PILA** [**Índice**](#INDICE)



"¡Ah, sí, la pila!" ¿Te suena? "¡Sí, claro! ¿La del reloj o la de la calculadora?" *Estooo*... ¿me dejas el hacha un momentín? ¿No? Pues entonces calla, que aquí el único que suelta estupideces soy yo, y sigamos con lo nuestro.

Imagina que te has pasado la tarde jugando al Nemesis en vez de hacer los deberes, y mamá te ha castigado a fregar los platos hasta que *Guilian Seed\** despierte de su hibernación. ¿Qué? ¿Que eres ya un poco mayorcito? Bueno, pues retrocede mentalmente hasta tus años mozos e imagina que te has pasado la tarde jugando al Nemesis, versión ENIAC.

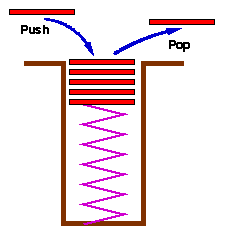
**\*** *Guilian Seed: Investigar quien es.*

Así, feliz, contento y motivado, empiezas tu apasionante tarea fregadora con una marcada expresión de odio en tu cara. Dejas reluciente el primer plato, y lo depositas en el escurridor. El segundo plato lo dejas sobre el primero, el tercero sobre el segundo, y así sucesivamente: estás formando una pila de platos.

De repente piensas en tu maldito destino; el castigo es tremendamente injusto (al fin y al cabo estabas salvando un planeta), y en un arrebato de rabia no se te ocurre otra cosa que romper platos. ¿Cuál es el primero que coges? El último que has fregado, que está sobre todos los demás, en la parte superior de la pila. Tu rabia continúa, y esta vez coges el siguiente plato, que es el penúltimo que habías liberado de materia roñosa.

Este arrebato te cuesta alargar el castigo hasta que James Burton libere Nemesis, pero tú te lo has buscado; lo que nos interesa ahora es la pila de platos. Si cambias éstos por datos, y el escurridor por la memoria, ya tenemos la pila del Z80. Se trata de una estructura de datos tipo **LIFO** ("Last In, First Out"), en la que, como su propio nombre proclama, el último en entrar es el primero en salir.

15

Para introducir datos en la pila disponemos de la instrucción **PUSH** rr ("empujar"), donde rr puede ser cualquiera de los pares AF, BC, DE, HL, IX, IY; esto significa que los datos se introducen en la pila de dos en dos bytes. Para sacar los datos existe la instrucción **POP** rr ("extraer"). El registro SP apunta, precisamente, al byte inferior del último dato introducido; cada vez que ejecutas **PUSH** o **POP** el Z80 actualiza automáticamente el puntero. Pero, ¿qué significa exactamente "actualizar el puntero"? En el Z80 la pila crece hacia abajo, es decir, hacia direcciones de memoria inferiores. Así, una instrucción **PUSH** implica un decremento de SP en dos unidades (recuerda que estamos introduciendo dos bytes), mientras que **POP** provoca el correspondiente incremento.

No pocas veces he visto esa expresión que ahora mismo tu cara refleja, y que me indica claramente que no has entendido nada de nada. Bueno, haz lo siguiente: léete otra vez el párrafo anterior, ahora más despacio, y luego observa este ejemplo que vale más que 34 millardos de palabras:

Supón que SP apunta a la dirección $1000. Representaremos un trozo de la memoria de la siguiente manera:

|  |  |  |
| --- | --- | --- |
|  | ???? | $0FFB |
|  | ???? | $0FFC |
|  | ???? | $0FFD |
|  | ???? | $0FFE |
|  | ???? | $0FFF |
| SP -> | ???? | $1000 |

Sí, he representado la memoria creciendo hacia abajo, al revés de lo que resultaría lógico. No, no se trata de otra de mis excentricidades: en cualquier texto que represente la memoria en forma esquemática verás el mismo sentido de crecimiento en las direcciones. Ignoro el motivo de esta convención, pero de todas formas nos resultará útil para comprender el funcionamiento de la pila, ya que de este modo sí responde al concepto intuitivo de pila creciendo hacia arriba, al menos gráficamente. "????" representa datos desconocidos para nosotros (en la memoria siempre hay datos, aunque no sepamos qué datos).

Bueno, tenemos SP apuntando a $1000, y ejecutamos lo siguiente:

**LD** HL, $1122

**PUSH** HL

La memoria y el puntero de pila quedan entonces de la siguiente forma:

|  |  |  |
| --- | --- | --- |
|  | ???? | $0FFB |
|  | ???? | $0FFC |
|  | ???? | $0FFD |
| SP -> | $22 | $0FFE |
|  | $11 | $0FFF |
|  | ???? | $1000 |

Ahora ejecutamos lo siguiente:

**LD** IX, $AABB

**PUSH** IX

Y en cuanto a la memoria...

|  |  |  |
| --- | --- | --- |
|  | ???? | $0FFB |
| SP -> | $BB | $0FFC |
|  | $AA | $0FFD |
|  | $22 | $0FFE |
|  | $11 | $0FFF |
|  | ???? | $1000 |

Ahora hacemos trampa:

**INC** SP

La instrucción **INC** sirve para incrementar cualquier registro o par de registros, incluido el SP.

Ahora tenemos la memoria tal que así:

|  |  |  |
| --- | --- | --- |
|  | ???? | $0FFB |
|  | $BB | $0FFC |
| SP -> | $AA | $0FFD |
|  | $22 | $0FFE |
|  | $11 | $0FFF |
|  | ???? | $1000 |

Y si ahora hacemos

**POP** BC

Ejercicio: intenta adivinar tú solo qué valor contendrá el par BC tras esta instrucción, y cómo quedará la memoria y el SP.

16

Solución: BC albergará un bonito $22AA en su interior, y la memoria quedará tal que así:

|  |  |  |
| --- | --- | --- |
|  | ???? | $0FFB |
|  | $BB | $0FFC |
|  | $AA | $0FFD |
|  | $22 | $0FFE |
| SP -> | $11 | $0FFF |
|  | ???? | $1000 |

Un detalle importante: la instrucción **PUSH** implica modificar la memoria mediante la introducción de dos bytes, pero **POP** no borra estos bytes; simplemente actualiza el par de registros implicado, y naturalmente SP, quedando los datos de la pila inalterados.

¿Ves cómo con ejemplos se entiende la gente? Ahora entiendes perfectamente el funcionamiento de la pila, y además, como en el fondo lo que estás haciendo es estudiar, mamá te ha levantado el castigo y ya no has de fregar.

Lo que tal vez no veas tan claro es la utilidad de la pila. Utilidad tiene, y mucha, te lo aseguro. La pila tiene dos usos principales: almacenamiento temporal de datos que no podemos perder pero están ocupando un registro que necesitamos para otros fines, y copia de la dirección de retorno de una subrutina.

Veamos un ejemplo de lo primero. Supón que en algún punto de tu programa necesitas negar el contenido de la posición de memoria $1974, pero tienes el acumulador ocupado con algún otro dato útil. Una solución es hacer lo siguiente:

**PUSH** AF

**LD** A,[$1974]

**NEG**

**LD** [$1974], A

**POP** AF

¿Lo has pillado? Es como si le dijeras a la pila "sostenme este dato un momento mientras niego este otro". Al terminar este trozo de programa el dato de $1974 ha sido negado, pero el par AF no se ha visto afectado.

Hablemos ahora de las subrutinas. Si recuerdas del BASIC, la instrucción **GOSUB** te permite ejecutar una subrutina (que no es más que otra porción de programa), volviendo al punto de partida con un simple **RETURN**. En ensamblador la instrucción de llamada es **CALL** dddd, donde dddd es la dirección de memoria en la que comienza la subrutina, y la instrucción de retorno es **RET**.

Cuando el Z80 ejecuta una subrutina necesita guardar en algún sitio la dirección en la que se ha realizado la llamada, para poder continuar con la ejecución del programa principal una vez se da la orden de retorno. Y este sitio es ni más ni menos que la pila. Así, la instrucción **CALL** dddd equivale a estas dos pseudoinstrucciones:

"**PUSH** PC"

"**LD** PC, dddd"

Estas instrucciones no existen, pero simbolizan lo que en realidad hace el Z80 cuando recibe una orden de llamada subrutina: guardar el contador de programa en la pila exactamente como hace la instrucción **PUSH** que he descrito antes, y cargarlo con la dirección de inicio de la subrutina, que empieza a ejecutarse inmediatamente.

Al final de dicha subrutina debe haber una instrucción **RET**, que equivale a:

"**POP** PC"

En efecto, el valor antiguo del PC es recuperado y el programa continúa desde la instrucción siguiente al **CALL**. Como ves, si no hubiera pila no habría subrutinas.

17

Una pregunta que debería haberte surgido: si la pila se usa para guardar datos temporales y también para guardar la dirección de retorno de las subrutinas, ¿no puede surgir algún conflicto entre ambas funciones? En otras palabras: dentro de una subrutina, ¿puede haber **PUSH** y **POP** sin que el Z80 se haga un lío con la pila? La respuesta es: sí, pero siguiendo una regla muy sencilla pero de muy fácil olvido (al menos en mi caso). Esta regla dice que dentro de una subrutina ha de haber el mismo número de **PUSH** que de **POP**, es decir, el puntero de pila ha de apuntar, al llegar al **RET**, exactamente a la misma dirección que apuntaba cuando la subrutina empezó tras el **CALL**. De no ser así, tras el **RET** el programa continuará en la primera dirección que encuentre en la pila, y aunque el Z80 tiene una gran resistencia y ni aun así explota, lo más probable es que el MSX se cuelgue cual windows con más de dos ventanas abiertas (*En las últimas versiones de Windows han arreglado esto, je, je*).

¿Qué más te podría contar de la pila? Existe la instrucción **LD** SP, rr donde rr puede ser HL, IX o IY, que te permite situar la pila en el lugar de la memoria que más te convenga; de hecho en el manual técnico del MSX se recomienda situar la pila en un lugar adecuado como primera acción a realizar en un programa hecho como ASCII manda. Yo debo ser un programador de los chapuceros, ya que nunca inicializo la pila; y de todas formas no es imprescindible, pues la plataforma de ejecución de un programa CM es siempre el DOS o el BASIC, entornos que ya se encargan de cargar el SP con un valor adecuado antes de pasar el control a tu programa. Pero ahí queda la recomendación de los expertos...

Y, bueno, esto es lo que hay con respecto a la pila. Si aún no te has convencido de su importancia tú tranquilo, ya me darás la razón a medida que cojas práctica programativa...

**11 INSTRUCCIONES DEL Z80** [**Índice**](#INDICE)

Pues sí, lo que vamos a ver ahora es una descripción, de algunas de las instrucciones del Z80. No incluyo toda la información, ya que no es imprescindible para programar y sólo serviría para liar más el asunto; al final están las tablas con todas las instrucciones, banderas, bytes y tiempos ir a [**TABLAS**](#TABLAS).

Supongo que no hace falta que te diga qué ocurrirá si intentas aprendértelas de memoria, pero para por si acaso te lo recuerdo: no lo conseguirás, te volverás loco, aprovecharé para *birlarte* el hacha y a partir de ahí puede pasar cualquier cosa. Así que ya sabes: tú lee tranquilamente, y con la práctica ya irás reteniendo.

Recuerda que hace algún tiempo te dije que el Z80 puede "tratar datos y alguna cosa más". Concretando un poco, las instrucciones del Z80 se pueden clasificar en cinco categorías:

- [Transferencia de datos](#INSTRUCCIONES_DE_TRANSFERENCIA_DE_DATOS).

- Tratamiento de datos, que a su vez se pueden clasificar en:

\* [Operaciones aritméticas](#INSTRUCCIONES_ARITMETICAS).

\* [Incremento y decremento](#INCREMENTO_Y_DECREMENTO).

\* [Operaciones lógicas](#OPERACIONES_LOGICAS).

\* [Rotación y desplazamiento](#ROTACION_Y_DESPLAZAMIENTO).

\* [Manipulación de bits](#MANIPULACION_DE_BITS).

- [Verificación y salto](#VERIFICACION_Y_SALTO).

- [Entrada y salida](#ENTRADA_SALIDA).

- [Control](#INSTRUCCIONES_DE_CONTROL).

"Pues cualquiera memoriza esto." ¿Qué has *dichooo*? "¡Ah! eh... nada, que Konami Man es un genio." Ah, creía...

Pues nada, como te veo ligeramente impaciente y ya no se me ocurren más paridas de relleno, vamos a hacer una visita a las...

**12 INSTRUCCIONES DE TRANSFERENCIA DE DATOS** [**Índice**](#INDICE)

Pues sí, una de las portentosas operaciones que nuestro genial Z80 es capaz de realizar es mover datos de un sitio a otro (bueno, en realidad no hay muchos tipos de "sitios": sólo memoria y registros).

18

Para empezar podemos cargar cualquier registro (o par de registros) con un dato inmediato (es decir, con un número puro y duro). Para ello tenemos las instrucciones:

**LD** r, n

**LD** rr, nn

donde "r" puede ser cualquiera de los registros A, B, C, D, E, H, L y "rr" cualquiera de los pares BC, DE, HL, SP, IX, IY. Como ya habrás adivinado, "n" ha de ser un dato de un byte, y "nn" uno de dos bytes. "¿Y no puedo cargar un par de registros con dato de un byte?" Claro que puedes: si ejecutas **LD** HL, 34, lo que harás es cargar L con 34 y H con 0, pero insisto en que el Z80 no explotará.

También puedes cargar cualquier registro con el contenido de otro registro:

**LD** r, r'

donde "r" y "r'" pueden ser A, B, C, D, E, H, L. Los pares de registros no son tan flexibles en este aspecto: sólo disponemos de

**LD** SP, rr

donde "rr" pueden ser únicamente HL, IX o IY.

"¿Y qué hay de la memoria?" A eso iba. Si queremos transferir un dato directo (número P. y D.) a memoria, lo tenemos difícil: sólo puedes hacer esto con datos de un byte, y con las instrucciones

**LD** [rr], n

donde "rr" pueden ser HL, IX+d o IY+d. Recuerda el significado de los paréntesis (indican "el contenido de la posición de memoria indicada en rr") y de "d" (un desplazamiento de un byte en complemento a dos, de **-128** a **127**).

Si lo que queremos indicar directamente es la dirección de memoria, para transferir datos de un byte sólo disponemos de una instrucción:

**LD** [nn], A

y para dos bytes,

**LD** [nn], rr

donde "nn" es una dirección de memoria (dos bytes), y "rr" puede ser BC, DE, HL, SP, IX, IY.

Si lo que queremos es usar registros para indicar tanto la dirección de memoria como el dato, podemos usar las instrucciones:

**LD** [rr], r'

donde "rr" puede ser HL, IX+d o IY+d y "r'", A, B, C, D, E, H o L.

Como siempre el acumulador es más chulo que nadie, y también existen las instrucciones

**LD** [BC], A

**LD** [DE], A

"Está bien esto de transferir datos a la memoria, pero... ¿y si lo que quiero es leer de la memoria?" Caramba, vas espabilando. Pues bien, la cosa no cambia mucho respecto de lo ya expuesto: las instrucciones disponibles para leer la memoria son

**LD** A,[nn]

**LD** r',[rr]

donde "r'" puede ser A, B, C, D, E, H, L y "rr" HL, IX+d o IY+d. Más:

**LD** A,[BC]

**LD** A,[DE]

Te estarás preguntando cómo funciona esto de las transferencia de dos bytes a memoria. Pues bien, justo al revés de como habías imaginado: primero se transfiere el byte bajo y después el alto. "¿*Lo cualo qué*?" Sí, que si haces

**LD** HL, $1234

**LD** [$1000], HL

conseguirás un $34 en $1000 y un $12 en $1001. Así de claro (y de raro).

19

¿Dónde vas? ¡Aún quedan instrucciones de transferencia de datos! Algunas son tan psicodélicas como

**LD** A, I

**LD** A, R

**LD** I, A

**LD** R, A

para acceder al registro de refresco y al de interrupción. Otras, como **PUSH**, **POP**, **EXX** y **EX** AF, AF' ya las conoces de sobras. Existe también una instrucción bastante curiosa:

**EX** [SP], rr

Que intercambia la parte superior de la pila (vamos, el dato que obtendrías con un **POP** cualquiera) con el par HL, IX o IY, talmente como si hicieras

"**POP** JK"

"**PUSH** rr"

"**LD** rr, JK"

donde JK sería una par de registros imaginario que haría de bufer... cualquier día uso esta instrucción, tiene su gracia. Y hablando de intercambios, también existe

**EX** DE, HL

cuya función no te voy a explicar porque soy así de malo y quiero que esta noche la pases en vela rompiéndote el coco. (Pista EXchange)



Las que vamos a ver con un poco más de calma son las de transferencia de bloques. "Este tío empieza a marearme, y yo ya tengo calambres en el brazo del hacha." Bueno, no opinarás lo mismo cuando compruebes que estas instrucciones pueden ahorrarte bastante trabajo a la hora de programar, así que tranquilo, y atiende. Las instrucciones de transferencia de bloques son las siguientes:

**LDI** ; Un solo dato, dirección de destino creciente.

**LDIR** ; Un bloque, dirección de destino creciente.

**LDD** ; Un solo dato, dirección de destino decreciente.

**LDDR** ; Un bloque, dirección de destino decreciente.

y con ellas puedes transferir de golpe un bloque de datos de una zona de la memoria a otra, de la forma más sencilla imposible. ¿Que no te lo crees? Pues mira, sólo tienes que hacer

**LD** HL, dir\_origen

**LD** DE, dir\_destino

**LD** BC, longitud\_del\_bloque

**LDIR**

¡¡ Y ya está el bloque transferido !! Te has quedado de piedra, ¿eh? Ya te dije que el Z80 es el invento del siglo.

Desmenucemos un poco más esta fantástica instrucción. Podríamos simular su comportamiento con este programa BASIC:

10 **POKE** HL,**PEEK**(DE)

20 HL=HL+1:DE=DE+1:BC=BC-1

30 **IF** BC=0 **THEN** **END** **ELSE** **GOTO** 10

¿Y qué pasa con las otras tres? **LDDR** se diferencia de **LDIR** en que el puntero de la dirección de destino se decrementa en lugar de incrementarse, con lo cual el bloque se transfiere invertido; vamos, que el programa BASIC simulador es el mismo, sólo que cambiando DE=DE+1 por DE=DE-1.

En cuanto a **LDI** y **LDD**, transfieren el dato y actualizan los punteros, pero a diferencia de **LDIR** y **LDDR** sólo transfieren un dato y no cierran el bucle. Sus equivalentes BASIC se obtendrían eliminando la línea 30 de los programas simuladores de **LDIR** (para **LDI**) y **LDDR** (para **LDD**).

20

Bueno, pues hemos acabado con las instrucciones de transferencia de datos. "¡Eh chavalín! ¿Qué pasa con las banderas?" ¡Huy perdón, es verdad! No puedo obviar el efecto de las instrucciones de carga sobre las banderas.

Las únicas instrucciones de carga "normales" (tipo **LD**) que modifican las banderas son **LD** A, R y **LD** A, I, que no nos deben preocupar porque no las usaremos. Obviamente, también modifican las banderas **EX** AF, AF' y **POP** AF. En cuanto a las instrucciones de transferencia de bloques, siempre ponen a cero las banderas H y N. **LDIR** y **LDDR** ponen la bandera P/V siempre a cero; **LDD** y **LDI** la ponen a cero si BC=0 tras la ejecución, en caso contrario la ponen a uno. Y ya está, como ves la familia **LD**ez no da mucho trabajo con esto de las banderas.

Pues esto es todo en cuanto a instrucciones de transferencia de datos. ¿Demasiadas? Bueno, ¿y para qué están las tablas? ¿Eh? Pues ¡tablas van!

TABLA DE INSTRUCCIONES DE TRANSFERENCIA DE DATOS "LD" (1 BYTE)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Orig,** | **A** | **B** | **C** | **D** | **E** | **H** | **L** | **[HL]** | **[BC]** | **[DE]** | **[IX+d]** | **[IY+d]** | **[nn]** | **n** |
| **A** | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| **B** | x | x | x | x | x | x | x | x |  |  | x | x |  | x |
| **C** | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| **D** | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| **E** | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| **H** | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| **L** | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| **[HL]** | x | x | x | x | x | x | x |  |  |  |  |  |  |  |
| **[BC]** | x |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **[DE]** | x |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **[IX+d]** | x | x | x | x | x | x | x |  |  |  |  |  |  |  |
| **[IY+d]** | x | x | x | x | x | x | x |  |  |  |  |  |  |  |
| **[nn]** | x |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **Dest.** |  | | | | | | | | | | | | | |

TABLA DE INSTRUCCIONES DE TRANSFERENCIA DE DATOS "LD" (2 BYTES)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Orig.** | **BC** | **DE** | **HL** | **SP** | **IX** | **IY** | **nn** | **[nn]** |
| **BC** |  |  |  |  |  |  | x | x |
| **DE** |  |  |  |  |  |  | x | x |
| **HL** |  |  |  |  |  |  | x | x |
| **SP** |  |  | x | x | x | x | x | x |
| **IX** |  |  |  |  |  |  | x | x |
| **IY** |  |  |  |  |  |  | x | x |
| **[nn]** | x | x | x | x | x | x |  |  |
| **Dest.** |  | | | | | | | |

Simplemente recuerda que el formato es **LD** Destino, Origen; si ves una "x" significa que la instrucción existe para ese origen y ese destino. Como ves me he saltado intencionadamente los registros I y R, para nuestros propósitos es más inútiles que la primera rebanada del pan de molde. Para completar la tabla, ahí van el resto de instrucciones de carga:

**PUSH** rr

**POP** rr -> "rr" = AF, BC, DE, HL, IX, IY

**EX** AF, AF'

**EXX**

**EX** [SP], HL

**EX** [SP], IX

**EX** [SP], IY

**EX** DE, HL

**LDI**

**LDIR**

**LDD**

**LDDR**

21

**13 INSTRUCCIONES ARITMÉTICAS** [**Índice**](#INDICE)

¡Yupi! ¡Por fin! Ya echaba de menos las matemáticas. Eestooo... por supuesto que no hablaba en serio. Te lo juro por el hacha que en estos momentos sostienes a escasa distancia de mi cabeza.

En realidad poco hay que decir en esta sección, pues el Z80 es un poco tonto; las únicas operaciones matemáticas que el pobre conoce son la suma y la resta. Así pues, disponemos de únicamente cuatro instrucciones: **ADD** (suma), **ADC** (suma con acarreo), **SUB** (resta) y **SBC** (resta con acarreo).

En las operaciones con un byte se suma (o resta) siempre el acumulador más (menos) el segundo operando (otro registro, un dato inmediato o [HL], [IX+d], [IY+d], y se deja el resultado en el acumulador. Dicho de otro modo (modo tabla, claro):

|  |  |  |
| --- | --- | --- |
| **Instrucción** | **Operación (CY = bandera de acarreo)** |  |
| **ADD** A, r | A = A + r | "r" = A, B, C, D, E, H, L |
| **ADD** A,[HL] | A = A + [HL] |  |
| **ADD** A,[IX+d] | A = A + [IX+d] |  |
| **ADD** A,[IY+d] | A = A + [IY+d] |  |
| **ADD** A, n | A = A + n |  |

|  |  |  |
| --- | --- | --- |
| **Instrucción** | **Operación (CY = bandera de acarreo)** |  |
| **ADC** A, r | A = A + r + CY | "r" = A, B, C, D, E, H, L |
| **ADC** A,[HL] | A = A + [HL] + CY |  |
| **ADC** A,[IX+d] | A = A + [IX+d] + CY |  |
| **ADC** A,[IY+d] | A = A + [IY+d] + CY |  |
| **ADC** A, n | A = A + n + CY |  |

|  |  |  |
| --- | --- | --- |
| **Instrucción** | **Operación (CY = bandera de acarreo)** |  |
| **SUB** r | A = A - r | "r" = A, B, C, D, E, H, L |
| **SUB** [HL] | A = A – [HL] |  |
| **SUB** [IX+d] | A = A – [IX+d] |  |
| **SUB** [IY+d] | A = A – [IY+d] |  |
| **SUB** n | A = A - n |  |

|  |  |  |
| --- | --- | --- |
| **Instrucción** | **Operación (CY = bandera de acarreo)** |  |
| **SBC** A, r | A = A - r - CY | "r" = A, B, C, D, E, H, L |
| **SBC** A,[HL] | A = A – [HL] - CY |  |
| **SBC** A,[IX+d] | A = A – [IX+d] - CY |  |
| **SBC** A,[IY+d] | A = A – [IY+d] - CY |  |
| **SBC** A, n | A = A - n - CY |  |

En cuanto a las operaciones de dos bytes, podemos sumar con y sin acarreo, pero sólo disponemos de resta sin acarreo (esto no es demasiado trágico: basta poner a cero el acarreo antes de restar. Retomaremos el asunto en la sección de operaciones lógicas). Mejor pasamos directamente a la consabida tabla:

|  |  |  |
| --- | --- | --- |
| **Instrucción** | **Operación** |  |
| **ADD** HL, rr | HL = HL + rr | \ |
| **ADD** IX, rr | IX = IX + rr | -> "rr" = BC, DE, SP |
| **ADD** IY, rr | IY = IY + rr | / |
| **ADD** HL, HL | HL = HL x 2 |  |
| **ADD** IX, IX | IX = IX x 2 |  |
| **ADD** IY, IY | IY = IY x 2 |  |
| **ADC** HL, rr | HL = HL + rr + CY | "rr" = HL, DE, BC, SP |
| **SBC** HL, rr | HL = HL - rr - CY | "rr" = HL, DE, BC, SP |

Como ves, el par HL tiene la patente de las operaciones de dos bytes con acarreo... vaaale, la próxima vez que suelte una parida me aseguraré de que es mínimamente graciosa. Te preguntarás de qué sirve sumar y restar el bit del acarreo... bien, recuerda la suma y resta de números binarios que vimos en la primera entrega. "¡¡NOOOOOO!!" Sí. Recuérdala. "Argh... ya está (*esta me la pagas*)."

22

Bien, ahora dime cómo harías para sumar dos números de 32 bits, supongamos que contenidos en HL-IX y IY-BC.

¿Ya? Si lo has conseguido, pues enhorabuena, tómate algo. Si no, no te rompas más el coco, la solución es bien sencilla:

**ADD** IX, BC ; Sumamos las partes bajas sin acarreo. IX = IX + BC

**ADC** HL, IY ; Sumamos las partes altas con acarreo. HL = HL + IY + CY

... y ya tenemos el resultado de la suma en HL-IX. Sí: la suma de los 16 bits bajos es independiente de la suma de los 16 bits altos, excepto por la posible generación de un acarreo del bit 15 al 16, acarreo que es recogido por CY y usado por **ADC**. Y lo mismo, más o menos, pasa con la resta.

Más cosas. Habrás observado que, aunque el Z80 no sabe multiplicar, podemos de hecho multiplicar por dos (y, por tanto, por cualquier potencia de dos) cualquier número de uno o dos bytes, mediante la autosuma (sí, me he inventado la palabreja, ¿qué pasa?). Por ejemplo, para multiplicar HL por cuatro:

**ADD** HL, HL

**ADD** HL, HL

Vamos a rizar el rizo: ¡¡Multiplicación de HL por siete!! "¡Estás loco! ¡No se puede!" Ah, ¿no? Pues échale un vistazo a este programilla:

**PUSH** HL ; Metemos HL en la pila

**POP** DE ; Sacamos HL de la pila DE = HL

**ADD** HL, HL ; \*= 2 -> \*= 2

**ADD** HL, HL ; \*= 2 -> \*= 4

**ADD** HL, DE ; += HL -> \*= 5

**ADD** HL, DE ; += HL -> \*= 6

**ADD** HL, DE ; += HL -> \*= 7

¡¡Y una vez más, queda demostrado que Konami Man es un genioooo!! ¡CHAS! ¡AH! De acuerdo, vale, me he pasado, pero si no llego a esquivar el hachazo, ahora existirían Konami y Man por separado, so bestia...



Seguro que has comprendido el programa, pero como hoy tengo ganas de teclear ahí va la explicación: Se introduce la cabeza por la abertura, se ajusta el chaleco con la correa, y se infla tirando fuertemente de los pasadores, pero NUNCA dentro del avión... un momento... creo que no era esta explicación (tanto viaje a Barcelona me va a dejar *gilipenes*). Bueno, que copio el contenido de HL en DE (como ves, he aquí una nueva utilidad de la pila), multiplico HL por cuatro (por dos y otra vez por dos) y le sumo tres veces el valor que tenía antes: ya hemos multiplicado por siete. Para terminar recuerda que también existe la instrucción **ADD** A, A para autosumas de un byte.

Me dejaba en el tintero tres instrucciones que no sé si se pueden clasificar exactamente como aritméticas, pero como tengo que ponerlas en algún sitio ahí van: se trata de **CPL** (invierte el acumulador, cambiando todos los 0 por 1 y viceversa), **NEG** (realiza el complemento a dos del acumulador, es decir, lo niega y le suma uno), y **DAA**, que realiza el ajuste del resultado una vez realizada una operación en formato BCD. "¿Becequéeeee?" Es un formato de codificación de números decimales del que ya hablaremos si nos hace falta, tú tranqui.

Bueno, pues hasta aquí las operaciones aritméticas; podemos pasar a otra cosa. "¡Quieto parao! ¿Y las banderas?" Está visto que no se te puede engañar. Pues bien, salvo tres excepciones, todas las operaciones de suma y resta modifican todas las banderas: S, Signo (según el ídem del resultado); Z, cero (a uno si el resultado es ídem); H, acarreo mitad (el criterio cambia según la instrucción, no vale la pena preocuparse); P/V, desbordamiento (si la operación produce un ídem); N, resta (0 tras una suma, 1 tras una ídem); y C, acarreo (si se produce un ídem).

¡Más banderas, es la guerra!: **CPL** pone a uno H y N; **NEG** activa C si A era cero antes de la instrucción, activa P/V si A era $80, siempre activa N y modifica el resto de las banderas de la forma usual; y en cuanto a **DAA**, por el momento la vamos a dejar más abandonada que aquel juego de cinta que heredaste de tu abuelo.

23

"¿Y las tres excepciones de las que hablabas?" Son **ADD** HL, rr, **ADD** IX, rr y la **ADD** IY, rr: modifican H, N y C pero no tocan S, Z ni P/V (sigo pensando que los diseñadores del Z80 estaban un pelín bebidos...) Tú tranquilo, recuerda que en caso de emergencia siempre puedes poner el acarreo a cero y usar **ADC**, que sí banderiza como toca.

Bueno, pues eso, que las instrucciones aritméticas ya están listas (vaya, cuantas líneas... y eso que dije que no había mucho que decir sobre ellas... menudo cubrimiento de gloria...)

**14 INCREMENTO Y DECREMENTO** [**Índice**](#INDICE)

Estas instrucciones no están clasificadas como aritméticas, pero casi que lo son... bueno, dejemos de filosofar y al grano (digo, al byte). Las instrucciones **INC** y **DEC** nos permiten incrementar y decrementar (sumar/restar uno), respectivamente, cualquier registro o par de registros, o incluso el contenido de cualquier dirección de memoria, siempre que la indiquemos convenientemente.

Pues eso, que disponemos de las instrucciones de incremento y decremento de un byte:

**INC** r

**DEC** r

donde "r" = A, B, C, D, E, H, L, [HL], [IX+d], [IY+d]; y las de dos bytes

**INC** rr

**DEC** rr

donde "rr" = HL, BC, DE, SP, IX, IY.

Poca cosa queda por decir sobre estas instrucciones, a no ser alguna tontería sin importancia; por decir algo, el efecto sobre las banderas. Por decir algo, **INC** r y **DEC** r modifican S, Z, H, P/V y N según el resultado (P/V actúa como bandera de desbordamiento, N se pone a 0 para **INC** y a 1 para **DEC**), mientras que **INC** rr y **DEC** rr no modifican ninguna (misterios del silicio). Bueno, lo dejo aquí porque el SaveR\* me dice que ha hecho una música con el OPL4\*\* y quiere torturarme un rato (como una puta cabra...)

**\*** *SaveR Ni Puta Idea, supongo que el típico compi cansino que te viene a casa.*

**\*\*** *chip Yamaha YMF278 OPL4, capaz de trabajar con 18 canales de sonido FM, así como 24 canales de 12 y síntesis con samples de 16 bits.*

**15 OPERACIONES LÓGICAS** [**Índice**](#INDICE)

Desde luego, la música del SaveR era claustrofóbica a más no poder... venga, ya está bien de secta. ¿Recuerdas las cuatro operaciones lógicas **NOT**, **OR**, **AND** y **XOR**? Pues bien, el Z80 tiene a nuestra disposición cuatro instrucciones que realizan estas operaciones: **CPL**, **OR**, **AND** y **XOR**, casualmente. Excepto **CPL**, de la que ya hemos hablado, la operación se realiza entre el acumulador y un operando que hemos de especificar, quedando el resultado final en el acumulador. "Pero si las operaciones lógicas se realizan entre dos bits, y el acumulador tiene ocho..." Cierto: la operación se realiza bit a bit, con cada uno de los ocho del acumulador y del operando. "¿Eins?" Ejemplo: **OR** B. Lo que esta instrucción hace es lo siguiente:

bit 0 de A = bit 0 de A OR bit 0 de B

bit 1 de A = bit 1 de A OR bit 1 de B

...

bit 7 de A = bit 7 de A OR bit 7 de B

Así de fácil: con una sola instrucción realizamos ocho operaciones lógicas. La única que no necesita segundo operando es **CPL**, que como ya vimos invierte todos los bits del acumulador. Es decir, que tras un **CPL** lo que el Z80 hace es:

bit 0 de A = NOT bit 0 de A

...

bit 7 de A = NOT bit 7 de A

24

Concretando un poco, las instrucciones de las que disponemos son:

**CPL**

**OR** r

**AND** r

**XOR** r

donde "r" = A, B, C, D, E, H, L, [HL], [IX+d], [IY+d] o un dato inmediato (lo que en mi pueblo se llama número de un byte).

Las operaciones lógicas con un dato inmediato tienen truco. No se lo digas a nadie, pero permiten poner a uno o a cero bits sueltos del acumulador, con un par de golpes de reloj (*uséase* con una instrucción). Ejemplo: supón que te da el *venazo* de poner a uno los bits 1 y 3 del acumulador, sin tocar los otros. Pues bien, no tiene más que hacer:

**OR** 00001010b

¡Sorpresa! En realidad no hay tal truco; si repasas la tabla de operaciones lógicas, verás que 0 OR lokesea = lokesea, y 1 OR lokesea = 1.

¿Y si ahora quisieras poner a 0 los bits 7 y 5? Observa y pásmate:

**AND** 01011111b

El colmo de las virguerías: ¡¡Vamos a invertir los bits 0 y 6!! Chupao, hombre, chupao:

**XOR** 01000001b

Estarás pensando que soy un tipo un poco *raro* si encuentro placer en modificar bits del acumulador... bueno, cuando empieces a programar serás consciente de la importancia que puede llegar a tener un triste bit.

¡Ya tardábamos en hablar de las banderas! Las instrucciones lógicas (**CPL** aparte) modifican S, Z y P/V (que actúa como bit de paridad), y ponen a 0 N y C; **OR** y **XOR** ponen H a 0, y **AND** la pone a 1.

Llegamos a otro *truqui*: ¿qué pasa si hacemos **AND** A o **OR** A? "Nada, pues lokesea **AND**/**OR** elmismolokesea = eselokesea". En efecto, el acumulador no cambia, pero sí las banderas, nunca lo olvides. De hecho, **OR** A y **AND** A se utilizan siempre que se quiere poner a cero la bandera de acarreo.

Un último *truqui*: **XOR** A, que si te fijas deja el acumulador al más mísero de los ceros. **XOR** A es usado normalmente para borrar el acumulador, ya que esta instrucción ocupa un byte y **LD** A, 0 ocupa dos. "Entonces, ¿**XOR** A y **LD** A, 0 tienen un efecto idéntico?" ¡¡NO!! Recuerda las banderas...

Ya está aquí el SaveR otra vez. "¡¡Que me pongas el OPL4, que me vi a hacer una música pero ya!!" Pos bueno, hasta luego...

**16 ROTACIÓN Y DESPLAZAMIENTO** [**Índice**](#INDICE)

¡El SaveR se ha largado a estudiar japonés! ¡¡Aprovechemos!!

Llegamos a un conjunto de instrucciones bastante curioso, que nos permitirá tratar los registros como si de vulgares rompecabezas se trataran.

Imagínate que, un buen día, el bit 7 del acumulador se pone cachondo, y le entra la perra de ocupar el lugar del bit 6. Como es más burro que el programador medio de PC (que cuando un programa le sale lento, en vez de optimizarlo se compra un *pestium* más rápido), le arrea a su vecino un empujón de los bastos, tras lo cual consigue su propósito y se instala cómodamente en la posición del bit 6. Este, por su parte, se ve obligado a desalojar al bit 5, ya que en alguna parte se ha de quedar. El bit 5 se traslada a su vez a la posición 4... y así hasta llegar al bit 1, que se coloca en la posición del bit 0. ¿Que qué hace el *cerésimo* bit? Según su estado de ánimo: puede mudarse al acarreo o puede trasladarse a la posición del bit 7. En cuanto a la posición 7, que queda libre, puede ser "okupada" por el bit 0, por el contenido del acarreo, o quedar vacante (en cuyo caso se carga con un 0).

25

Pues en esto consisten las operaciones de rotación y desplazamiento, en *idemciones* e *idemzamientos* de todos los bits de un registro. La diferencia entre ambas estriba en el comportamiento del último bit (el 0 para movimientos hacia la derecha, el 7 para movimientos hacia la izquierda). A su vez, los desplazamientos se dividen en lógicos y aritméticos; las rotaciones pueden ser a través del acarreo y con copia en el acarreo.

"Perdone, me he equivocado... yo buscaba el curso de ensamblador F-A-C-I-L." ¡¡Eh, eh, un momento!! ¡Que es muy fácil! "¿Seguro?" ¡Claro! Comprendo que una parrafada teórica siempre queda bastante pastosa, pero ya deberías conocer mi estilo: tras el *rollazo*, ¡siempre vienen las tablas! Conque he aquí una tabla (aunque en esta ocasión mejor denominarla lista) de las instrucciones de *rotazamiento* (palabreja que me viene de perlas para abreviar):

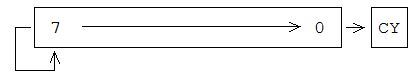
- Desplazamiento aritmético a la izquierda: **SLA** r



El bit 7 pasa al acarreo. El bit 0 se carga con un 0.

Equivale a la operación aritmética r = r\*2.

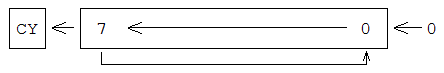
- Desplazamiento aritmético a la derecha: **SRA** x



El bit 7 no varía. El bit 0 pasa al acarreo.

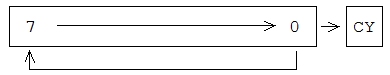
Equivale a la operación aritmética r = r/2, respetando el bit de signo.

- Rotación con copia en el acarreo hacia la izquierda: **RLC** r



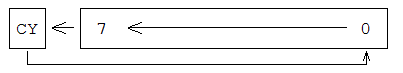
El bit 7 se copia en el acarreo y pasa a la posición 0.

- Rotación con copia en el acarreo hacia la derecha: **RRC** r



El bit 0 se copia en el acarreo y pasa a la posición 7.

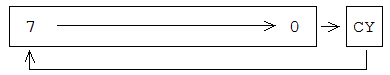
- Rotación a través del acarreo hacia la izquierda: **RL** r



El contenido del acarreo antes de la instrucción pasa al bit 0.

El bit 7 pasa al acarreo.

- Rotación a través del acarreo hacia la derecha: **RR** l



El contenido del acarreo antes de la instrucción pasa al bit 7.

El bit 0 pasa al acarreo.

¿Ves como no era tan difícil? Puede que ahora mismo no veas la utilidad de estas instrucciones. "Sí, llenar páginas de Easymbler". Je, je, qué gracia, me parto de ídem. Bueno, imagínate que quieres multiplicar/dividir el contenido de algún registro por una potencia de dos, o comprobar el contenido de todos los bits de un registro con un solo bucle... anda, parece que estas instrucciones sirven para esto, ¿verdad? Qué curioso...

26

¡¡Eh, eh, quieto parao, que no hemos terminado con las instrucciones de *rotazamiento*!! Concretamente, me olvidaba de cuatro detallitos.

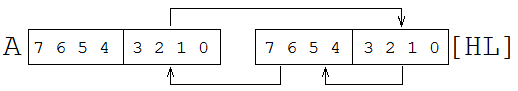
Uno: 'r' puede ser A, B, C, D, E, H, L, [HL], [IX+n] o [IY+n] para todas las *rotazacciones* (¡Toma compresión de "instrucciones de rotación y desplazamiento"!)

Dos: el comportamiento de las banderas. Todas las *rotazacciones* modifican todas las banderas (P/V actúa como bit de paridad, H y N se ponen siempre a 0).

Tres: existen unas instrucciones de rotación exclusivas del acumulador, heredadas del abuelito 8080. Se trata de **RLA**, **RRA**, **RLCA** y **RRCA**; tienen la ventaja de ocupar un sólo byte, y sólo se diferencian de sus equivalentes generales en el efecto sobre las banderas (no modifican S, Z ni P/V).

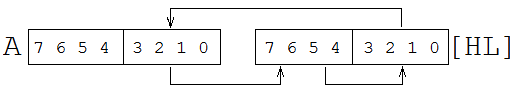
Cuatro: hay dos instrucciones especiales para rotación decimal. Estas instrucciones trabajan sobre A y [HL], y tratan los bytes como grupos de dos nibbles (normalmente, dos dígitos BCD). Ahí van:

- Rotación decimal a la izquierda: **RLD**



El nibble inferior de A pasa al nibble inferior de [HL], éste al nibble superior de [HL], y éste al nibble inferior de A. El nibble superior de A no se ve alterado.

- Rotación decimal a la derecha: **RRD**



El nibble inferior de A pasa al nibble superior de [HL], éste al nibble inferior de [HL], y éste al nibble inferior de A. El nibble superior de A no se ve afectado.

Ya queda poco... el efecto sobre las banderas es el mismo que el de las instrucciones normales, salvo que el acarreo no se ve afectado. Y ahora sí que hemos terminado con las *rotazacciones* (Sayonara... beibi.)

**17 MANIPULACION DE BITS** [**Índice**](#INDICE)

Manque te parezca increíble (o manque no), este es el último subgrupo de las llamadas operaciones de tratamiento de datos. Como su propio nombre grita (tan alto que algún vecino se va a mosquear y va a llamar a la Policía *Renderizada*), con estas instrucciones puedes manipular bit sueltos de cualquier registro. ¿Y qué es exactamente "manipular"? Pues lo que se conoce como "las tres pes": Poner a 0, Poner a 1 y Ponerse a mirar qué hay (bueno, la tercera operación se suele abreviar con "comprobar" o "verificar", pero entonces no me cuadraba lo de "las tres pes", y no me salía el chiste malo de turno).

Para empezar, supongamos que estamos en plan chafardero, y queremos ver qué hay en el bit 'b' (de 0 a 7) del registro 'r' (que puede ser A, B, C, D, E, H, L, [HL], [IX+d] o [IY+d]). Pues nada, no tenemos más que usar la instrucción

**BIT** b, r

... y tendremos la bandera Z convertida en espía (Z=1 si el bit es 0, Z=0 si el bit es 1). En cuanto al resto de las banderas, H se pone a 1, N a 0, y S y P/V varían aleatoriamente.

Si lo que queremos es pasar a la acción y modificar a nuestro antojo el bit 'b' del registro 'r' (los mismos que **SET**), usaremos

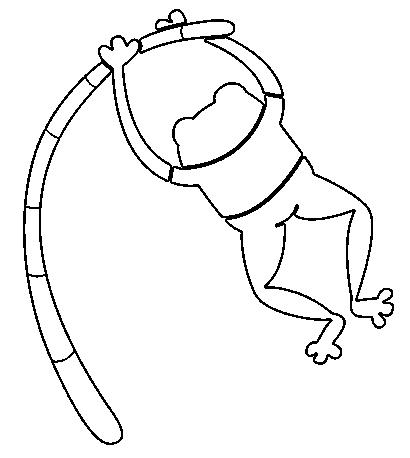
**SET** b, r ; pone el bit a 1

**RES** b, r ; pone el bit a 0

27

Estas dos instrucciones no actúan sobre las banderas. ¡Caramba!, me dejaba de un par de instrucciones, se trata de **SCF** (que pone el acarreo a 1) y **CCF** (que complementa el acarreo). Pues bueno, ahí quedan, y ya podemos pasar tranquilamente al siguiente grupo...

**18 VERIFICACIÓN Y SALTO** [**Índice**](#INDICE)



Vaya, vaya, mira por donde esto no nos pilla totalmente por sorpresa. Sí: ya habíamos visto una instrucción de salto. "¡¿Qué?! ¿Dónde?" No disimules, en la sección sobre la pila ya habíamos visto la instrucción **CALL**. Se trataba de una instrucción de llamada incondicional. También vimos **RET**, retorno incondicional.

Las instrucciones de este grupo se dividen en salto absoluto, salto relativo, llamada y retorno; a su vez, cada una de éstas puede ser incondicional (la instrucción se ejecuta "by noses") o condicional (la instrucción se ejecuta sólo si se cumple una determinada condición, es decir, si determinada bandera está como nosotros queremos que esté). ¿Recuerdas la charla que tuvimos sobre las banderas? Quedó claro (o debió quedar claro) que las banderas juegan un papel fundamental en las instrucciones condicionales, equivalentes a **IF**...**GOTO**. Pues bien, sin más ni more paso a describir las instrucciones de salto y comprobarás que tal afirmación no fue gratuita (son 34 millardos, por si le place). Ahí van:

- Salto absoluto: **JP** aa. El PC se carga con la dirección 'aa' y la ejecución continúa. Vamos, como un vulgar **GOTO**. También existen JP [HL], JP [IX] y JP [IY].

- Salto absoluto condicional: JP cc, aa. El PC se carga con 'aa' sólo si se cumple la condición 'cc', que puede ser:

Z: cero (Z=1)

NZ: no cero (Z=0)

C: acarreo (C=1)

NC: no acarreo (C=0)

PE: paridad par (P/V=1)

PO: paridad impar (P/V=0)

P: positivo (S=0)

M: negativo (S=1)

Como ves, equivale a **IF** cc **GOTO** aa.

- Salto relativo: **JR** d. Al contador de programa se suma el desplazamiento 'd' (un byte en complemento a dos). El valor del PC se toma tras la instrucción, que ocupa dos bytes; por tanto 'd' puede ir de **-126** a **+129**. No te rompas el coco con estos detalles, a la hora de ensamblar usarás etiquetas, que te ahorrarán cualquier cálculo. Dale paciencia a la ídem.

El salto relativo tiene varias ventajas respecto al absoluto: es más corto, y permite realizar programas reubicables (si un programa sólo contiene saltos relativos, da igual si empieza en $0034 o en $AFBE). La desventaja es el insuficiente y limitado alcance (de **-126** a **+129** bytes).

- Salto relativo condicional: **JR** cc, d. El salto relativo sólo se produce si se cumple la condición, que en esta ocasión sólo puede ser C, NC, Z o NZ.

- La instrucción **CP r**, compara A con r y modifica las banderas en función del resultado:

Z=0 y C=1 si A < r

Z=1 y C=0 si A = r

Z=0 y C=0 si A > r

Pues con **CP** y los saltos condicionales podemos componer instrucciones del tipo **IF** A<B **GOTO**... Fantástico, ¿verdad?

- Llamada a subrutina: **CALL** aa. Esta instrucción quedó ampliamente explicada en la sección sobre la pila, así que no me enrollaré aquí sobre ella.

28

- Llamada condicional: **CALL** cc, aa. La llamada se realiza si se cumple la condición, que puede ser cualquiera de las expuestas para **JP**. Equivalente a **IF**...**GOSUB**.

- Retorno de subrutina: **RET**. Idem de lo dicho para **CALL**.

- Retorno condicional: **RET** cc. El valor del PC es restaurado de la pila, sólo si se cumple la condición (puede ser cualquiera de las usables en **JP** y **CALL**).

Ahora veremos algunas instrucciones especiales. Una bastante interesante es **DJNZ** d: primero decrementa el registro B, después comprueba si ha llegado a cero; en caso contrario realiza un salto relativo de longitud 'd' (igual que **JR**). O más BASICamente,

B=B-1:**IF** B<>0 **THEN GOTO** PC+d

Vamos, que ni hecha a propósito (de hecho, hecha a propósito) para componer bucles.

Un par de instrucciones rarillas son **RETI** y **RETN**, usadas para volver de una rutina de atención a una interrupción, enmascarable y no enmascarable respectivamente. Como el MSX no usa interrupciones no enmascarables, y las enmascarables las gestiona normalmente el sistema operativo, no entraremos en detalles.

Esta es bastante curiosa: **RST** a. Equivale a **CALL** a. ¿La ventaja? Ocupa un sólo byte. ¿La desventaja? 'a' sólo puede ser $00, $08, &10, $18, $20, $28, $30 y $38. Caprichitos de los zilogueros...

Para acabar con este grupo, una frase histórica que dijo no sé quién: "Las instrucciones de verificación y salto no modifican ninguna bandera". Si la miras bien es bastante profunda...

**19 ENTRADA/SALIDA** [**Índice**](#INDICE)

Seguramente habrás oído hablar de los puertos de entrada/salida, a través de los cuales el Z80 puede intercambiar datos con los diferentes periféricos del MSX (el VDP, el PSG, el PPI y compañía). También recordarás que dije algún tiempo algo así como "cualquier periférico conectable a un MSX puede ser contemplado como memoria". Vamos a ver esto con un poco más de calma.

Vimos que el Z80 puede direccionar 64K de memoria, a base de instrucciones LD y similares. Pues bien, el Z80 también dispone de un segundo espacio de direccionamiento, llamado espacio de entrada/salida. Este es de 8 bits (es decir, "mide" únicamente 256 direcciones, o "puertos") y dispone de instrucciones específicas para su control. Normalmente se usa el espacio "normal" para direccionar memoria y el espacio de entrada/salida para controlar periféricos, aunque nada impide que sea al revés (por ejemplo, el SCC se controla a través del espacio normal de direcciones del cartucho que lo contiene). De hecho, en el MSX la mayoría de los 256 puertos del Z80 ya están reservados para periféricos estándar, y lo más seguro a la hora de diseñar uno nuevo es controlarlo a través del espacio de memoria del cartucho controlador.

Resumiendo, que además de 64K de memoria, el Z80 puede direccionar 256 puertos de entrada/salida. En primer lugar veremos las instrucciones de entrada:

**IN** A,[n]

**IN** r,[C] -> 'r' = A, B, C, D, E, H, L

**INI**

**INIR**

**IND**

**INDR**

La primera instrucción lee el puerto indicado de forma inmedianta mediante el número 'n', depositándolo en A. La segunda deposita en el registro 'r' el contenido del puerto indicado en el registro C. ¡¡OJO!! **IN** A,[n] no modifica ninguna bandera, pero **IN** r,[C] modifica S, Z, H, P/V (paridad) y N (siempre a 0).

29

Las instrucciones **INI**, **INIR**, **IND** e **INDR** son similares a **LDI**, **LDIR**, **LDD** y **LDDR**: permiten la entrada de bloques de datos con pocas instrucciones. En concreto, **INIR** equivale a:

10 **POKE** HL, **INP**(C)

20 HL=HL+1:B=B-1

30 **IF** B=0 **THEN END ELSE GOTO** 10

Si eliminamos la línea 30 tenemos **INI**; si cambiamos HL=HL+1 por HL=HL-1 tenemos **INDR**; y si hacemos las dos cosas tenemos **IND**. Observa que siempre leemos el mismo puerto (el indicado en el registro C), y que sólo podemos leer un máximo de 256 datos (el bucle es controlado por el registro B).

En cuanto a las banderas, S, Z y P/V se modifican aleatoriamente; N se pone a 1, y Z se pone 1 si B=0 tras la ejecución (cosa que siempre pasa con **INIR** e **INDR**).

¿Y la salida? Para escribir en los puertos disponemos de:

**OUT** [n], A

**OUT** [C], r -> 'r' = A, B, C, D, E, H, L

**OUTI**

**OTIR**

**OUTD**

**OTDR**

Ni **OUT** [n], A ni **OUT** [C], r modifican las banderas. **OUTI**, **OTIR**, **OUTD** y **OTDR** son equivalentes a **INI**, **INIR**, **IND** e **INDR** (el efecto sobre las banderas es idéntico), basta cambiar en el programa BASIC equivalente la primera línea:

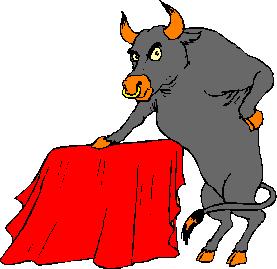
10 **OUT**(C), **PEEK**(HL) <-- 10 **POKE** HL, **INP**(C)

20 HL=HL+1:B=B-1

30 **IF** B=0 **THEN END ELSE GOTO** 10

Y ahora una frase que ya debes estar harto de oír, pero que no por ello deja de ser cierta: con la práctica (otra vez...) acabarás por saberte de memoria los puertos más importantes del MSX. *Prometible* y prometido.

**20 INSTRUCCIONES DE CONTROL** [**Índice**](#INDICE)

Según el manual del Z80, estas instrucciones "modifican la situación operativa de la CPU o manipulan la información de su estado interno". Bueno, yo prefiero decir que sirven para torear un poco al Z80. Vamos a verlas.

Para empezar tenemos **NOP**, que realiza una portentosa demostración de vagancia y no hace nada. Es tan inútil que su código de operación es $00. ¿Que para qué sirve? Como retardo (imprescindible, por ejemplo, en algunas operaciones con el VDP), como relleno (¿quieres eliminar una instrucción sin tener que ensamblarlo todo de nuevo? La pisas con ceros, y listo) o para reservar un cacho de memoria para datos (también podrías rellenarla con 34s, pero suele ser más limpio, y más claro a la hora de desensamblar, usar ceros). He ahí la utilidad de lo inútil.

**HALT**: provoca un infarto al Z80, es decir, lo deja inactivo (ejecutando **NOP**s para refrescar la memoria) hasta que recibe una interrupción. Curioso...

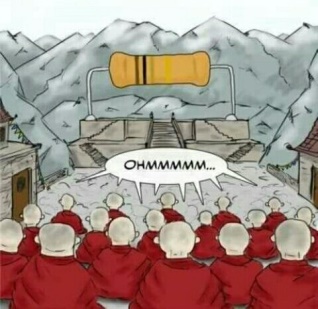
**DI** & **EI**: invalida y valida interrupciones, respectivamente. Trabajando en modo **DI** ganas velocidad, pero si no vas con cuidado hay riesgo de cuelgue. Vamos, que es como contener la respiración... (El crimen perfecto: ¡**DI** seguido de **HALT**!)

**IM** 0, **IM** 1 & **IM** 2: selecciona uno de los tres modos de interrupción del Z80. El MSX sólo usa **IM** 1, es decir, salto automático a $38 al recibir una interrupción, y a partir de ahí ya te apañarás...

¿Las banderas? Qué vaaa, estas instrucciones no las tocan para nada...

30

**21 CONCLUSION CITY** [**Índice**](#INDICE)

Bbbuuuuuuuffff... se acabó lo que se tecleaba. Ya hemos visto todas las instrucciones del Z80.

Reconozco que esta entrega de Easymbler también me ha quedado un pelín pesada, pero piensa que es difícil hacer secta con un **LD** A, 34 o un **DJNZ**. Lo que has de hacer (nuuuunca me cansaré de reptirlo) es leerlo todo con calma, sin intentar retener nada; más que como texto docente (eeks, parezco un profesor) esta lista de instrucciones te servirá como referencia en el futuro.

¿De qué irá el próximo plomazo? Bueno, ya va siendo hora de profundizar un poquito más en la estructura del MSX, de aprender a usar un ensamblador de verdad (aunque lo que veremos se puede aplicar a todos, no me cansaré de recomendarte el mejor, el que está a eones luz de los demás: COMPASS, claro) y de meterle mano a unas cuantas técnicas básicas de programación.

*Nota de PipaGerardo:* En esta adaptación de Easymbler usaremos el “SjasmPG” porque es gratuito, liviano y está especialmente adaptado para el Z80 y el MSX. He buscado por internet “COMPASS” y es de pago y yo personalmente no puedo obligaros a gastaros dinero en un compilador habiendo alternativa.

Mientras, con lo que has visto, ya puedes ir haciendo alguna cosilla. Yo empecé con un programa que llenaba la pantalla de As... sólo necesitas disponer de la rutina de impresión de un carácter en pantalla (con la BIOS conectada, o lo que es lo mismo, con el BASIC activo, se encuentra en $00A2; tienes que hacer **LD** A, ‘A’ y llamarla) y componer un bucle. ¿Serás capaz?

¡¡¡Claro que sí!!!

¡Hasta incluso!

///// KONAMI MAN \\\\\

**Ejercicio transforma este programa de BASIC en Ensamblador SjasmPG:**

10 **LET** ANC%=39

20 **LET** ALT%=23

30 **SCREEN** 0 : **WIDTH** 40

40 **FOR** A%=ASC("A") **TO** ASC("Z")

50 L%=0

60 AS$= CHR$( A% )

70 **FOR** C%=1 **TO** ALT%

80 **LOCATE** 0, L%

90 **FOR** B%=1 **TO** ANC%

100 **PRINT** ; AS$ ;

110 **NEXT** B%

120 L%=L%+1

130 **NEXT** C%

140 **NEXT** A%

150 **CLS**

160 **GOTO** 160

Directivas para compilar una ROM de 16K:

**DEFPAGE** 2, $8000, $4000 ; Nuestra ROM 16K

**MAP** $C000 ; RAM 16K

**PAGE** 2 ; Seleccionamos nuestra ROM

**DB** $41, $42 ; Cartucho ROM

**DW** INICIO ; Dirección de Inicio

**DW** 0, 0, 0, 0, 0, 0 ; 12 Bytes para completar cabecera

Rutinas de la BIOS a usar:

**CALL** INITXT ; Inicia el SCREEN 0 texto de 40 x 24 con 2 colores

**CALL** POSIT ; Posiciona el cursor H para columnas y L para las filas.

**CALL** CHPUT ; Escribimos el carácter contenido en A.

**CALL** CLS ; Limpia la pantalla, requiere Z=1 truqui “**CP** A”

En la siguiente página el resultado...

31

;-----------------------------------------------------------

; DEFINIR CONTANTES DEL SISTEMA

;-----------------------------------------------------------

LINL40: **EQU** $F3AE

INITXT: **EQU** $006C

CLS: **EQU** $00C3

POSIT: **EQU** $00C6

CHPUT: **EQU** $00A2

ANCHO: **EQU** 40 ; Constante ANCHO=40

ALTO: **EQU** 23 ; Constante ALTO=23

;-----------------------------------------------------------

; DIRECTIVAS PARA EL ENSAMBLADOR ( SjasmPG )

;-----------------------------------------------------------

**DEFPAGE** 2, $8000, $4000 ; Nuestra ROM 16K

**MAP** $C000 ; RAM 16K

**PAGE** 2 ; Seleccionamos nuestra ROM

**DB** $41, $42 ; Cartucho ROM

**DW** INICIO ; Dirección de Inicio

**DW** 0, 0, 0, 0, 0, 0 ; 12 Bytes para completar

;-----------------------------------------------------------

INICIO:

; INICIO DEL PROGRAMA

;-----------------------------------------------------------

; ETIQUETAS: TIEMPO

; INSTRUCCIONES COMENTARIOS

;-----------------------------------------------------------

**LD** A, 40 ; 7 A=40

**LD** [LINL40], A ; 13 Definimos 40 como ancho de línea.

**CALL** INITXT ; 17 SCREEN 0 texto de 40 x 24 con 2 colores

**LD** D, 'A' ; 7 D='A'

.BUCLE:

**CALL** IMPRIME ; 17 Imprimir la pantalla

**INC** D ; 4 ++D;

**LD** A, 'Z'+1 ; 7 A='Z'+1

**CP** D ; 4 (A-D)

**JP** **NZ**, .BUCLE ; 10 if(A!=0) goto .BUCLE;

; LIMPIAMOS LA PANTALLA

**CP** A ; 4 (A-A)==0 -> Z=1

**CALL** CLS ; 17 Limpia la pantalla machaca AF, BC y DE

; BUCLE FINAL

FIN:

**JP** FIN ; 10 goto FIN

;---------------------------------------------------------

IMPRIME:

; RUTINA QUE IMPRIME EL TEXTO EN PANTALLA

; ENTRADAS: D

; MODIFICA: AF, BC, y HL

;---------------------------------------------------------

**LD** H , 1 ; 7 H=1; Situamos la Columna X

**LD** L , 1 ; 7 L=1; y la Fila Y para

**CALL** POSIT ; 17 fijar el cursor a escribir.

**LD** A, D ; 4 A=D; El carácter a imprimir.

**LD** B, ANCHO ; 7 B=ANCHO;

**LD** C, ALTO ; 7 C=ALTO;

.BUCLE:

**CALL** CHPUT ; 17 Escribimos el carácter contenido en A.

**DJNZ** .BUCLE ; 13 --B; if(B!=0) goto .BUCLE;

**LD** B, ANCHO ; 7 B=ANCHO;

**DEC** C ; 4 --C; Decrementamos ALTO.

**JP** **NZ**, .BUCLE ; 10 if(C!=0) goto .BUCLE;

**RET** ; 10 FIN

;---------------------------------------------------------

**ENDMAP** ; FIN DEL MAPA DE MEMORIA

**END** ; FIN DE PROGRAMA

;---------------------------------------------------------

32

**22 LIGERISIMA INTRODUCCION A LA ARQUITECTURA DEL MSX** [**Índice**](#INDICE)

Y tan ligera. Sólo nos vamos a ocupar de la organización de la memoria, y sin muchos detalles: lo justo para empezar a programar. Tú tranquilo, está *chupao*.

Tenemos un Z80 como microprocesador, o lo que es lo mismo, acceso a 65536 direcciones de memoria, de $0000 a $FFFF. Este espacio de direcciones se divide en cuatro páginas:

|  |  |  |  |
| --- | --- | --- | --- |
| Página 0 | $0000 | a | $3FFF |
| Página 1 | $4000 | a | $7FFF |
| Página 2 | $8000 | a | $BFFF |
| Página 3 | $C000 | a | $FFFF |

Estas son las direcciones que el Z80 puede leer y escribir a través de las instrucciones apropiadas, como ya vimos. ¿Y qué es lo que estamos leyendo/escribiendo realmente en un MSX? Pues, ¿qué va a ser? Memoria, ni más ni menos. ¿Y cómo se le enchufa memoria al Z80 de los MSX? A través de los slots.



Un MSX dispone de cuatro slots o bancos de memoria, salvo alguna excepción son dos internos y dos externos (sí, éstas son las famosas ranuras para cartuchos). A cada uno de estos bancos podemos conectar una memoria de 64K, que también se divide en cuatro páginas de 16K.

Una vez que tenemos la memoria conectada al slot y el ordenador encendido podemos, mediante un puerto llamado registro selector de slot (por si tienes curiosidad es el $A8), conectar cualquier página de cualquier slot a la página del Z80 que tenga el mismo número (es decir, si quieres conectar al Z80 la página 2 de un slot, tienes que hacerlo por narices en la página 2 del Z80), y a partir de ahí leerla y escribirla normalmente con LDs y similares.

Los slots externos están para que el usuario conecte lo que le dé la gana, desde el Nemesis hasta el Mega-SCSI, sin olvidarnos de las ampliaciones de memoria o las unidades de disco externas. ¿Y qué hay de los slots internos? Vienen conectados de fábrica con ROMs indispensables para el funcionamiento del MSX. Principalmente la BIOS, el intérprete del BASIC, la controladora de disco, la BIOS extendida de los MSX2/2+/TR y por supuesto la RAM; luego están los "extras" como el FM-PAC de los 2+/TR, el Turbo-BASIC del MSX2+ de Sanyo o la agenda del Sony F9S.

El MSX tiene dos modos de funcionamiento, o dos ambientes, que como ya sabrás son el MSX-BASIC y el MSX-DOS. De momento trabajaremos en modo BASIC, cuya disposición de páginas conectadas es la siguiente:

\* Página 0: ROM-BIOS. Se trata de una serie de rutinas destinadas a la gestión de los recursos del MSX, están estandarizadas y garantizan un funcionamiento idéntico independientemente del MSX que las ejecute. Permiten realizar programas 100% compatibles, pero resultan un poco lentas. De momento, y como estamos aprendiendo, usaremos mucho la BIOS. Cuando tengamos más experiencia y sepamos bien dónde pisamos podremos ir prescindiendo poco a poco de ella, con lo que ganaremos en velocidad, pero tendremos que ir con muuuucho cuidado para no hacer programas incompatibles.

\* Página 1: Intérprete BASIC. El responsable de que el SaveR haya sido capaz de empezar un clon del SD-Snatcher. No es más que un larguísmo (16K) programa en ensamblador grabado en una ROM.

\* Página 2: RAM. Aquí se almacena el programa que el usuario teclea, carga o un cartucho ROM de 16K.

\* Página 3: RAM otra vez, pero gran parte de ella (unas 10K) está reservada como zona de trabajo para el sistema; vamos, que el usuario no la puede tocar si el ordenador no quiere colgar. En total, el usuario dispone de unas 23K, a partir de $8000, para sus programas BASIC y ensamblador (sí, pueden convivir a la vez en memoria siguiendo unas reglas básicas).

33

Para convivir BIOS, BASIC y nuestros programas los podemos ensamblar a partir de ORG $A000, con lo que podremos meter programas BASIC entre $8000 y $9FFF y mezclarlos con nuestras rutinas por medio de USRs y similares (práctica aceptable para principiantes, pero con el tiempo debes aprender a pasar olímpicamente del BASIC y programar al 100% en ensamblador).

En el caso de no convivir con el BASIC pero no querer eliminarlo (para cargar cintas con BLOAD por ejemplo) se puede usar ORG $8000 para crear una ROM o una archivo binario de 8K a 16K, y memoria RAM libre en la tercera página (menos las variables de sistema BIOS y BASIC).

En el caso de ser un cartucho o un binario que no usa el BASIC absolutamente para nada (ni para BLOAD) se puede usar ORG $4000, para crear un cartucho ROM o binario de hasta 32K y memoria RAM libre en la tercera página (menos las variables de sistema BIOS).

Y si eres tan chulo que tampoco vas a usar la BIOS para nada pues tu solito te lo trabajas y puedes crear hasta 48K de ROM o archivo BIN y 16K de RAM en la página 3.



Un inciso, ¿Qué pasa si quieres crear un *cassette* de 48K de programa y 16K de RAM... Pues no se puede, tendrías que hacer un cargador de archivos binarios que no use BASIC ni BIOS y luego un archivo binario de 48K con el programa a cargar, pero esto de momento lo dejamos para más adelante...

¿Y que pasa con el MSX-DOS? En ensamblador se pueden usar diferentes tipos de encabezados y arrancando en ORG $0100 también se puede producir un archivo COM ejecutable desde MSX-DOS. De todos modos del entorno MSX-DOS ya hablaremos cuando nos haga falta, tú *tranqui*.

Uff, me ha quedado más largo de lo que pensaba. He omitido a propósito detalles como los slots expandidos, la RAM mapeada o el misterio de los Mega-ROMS, ya que no nos interesan para empezar a programar. Resumiendo que usando diferentes cabeceras y arrancando en diferentes direcciones de memoria se pueden crear los distintos tipos de programas que ejecuta un ordenador MSX.

**23 INTRODUCCION A LOS ENSAMBLADORES DE VERDAD** [**Índice**](#INDICE)

Veo que no sabes qué hacer con el hacha... te estás impacientando y quieres empezar a programar, pero lo que te estoy contando no lo sabías y te parece interesante... pues nada, tú sigue dudando, que yo seguiré corriendo, y por el camino explicando.

Bueno, el último plomazo que nos queda por mirar es el funcionamiento del programa ensamblador con el que trabajaremos. Aunque soy un plasta y no paro de recomendar el Compass, lo que diré es igualmente válido para cualquier ensamblador y en este caso usaremos “SjasmPG” como ya comenté anteriormente.

Un programa de estas características suele incluir tres módulos. Para empezar el ensamblador propiamente dicho, que no sería más que un procesador de textos normal y corriente (con algunas facilidades de edición como copia de bloques, búsqueda y sustitución, etc), de no ser por la opción de ensamblar, que convierte el código fuente (el listado que hemos tecleado) en código objeto (el programa ya ensamblado en memoria), pudiendo después grabar ambos en disco. En este caso yo voy a usar “Geany” junto con “SjasmPG”.

Otro módulo importante es el desensamblador/simulador, que nos muestra el contenido de la memoria en forma de instrucciones del Z80. Desde aquí podemos ejecutar una instrucción concreta o un grupo de ellas y observar/modificar el contenido de los registros durante el proceso. Indispensable para probar las rutinas que iremos haciendo. Y por último, el monitor nos muestra la memoria tal cual, en forma de bytes y en formato ASCII simultáneamente; una utilidad de este módulo es, entre otras, buscar textos en memoria. Para esto es recomendable usar el emulador de MSX “BlueMSX” con su depurador.

34

Que nooo... que aún no hemos acabado... que nos quedan por ver un par de cosas del módulo ensamblador... que dejes de afilar el hacha...

El módulo ensamblador no sólo se limita a traducir texto a bytes. También nos ofrece una serie de facilidades como son las etiquetas, las directivas, las macros y el ensamblado condicional. De momento sólo nos fijaremos en las dos primeras.

Una etiqueta no es más que un nombre que das a la dirección de memoria en la que comienza justamente la instrucción o dato que a ti te interesa. Esto es imprescindible para saltos y bucles, por ejemplo:

**LD** A,[DATO] ;

**CP** B ;

**JR Z**,IGUALES ;

DISTINT: ;

**SUB** C ;

**RRA** ;

**RET** ;

IGUALES: ;

**ADD** E ;

**EI** ;

**RET** ;

DATO:

**DB** $34 ;

Tras la comparación, la ejecución saltará a IGUALES si (A==B), de lo contrario seguirá en DISTINT. Como ves no nos preocupan las direcciones concretas a las que se refieren estas etiquetas, si bien es posible consultarlas mediante una opción del ensamblador. El límite en el número de caracteres para una etiqueta depende del ensamblador en concreto, aunque no es recomendable pasarse de ocho caracteres (más que nada, no queda bonito). Además, es posible definir etiquetas locales, precediendo las etiquetas con un punto “.”. La particularidad de las etiquetas locales es que sólo están disponibles entre una etiqueta global y otra, o dentro de un módulo.

Ánimo, que ya queda menos. Las directivas son órdenes especiales para el ensamblador que se camuflan con las instrucciones normales del Z80. Las directivas básicas que todo ensamblador tiene y que nos resultarán imprescindibles para programar son:

\* **ORG**: Indica la dirección de memoria a partir de la cual será ensamblado el listado situado a continuación. Como hemos visto, nuestros listados que no usan el BASIC comenzarán siempre en ORG $8000 o lo que es lo mismo al principio de la página 2.

\* **PAGE**: Parecido que ORG pero se especifica la página de memoria física ( de 0 a 3 ) o como en el “SjasmPG” la página lógica lo que nos permite usar hasta 256 páginas de 0 a 255 con diferentes posiciones de memoria y capacidades, claro que primero hay de definirlas con DEFPAGE número\_de\_pagina [, origen [, tamaño]]

\* **EQU**: Define una constante, es decir, un nombre al que se asocia un valor concreto de uno o dos bytes. Cada vez que el ensamblador encuentre ese nombre lo sustituirá por el valor correspondiente. Ejemplo:

TOTREN: EQU $34

DIREC: EQU $ABCD ; Lo que el ensamblador hará en realidad, esto:

**LD** HL, DIREC ; LD HL, $ABCD

**LD** A, TOTREN ; LD A, $34

**LD** [HL],A

\* **DB** o **DEFB** (define byte): Esta instrucción sirve para ensamblar bytes sueltos que han de ser interpretados como datos, es decir, que no son instrucciones. Es posible definir con un solo .DB varios bytes en formato decimal, hexadecimal o ASCII:

**LD** DE, CADENA

**JP** PRINT

CADENA: **DB** $0C, "Esto se imprimirá", 13, 10, ‘$’

DATOS: **DB** 0, 3, 2, $34, 5, 220

Si ensamblamos esto y después nos vamos al monitor podremos leer "Esto se imprimirá" mezclado entre los datos e instrucciones: el ensamblador se ha encargado de transformar cada carácter de la cadena en un byte equivalente a su código ASCII.

35

\* **DW** o **DEFW** (define word): Actúa casi igual que DB, sólo que ahora los datos son de dos bytes. ¡¡MUCHO CUIDADO!! Los datos de dos bytes definidos con DW son almacenados según el formato estándar del Z80, es decir, el byte bajo antes que el alto. Así, si ensamblamos esto:

**DW** $1234, $5678 ; lo que tendremos en memoria será: $34, $12, $78, $56

\* **DS** o **DEFS** (define space): De este modo se reserva un espacio de X bytes en el punto actual de memoria. Puede servir para identificar variables en RAM.

**DS** 10 ; Reserva 10 bytes de espacio.

\* **INCLUDE** : “archivo.asm” : Incluye código fuente almacenado en un fichero externo. Muy útil para proyectos grandes para partir el programa en módulos independientes.

\* **INCBIN** : “datos.bin” : Incluye datos o código ya compilado y lo inserta en el programa. También se suele poder especificar el inicio y la longitud del archivo.

Pues ya está, con estas directivas tenemos de sobras para programar, y si necesitamos más podemos y debemos consultar el manual del ensamblador “SjasmPG”. Como soy así de malo y quiero liarte un poco, te recuerdo que es posible mezclar las directivas, lo cual muchas veces hace el listado más comprensible:

CHPUT: **EQU** $00A2

CLS: **EQU** $0C

CR: **EQU** 13

LF: **EQU** 10

ENDSTR: **EQU** ‘$’

; Cabecera de un cartucho de cartucho 16K:

**OUTPUT** main.rom ; SjasmPG.exe –s main.asm main.rom

**DEFPAGE** 0, $8000, $4000 ; Definimos la página virtual 0

**PAGE** 0 ; Activamos la página virtual 0

**CODE @** $8000 ; Forzamos empezar el código en $8000

**BYTE** $41, $42 ; Cabecera para ROM

**WORD** INICIO ; Punto de inicio del código.

**WORD** 0,0,0,0,0,0

CADENA:

**DB** CLS, "¡Suelta el hacha,", CR, LF

**DB** "que ya empezamos a programar!", CR, LF, CR, LF

**DB** "¡¡Te lo juro!!", CR, LF, ENDSTR

INICIO:

**LD** HL, CADENA

**CALL** PRINTXT

**RET**

PRINTTXT:

**LD** HL, TEXTO

.BUCLE:

**LD** A, [HL]

**OR** A ; Más corto que CP 0

**RET Z** ; Termina si el byte leído es 0

**CALL** CHPUT

**INC** HL

**JR** .BUCLE

36

**24 SEÑORAS Y SEÑORES, PROGRAMEN Y VEAN** [**Índice**](#INDICE)

Pues ya está. ¿Has cargado el ensamblador? ¿Dispuesto a machacar el teclado? ¡Vamos allá! Lo primero que haremos será ese programilla para llenar la pantalla de As que propuse al final del capítulo 21. Como en modo texto la pantalla mide 40 columnas por 23 filas, hemos de imprimir 920 As. Vaya, qué fastidio; si fueran menos de 256 lo teníamos más fácil verdad:

; Cabecera de un cartucho de cartucho 16K:

**OUTPUT** main.rom ; SjasmPG.exe –s main.asm main.rom

**DEFPAGE** 0, $8000, $4000 ; Definimos la página virtual 0

**PAGE** 0 ; Activamos la página virtual 0

**CODE @** $8000 ; Forzamos empezar el código en $8000

**BYTE** $41, $42 ; Cabecera para ROM

**WORD** INICIO ; Punto de inicio del código.

**WORD** 0,0,0,0,0,0

INICIO:

**LD** B, 200 ; o los que sean, hasta 255

**LD** A, ‘A’ ; o LD A,65

BUCLE:

**CALL** $A2

**DJNZ** BUCLE

**RET**

Parece mentira que un programa tan corto me permita hacer tantos comentarios:

- No olvides nunca las cabecera de un cartucho de 16K, a partir de ahora yo no la pondré más, esta cabecera define una ROM de 16K pero puedes cambiarla para crear por ejemplo un binaro o cassette de audio.

- Observa la enorme utilidad de la etiqueta BUCLE.

- Ojo: estamos llamando a $A2 alegremente, porque estamos trabajando con la BIOS conectada. Recuerda que $A2 imprime en pantalla el carácter indicado en el acumulador.

- La rutina de impresión de un carácter no modifica el acumulador. De lo contrario, la instrucción **LD** A, 'A' debería estar dentro del bucle.

- Es imprescindible dotar al programa de una buena documentación. No ahorres comentarios: recuerda que al ensamblar son ignorados.

- Una vez ensamblado el programa, la forma más rápida de ejecutarlo es desde el propio editor “SciTE” que si se configura como es debido además de llamar al “SjasmPG” para ensamblar el código, también puede llamar a un emulador de MSX como “OpenMSX” por ejemplo y correr al ROM. Es necesario el **RET** al final, ya que de lo contrario el Z80 ejecutaría lo primero que encontrara tras tu programa, y lo más probable en estos casos es conseguir una perfecta emulación del Windows.

Bueno, bueno, lo que queríamos era un bucle de 920 pasos, ¿verdad? Pues vamos allá:

ANCHO: **EQU** 40 ; Constante ANCHO=40

ALTO: **EQU** 23 ; Constante ALTO=23

CARACT: **EQU** ‘A’ ;

**LD** A, CARACT ; El carácter a imprimir.

**LD** B, ANCHO ; B=ANCHO;

**LD** C, ALTO ; C=ALTO;

.BUCLE:

**CALL** CHPUT ; [$00A2]; Escribimos el carácter contenido en A.

**DJNZ** .BUCLE ; --B; if(B!=0) goto .BUCLE;

**LD** B, ANCHO ; B=ANCHO;

**DEC** C ; --C; Decrementamos ALTO.

**JP** **NZ**, .BUCLE ; if(C!=0) goto .BUCLE;

**RET** ; FIN

Para empezar, observa que el uso de etiquetas hace el programa mucho más comprensible, y de fácil modificación (si tu programa tiene 34 bucles parecidos a este y te da el venazo de imprimir la ‘B’ en vez de la ‘A’, sólo tienes que cambiar la definición de la etiqueta CARACT). El nombre de la rutina $A2 no me lo he inventado yo: cada rutina de la BIOS tiene asignado un nombre que nadie nos obliga a usar, pero es recomendable hacerlo, sobre todo si otra persona va a leer nuestro listado.

37

Pero veamos cómo hemos hecho los bucles. **DJNZ** se encarga de decrementar B y comprobar si aún no es cero, este bucle lo usamos para imprimir 40 caracteres de ancho. Y en el otro bucle **JP NZ** decrementamos C y comprobamos si aún no es cero, con lo que hacemos 23 líneas. Si multiplicamos 40 caracteres por 23 líneas tenemos 920 CHPUT y toda la pantalla llena de caracteres.

Otro detalle importante: ¿qué pasaría si CHPUT machacara el par BC? Nos veríamos obligados a guardarlo y recuperarlo en cada iteración. Pues para eso está la pila:

...

**PUSH** BC

**CALL** CHPUT

**POP** BC

...

Como ya vimos hace algún tiempo, la pila es un buen lugar para guardar datos temporalmente mientras usamos los registros para otra cosa. Otra solución es usar otros registros; por ejemplo, si sólo usáramos el registro B y estuviéramos seguros de que CHPUT no toca E podríamos hacer:

...

**LD** E, B

**CALL** CHPUT

**LD** B, E

...

Ahorrando **PUSH** y **POP** ahorramos accesos a memoria, con lo cual el programa gana velocidad. Pero no te comas el coco: rara vez necesitarás un control tan crítico de la velocidad de ejecución, y menos ahora que estás empezando. Así que ya lo sabes: si necesitas guardar un dato temporalmente y no tienes muy clara la disponibilidad de los registros, no te lo pienses; usa la pila, que para eso está, y a otra cosa.

Y ahora vamos a ver otro ejemplo de bucle: una rutina de impresión. No, no es que tenga efectos especiales ni realidad virtual ni 34 japonesas renderizadas; aquí "impresión" viene de "imprimir", no de "impresionar". La rutina en cuestión imprime la cadena que comienza en TEXTO y termina con un byte cero:

PRINTTXT:

**LD** HL, TEXTO

.BUCLE:

**LD** A, [HL]

**OR** A ; Más corto que CP 0

**RET Z** ; Termina si el byte leído es 0

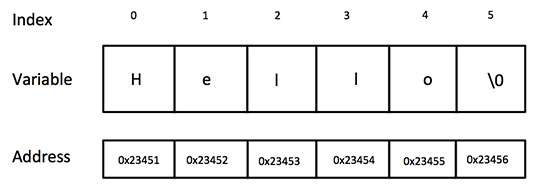
**CALL** CHPUT

**INC** HL

**JR** .BUCLE

TEXTO: .DB "I hate axes!!", 0

La técnica de almacenar cadenas de texto con un byte de terminación es muy usada, y permite realizar este tipo de bucles, en el que no conocemos de antemano el número de iteraciones: simplemente, vamos imprimiendo hasta encontrar el byte de terminación, el cero en este caso. "¿Y por qué el cero?" Sí, podríamos haber usado cualquier otro, pero el cero viene bastante bien para esto: no existe ningún carácter cuyo código ASCII sea el 0, con lo cual no hay peligro de que un carácter de la cadena sea confundido con el byte de terminación; y además podemos comprobar el carácter con un OR A (que equivale a CP 0 pero ocupa un byte menos).



38

**25 INSISTO: ¡¡DOCUMENTACION, POR FAVOR!!** [**Índice**](#INDICE)

Como eres tan avispado ya te habrás dado cuenta de que el programa anterior sólo funciona si CHPUT no modifica el par HL, y antes ya hemos supuesto que no modifica BC. La pregunta ahora es: ¿qué demonios modifica o no modifica CHPUT? O más generalmente, ¿cómo podemos saber qué registros modifica una rutina, y ya que estamos, cuáles son sus parámetros de entrada y salida?

Si recuerdas, ya nos hicimos una pregunta parecida hace algún tiempo: ¿cuál es el efecto de tal o cual instrucción sobre las banderas? La solución está, de nuevo, en la documentación. Toda rutina ha de ir acompañada de una buena documentación (lo más fácil es incluirla como cabecera de la propia rutina en forma de comentario), que indique:

- Nombre de la rutina (Dirección de inicio BIOS).

- Función que realiza la rutina.

- Parámetros de entrada.

- Parámetros de salida.

- Registros modificados.

- Ejemplo de uso.

- ...

En el caso de la CHPUT, ya conocemos la dirección de inicio $00A2, los parámetros de entrada (el carácter a imprimir en A), y nos faltaba saber que no hay parámetros de salida ni registros modificados. Es decir, su descripción es algo como esto:

; ----------------------------------------

; CHPUT (#00A2)

; Imprime un carácter.

; Entrada: A = Carácter a imprimir.

; Salida: --

; Registros: --

; ----------------------------------------

Todas las rutinas de la BIOS tienen una descripción como esta, y no tienes más que agenciarte un listado de la BIOS (hay para elegir: en el MSX2 Technical Handbook lo tienes en inglés, en un fichero que acompaña al COMPASS en holandés, y en HNOSTAR#35 en castillero) para ponerte a investigar y probar las que más te gusten. De todas formas, cada vez que te ponga un ejemplo que use una rutina de la BIOS, pondré su descripción.

Pero lo importante es que te metas esto en el hacha, digo en la cabeza: la mayor parte del tiempo que dedicarás a programar en ensamblador se te irá en programar subrutinas, y es importante que dotes a cada una de ellas de una buena documentación: como mínimo la descripción, y mejor si vas poniendo comentarios a lo largo del listado. De esta forma no te costará nada retomar el hilo de lo que estabas haciendo si un buen día decides ampliar aquella rutina que dejaste a la mitad cuando te fuiste de la guardería, o aquella otra que heredaste de tu abuela; o reusar una rutina antigua (propia o ajena) en otro programa...

Como ejemplo vamos a rehacer nuestra rutina de impresión, esta vez un poco más sofisticada:

; ----------------------------------------

PRINTXT:

; Imprime una cadena de caracteres.

; Entrada: HL Dirección de inicio de la cadena.

; A Carácter de terminación.

; Salida: BC Longitud de la cadena, sin el carácter de terminación.

; Registros: HL, AF

; ----------------------------------------

39

**PUSH** DE

**LD** BC, 0 ; Inicialización del contador de longitud

**LD** D, A ; Guardamos en D el carácter de finalización

.BUCLE:

**LD** A,[HL]

**CP** D

**JR Z**, FIN ; Finaliza si hemos encontrado el carácter de fin.

**CALL** CHPUT

**INC** HL ; Siguiente carácter

**INC** BC ; Incremento del contador

**JR** .BUCLE

FIN:

**POP** DE ; Finalización

**RET**

; ----------------------------------------

Como puedes ver, gracias a la cabecera cualquier persona puede usar esta rutina, sin necesidad de saber qué instrucciones contiene ni cuál es su estructura (mismamente como si formara parte de la BIOS); y gracias a los comentarios te resultará fácil comprender su estructura y funcionamiento, cuando dentro de 34 años decidas modificarla o ampliarla.

Un comentario sobre los registros modificados. Habrás visto que al principio de la rutina guardamos el par DE, recuperándolo al final; esto es lógico, ya que este par no está incluido en la lista de registros modificados y en un momento dado usamos el registro D. La pregunta es: ¿podríamos habernos ahorrado el **PUSH** DE y el **POP** DE, e incluir DE en la lista de registros modificados? ¿Y no podríamos también haber hecho un **PUSH** HL/**POP** HL, eliminando así HL de la lista de registros modificados? Pues sí. Como pone en los yogures, hay miles de premios. En el arte de la programación siempre hay muchas formas de hacer lo mismo, y encontrar la mejor no siempre es fácil; de hecho, muchas veces tendrás que elegir entre varias opciones igualmente válidas. La decisión final es tuya.

En concreto, para este programa de impresión sí que habría sido una buena idea olvidarse de guardar el par DE, ya que de esta forma podemos cambiar el **JR Z**, FIN por un **RET Z** y eliminar las dos últimas líneas, con lo cual acortamos la rutina. Con la práctica cazarás estos detalles al vuelo, lo que te permitirá realizar programas cada vez más cortos y optimizados. Y, por favor, permíteme insistir: no ahorres documentación.

¡¡¡**MUCHO CUIDADO**!!! A mí ya me ha pasado una vez, y es un error difícil de detectar. Si se te ocurre poner en una rutina algo así...

; Inicio de la rutina

 **PUSH** uno, otro

...

...

**CP** lokesea

**RET Z**

...

...

**POP** otro,uno

**RET**

...te volverás loco. Algunas veces (cuando la comparación resulte en Z==0) la rutina funcionará de maravilla, pero las demás se te colgará, o empezará a salir roña por la pantalla, o sonará una música del SaveR... Para evitar problemas, NUNCA pongas un **RET** Z (o equivalente con otra bandera) en medio de una rutina. Cuando quieras colocar una finalización de la rutina usa la técnica del **JR Z**, FIN como en la rutina PRINTXT que hemos visto antes; sólo si al terminarla ves que no has guardado ningún registro al principio, cambia estos **JR** FIN por **RET**. Y cuando una rutina se te cuelgue, ya sabes qué es lo primero que tienes que comprobar...

40

**26 INVERSIONATOR** [**Índice**](#INDICE)

Como veo que sigues vivo después del sermón sobre la documentación (parezco un guardia civil cualquiera), y ahora que ya eres un experto (más o menos) en bucles y subrutinas, vamos a diseñar una ídem un pelín más complicada: **Inversionator**. Bajo este estúpido nombre se esconderá una no menos estúpida rutina cuya función será... (redoble: trrrrr...) ¡¡Transformar las mayúsculas de una cadena en minúsculas y viceversa!!

¿Qué te crees, que porque hayas afilado y abrillantado el hacha me da más miedo? ¡Pues no señor, ya es imposible que me dé más miedo, sobre todo cuando la coges con cara de sádico asesino como ahora! Ya sé que a ti no te interesa para nada *desmayuscular* o **desminuscular** o lo que sea una cadena, que lo que quieres es programar el Nemesis 34, pero hay que ir poco a poco, y este será un buen ejemplo de diseño de una rutina. Seguro que los de Konami empezaron así. "¡Pero si no hay kanjis mayúsculos ni minúsculos!" Estooo... bueno, pues pregúntale a esa japonesa cómo empezaron los de Konami.

(Es increíble... ha picado... se ha girado y he aprovechado para salir corriendo...)

Bueno, a lo que íbamos. Queremos diseñar una rutina, ¿por dónde empezamos? Antes de preguntarnos COMO haremos tal cosa siempre hemos de saber QUE queremos hacer exactamente, así que vamos a empezar por la punta del iceberg, o el meollo de la cuestión, o la madre del cordero: llámese a gusto del consumiente a la cabecera de la rutina.

; ----------------------------------------

MAYMIN:

; Convierte las mayúsculas de una cadena en minúsculas y viceversa.

Procura elegir un nombre para la subrutina que sea corto pero lo suficientemente auto explicativo, piensa que si en medio de un listado te encuentras un **CALL** JFDR&!#FGL, por mucha relación que tenga esa etiqueta con la filosofía intrínsecamente trascendental de la rutina, no vas a recordar para qué demonios sirve. En cuanto a la descripción, no conviene pasarse de dos o tres líneas.

Vamos con las entradas. Está claro que la entrada de esta rutina será una cadena, o mejor dicho, un puntero a cadena. Y aquí ya tenemos que empezar a tomar decisiones: ¿La cadena ha de estar terminada con un carácter especial? En ese caso, ¿Fijo, o a elegir? O por el contrario, ¿será necesario especificar la longitud?

Cualquiera de las tres opciones es válida. Tenemos entonces tres posibles entradas:

; Entrada: HL Cadena, acabada en 0 (o en lo que sea).

; Entrada: HL Cadena

; A Carácter de terminación de la cadena.

; Entrada: HL Cadena

; B Longitud de la cadena.

Otra opción algo más retorcida, pero que hará nuestra rutina más flexible, es la que nosotros vamos a implementar (yej, yej):

; Entrada: HL Cadena

; CY==0 -> A = Carácter de terminación, B ignorado.

; CY==1 -> B = Longitud de la cadena, A ignorado.

Ahora, quien use la rutina podrá elegir entre ambas opciones, simplemente poniendo el acarreo a cero (**OR** A) o a uno (**SCF**) y usando A o B a su conveniencia. Además, como una opción bastante frecuente es la de terminar una cadena con el carácter 0, basta un **XOR** A para matar dos pájaros de un tiro (repasa las operaciones lógicas y las banderas). Es frecuente usar el acarreo como bandera para elegir entre dos opciones de entrada (por ejemplo, la rutina de lectura/escritura de disco de la BIOS lee sectores si CY=0 a la entrada, y los escribe si CY=1).

41

También podríamos haber hecho lo mismo prescindiendo del acarreo:

; Entrada: HL Cadena

; B == 0 -> A = Carácter de terminación de la cadena.

; B <> 0 -> B = Longitud de la cadena, A ignorado.

Ahora pasemos a la salida. Evidentemente, la rutina ha de devolver la cadena debidamente modificada, pero ¿ha de sobrescribir la cadena original, o crear una nueva? En este último caso, necesitamos un nuevo parámetro de entrada: la dirección en la que depositar la cadena nueva...

; Entrada: HL Cadena de origen.

; DE Dirección cadena de destino.

; CY==0 -> A = Carácter de terminación, B ignorado.

; CY==1 -> B = Longitud de la cadena, A ignorado.

Y la salida queda tal que así:

; Salida: DE Cadena convertida, acabada con el mismo cáracter

; que la original si CY==0 a la entrada.

; Cadena original inalterada en HL

; B Longitud de la cadena sin contar la terminación

Es bastante razonable que si la cadena original tiene un carácter de terminación, la cadena creada también lo tenga y sea el mismo. ¿Y por qué devolvemos la longitud de la cadena en B? No es imprescindible, pero gracias a esta función (que tampoco nos costará mucho añadir) podremos usar esta función para obtener la longitud de una cadena con carácter de terminación.

Antes de terminar con los parámetros de salida vamos a tratar una cuestión importante. Hasta ahora hemos sido muy optimistas. "¡Sí, has sido muy optimista al creer que no te alcanzaría!" ¡¡Cielos, el hombre hacha!! Menos mal que corro a más *megaherzios* que él.

Decía que hemos sido muy optimistas, porque hemos supuesto que nuestras rutinas siempre realizan correctamente su función. Pero en la mayoría de las rutinas que realicemos siempre habrá una o más condiciones de error, generadas normalmente por unos parámetros de entrada incorrectos; por tanto hemos de diseñar las rutinas de forma que sean capaces de detectar cualquier error y avisarnos adecuadamente (por ejemplo, tú mismo estarás harto de ver "Axe error: target escaped").

¿Y cómo se hace esto? Normalmente se utiliza el acumulador. Si la rutina ha terminado bien, valdrá cero; en caso contrario contendrá un código de error:

; Salida: Cadena tal

; Cadena cual

; B = Longitud

; A = Error:

; 0 -> No hay error

; 1 -> La cadena contiene códigos de control (<32)

; 2 -> Se han procesado 255 carácteres y no se ha encontrado

; el carácter de terminación (si CY==0 a la entrada)

; 3 -> CY==1 y B==0 a la entrada, o

; CY==0 y la cadena sólo contiene el carácter de terminación

Bueno, vale, tal vez he exagerado un poco. El error 1 es bastante surrealista, porque en una cadena de texto real puede haber códigos de control como cambios de línea o tabuladores; el error 2 puede eliminarse si se diseña la rutina de tal forma que admita textos de más de 255 carácteres (usando BC para la longitud en lugar de sólo B); y el error 3 puede, simplemente, ser ignorado (si la cadena está vacía, la rutina no hace nada y vuelve).

Así pues, vemos cómo una misma rutina puede no tener ninguna condición de error o tener tres, o incluso 34, dependiendo de lo que consideres como error. Como siempre la decisión es tuya.

42

Un último detalle respecto a los errores. Al igual que los propios errores, tú eres el que se inventa el código asociado a cada error, pero es costumbre asociar el código 0 a la ausencia de error, para poder hacer esto:

**CALL** MAYMIN

**OR** A

**JP NZ**, ERROR

OK:

**CALL** YASTA

...ya que, como dice el refrán, un **OR** vale más que mil **CP**s.

****

**"Bad joke detected. Target locked. Axe approaching."**

**¡¡¡KRAAASCH!!!**

**"Target escaped. Error distance: 0.0034 mm"**

Tío, esta vez sí que ha estado cerca. Ta has superado. Voy a tener que espabilar si quiero seguir operativo...

Bueno, pues sólo nos queda la lista de registros modificados. Aquí tenemos bastante libertad, pero es costumbre preservar los punteros a cadenas y dejar el par AF a merced de las hachas divinas; en cuanto a IX e IY, si no son usados como parámetros de entrada o salida lo mejor es dejarlos también ilesos. Queda el par BC. Como vamos a usar B como entrada y salida no tiene mucho sentido guardar BC, por tanto abandonaremos C a su suerte, y ya tenemos la última línea de la cabecera:

; Registros: C

Las banderas normalmente se presuponen modificadas. Esto se puede dejar así, pero si una vez terminada la rutina nos damos cuenta de que no tocamos C para nada no estaría mal sacarlo de la lista de registros modificados.

Bueeeno, pues ya tenemos completa la cabecera de nuestra rutina; nos ha costado unas cuantas líneas pero ha valido la pena, porque ahora sabemos EXACTAMENTE qué es lo que queremos que haga nuestra rutina, con lo cual la programación de la misma será un proceso, al menos, ordenado (al contrario que mi cuarto... snif...) Bueno, resumiendo:

; ----------------------------------------

MAYMIN:

; Convierte las mayúsculas de una cadena en minúsculas y viceversa.

; Entrada: HL = Cadena de origen

; DE = Buffer para la cadena de destino

; CY = 0 -> A = Carácter de terminación, B ignorado

; CY = 1 -> B = Longitud de la cadena, A ignorado

; Salida: DE Cadena convertida, acabada con el mismo carácter que la

; original si CY==0 a la entrada.

; Cadena original inalterada en HL

; B Longitud de la cadena sin contar la terminación.

; A Error:

; 0 -> No hay error.

; 1 -> La cadena contiene códigos de control (<32).

; 2 -> Se han procesado 255 carácteres y no se ha encontrado

; el carácter de terminación (si Cy=0 a la entrada)

; 3 -> CY==1 y B==0 a la entrada, o CY==0 y la cadena sólo

Contiene el carácter de terminación.

; Registros: C

; ----------------------------------------

Puede parecer larga, pero créeme, cuando tu biblioteca de rutinas alcance un tamaño respetable y tus programas no sean más que un puzzle de unas y otras, agradecerás tenerlas todas bien documentadas. El tiempo que pierdes diseñando y documentando una rutina lo compensas con creces a la hora de usarla.

43

Una vez tenemos la cabecera, podemos empezar con la rutina. Las dos primeras líneas están cantadas:

MAYMIN:

**PUSH** HL

**PUSH** DE

...y nada más empezar ya hemos de pararnos y darle curro a las neuronas. Vamos a ver, nos han pasado una cadena que hemos de recorrer, pero nos pueden haber pasado su longitud de dos formas distintas. Así pues, ¿programamos dos bucles de búsqueda? Uséase, ¿programamos un bucle que recorra la cadena hasta que ya no queden carcácteres (como la rutina de imprimir tropecientas "A") y otro que la recorra buscando un carácter de terminación (como PRINTXT), para usar uno u otro según la entrada?

No es una solución muy limpia. Lo mejor es programar un solo bucle para un tipo de entrada, y en caso de recibir la entrada del otro tipo, convertirla. Es decir, tenemos dos opciones:

- Programar un bucle que recorra la cadena de entrada hasta encontrar un carácter de terminación. Si a la entrada CY==1, añadir primero un carácter de terminación cualquiera al final de la cadena.

- Programar un bucle que recorra la cadena de entrada conociendo de antemano el número de iteraciones, es decir, su longitud. Si a la entrada CY==0, hallar primero la longitud de la cadena.

Parece un difícil dilema, pero si piensas un poco más (se piensa mejor dejando el hacha en el suelo, te lo garantizo) verás enseguida que la mejor opción es la segunda, porque:

1. Nadie nos ha dado permiso para modificar la cadena original añadiéndole nada: sólo tenemos derecho a escribir en la zona de memoria que comienza en DE.

2. De todas formas hemos de devolver la longitud de la cadena en B, así que si ya la medimos al principio afilamos dos hachas de un tiro.

Pues nada, visto esto ya podemos continuar con la rutina:

**LD** [TERM], A

**CALL NC**, MEDIR

**JR NC**, FIN2 ; Si MEDIR devuelve CY==0, terminamos con error 2

**LD** A, B

**OR** A

**JR Z**, FIN3 ; Error 3 si la cadena está vacía

**LD** [LONGIT], A

"¡Eh, un momento, para el carro!" Vale, admito que esto merece un par de explicaciones. TERM y LONGIT son dos variables en las que guardaremos, respectivamente, el carácter de terminación y la longitud de la cadena; el uso de variables para datos "fijos" de la rutina hace la misma más comprensible y evita las complicaciones derivadas de ir arrastrando datos en le pila o en otros registros.

¿Que dónde están estas variables? Si estamos usando memoria RAM pues dentro de la rutina, concretamente al final de la misma (para que no interfieran con el código ejecutable y para seguir un orden). Es decir, que después de todo el código antes de dar por terminada la rutina añadiremos:

TERM: DS 1

LONGIT: DS 1

Pero si estamos programando un cartucho en ROM hay que alojarlas en RAM en la página 3 por ejemplo “PAGE 3” o “ORG $C000”. Hay que tener mucho cuidado de no usar ROM para variables.

Más cosas. MEDIR es otra subrutina, ejecutada sólo si CY==0, que se encargará de medir la longitud de la cadena. Como se trata de una subrutina secundaria (interna a otra rutina), su cabecera puede ser un poco más chapucera. He aquí su código:

44

; ----------------------------------------

MEDIR:

; ----------------------------------------

; Devuelve en B la longitud de la cadena HL

; Termina con CY=0 si hay error 2,

; en caso contrario con CY=1

; Registros: A y C

; ----------------------------------------

**PUSH** HL

**LD** C, A ; A = Carácter de terminación.

**LD** B, 0 ; B = Longitud de la cadena.

**LD** A, $FF ; Informamos que CY=0 a la entrada

**LD** [TERM\_F], A

BUCMED:

**LD** A,[HL] ; Cogemos carácter

**CP** C ; ¿Es el de terminación?

**JR Z**, FINMED ; Sí: terminamos

**INC** B ; No: incrementamos contador...

**JR Z**, FINM2 ; (¿Error 2?)

**INC** HL ; ...y pasamos al siguiente carácter

**JR** BUCMED

FINMED:

**POP** HL

**SCF** ; CY = 1

**RET**

FINM2:

**POP** HL

**OR** A ; CY = 0

**RET**

; ----------------------------------------

Aunque no lo parezca, sí hemos tenido en cuenta la disponibilidad de los registros:

- A: Ya la hemos guardado en TERM antes de ejecutar la rutina, así que nos lo podemos cargar.

- B: Si hemos llamado a esta rutina es porque CY==0 a la entrada, es decir, B no contenía ningún dato útil.

- C: Está en la lista de registros modificados.

- HL: Esta sí hemos de guardarla, pues apunta a la cadena.

Tanto si MEDIR se ejecuta como si no, tras el **CALL NC**, MEDIR tendremos en B la longitud de la cadena, que almacenaremos en LONGIT. TERM\_F es una variable (la F es de "flag", bandera) que ponemos a $FF si hemos pasado un carácter de terminación, en caso contrario la dejamos a su valor inicial, es decir, 0. Nos hará falta para saber, al final de la rutina, si hemos de poner o no carácter de terminación en la cadena creada.

La detección del error 2 la realizamos examinando B una vez incrementado. Si se pasa de 255 volverá a 0, y en ese caso activará Z tras el **INC**. Volveremos entonces con CY=0 (no confundir con el acarreo de entrada), situación que el programa principal detectará y terminará con error 2. He aquí otro método para tratar errores sencillos (del tipo "hay o no hay error"): usar el acarreo. Observa por qué no podemos saltar directamente a FIN2 desde dentro de MEDIR...

Cuestión importante: ¿no sería más lógico volver con CY=1 si hay error, y CY=0 si no lo hay? Sí, pero esto implicaría esta modificación en el programa:

**CALL NC**, MEDIR

**JR C**, FIN2

Entonces, si a la entrada tenemos CY=1, MEDIR no se ejecutará, y al llegar al **JR C**, FIN2 continuaremos con CY=1, con lo cual el error 2 es inevitable. En cambio, tal como lo hemos hecho nosotros tenemos:

**CALL NC**, MEDIR

**JR NC**, FIN2

45

Si CY=1 a la entrada, tanto el **CALL** como el **JR** son ignorados; si CY=0 a la entrada, se realiza la llamada a MEDIR, de cuyo resultado depende que la ejecución salte o no a FIN2. No pongas esa cara, que no es tan complicado...

¿Y dónde se ponen las subrutinas de una rutina? Después del programa principal pero antes de las variables.

Pues parece que ya, sin más prolegómenos, podemos ponernos a masacrar la cadena (en sentido figurado, así que suelta el hacha). Pero antes hemos de hacernos una pregunta aparentemente bastante estúpida pero imprescindible si queremos continuar: ¿qué es una mayúscula y qué es una minúscula?

La respuesta correcta en este contexto (porque no creo que tu profe de lengua estuviera muy convencido) es la siguiente: una mayúscula es un carácter cuyo código ASCII está comprendido entre 65 y 90. Una minúscula, idem de idem entre 97 y 122. Si representamos la tabla ASCII linealmente, tenemos:

xxx...xxx ccc...ccc MMM...MMM ccc...ccc mmm...mmm ccc...ccc

1 - 31 32 - 64 65 - 90 91 - 96 97 - 122 123 - 255

Códigos Caráct. Mayúsc. Caráct. Minúsc. Caráct.

Control no alfab. no alfab. no alfab.

Ya sé que no estamos en la EGB para andar con dibujitos estúpidos, pero tampoco estamos en *Pesadilla en Ca'n Sbert* y bien que le das al hacha (o lo intentas). Este esquema nos permitirá diseñar la estrategia a la hora de convertir los caracteres, estrategia que será algo así como esto:

- Coger carácter si no ha acabado el bucle.

- Si es menor de 32, terminar con error 1.

- Si es menor de 65, no es alfabético: no modificarlo, escribirlo y goto coger.

- Si es mayor de 122, idem.

Si llegamos a este punto, es que el carácter está entre 65 y 122:

- Si es menor de 91, es una mayúscula. Convertir a minúscula, escribirlo y goto coger.

- Si es mayor de 96, es una minúscula. Convertir a mayúscula, escribirlo y goto coger.

- Si hemos llegado aquí es que está entre 91 y 96: no es alfabético, escribirlo sin modificarlo y goto coger.

¿Qué te ha parecido? Puede ser un planteamiento de una lógica aplastante, pero no es inmediato. Vale, igual sigues pensando que tanto trabajo para convertir una mayusculación (no sé si este palabro existe pero me da igual) es un poco absurdo, pero piensa que en el futuro, si aspiras a realizar programas mínimamente complejos tendrás que pelearte con estructuras de datos algo más complicadas (que muchas veces diseñarás tú mismo), y es importante que adquieras una buena metodología (es decir, ordenada).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 00 | | 16 | | 32 | | 48 | | 64 | | 80 | | 96 | | 112 | | 128 | | 144 | | 160 | | 176 | | 192 | | 208 | | 224 | | 240 | |
| 00 |  | 10 | † | 20 |  | 30 | 0 | 40 | @ | 50 | P | 60 | ` | 70 | p | 80 | Ç | 90 | É | A0 | á | B0 | Å | C0 | ▄ | D0 | ◄ | E0 | ⱷ | F0 | ≡ |
| 01 | ☺ | 11 | ┴ | 21 | ! | 31 | 1 | 41 | A | 51 | Q | 61 | a | 71 | q | 81 | ü | 91 | æ | A1 | í | B1 | å | C1 |  | D1 |  | E1 | β | F1 | ± |
| 02 | ☻ | 12 | ┬ | 22 | “ | 32 | 2 | 42 | B | 52 | R | 62 | b | 72 | r | 82 | é | 92 | Æ | A2 | ó | B2 | Ĩ | C2 | ■ | D2 |  | E2 | Ӷ | F2 | ≥ |
| 03 | ♥ | 13 | ┤ | 23 | # | 33 | 3 | 43 | C | 53 | S | 63 | c | 73 | s | 83 | å | 93 | ô | A3 | ú | B3 | ĩ | C3 |  | D3 |  | E3 | Π | F3 | ≤ |
| 04 | ♦ | 14 | ├ | 24 | $ | 34 | 4 | 44 | D | 54 | T | 64 | d | 74 | t | 84 | Á | 94 | ö | A4 | ñ | B4 | Õ | C4 |  | D4 |  | E4 | ∑ | F4 | ⌠ |
| 05 | ♣ | 15 | ┼ | 25 | % | 35 | 5 | 45 | E | 55 | U | 65 | e | 75 | u | 85 | à | 95 | ò | A5 | Ñ | B5 | õ | C5 | ▀ | D5 | ▀ | E5 | ∂ | F5 | ⌡ |
| 06 | ♠ | 16 | │ | 26 | & | 36 | 6 | 46 | F | 56 | V | 66 | f | 76 | v | 86 | ¨ | 96 | û | A6 | ª | B6 | Ũ | C6 | ▌ | D6 | ▄ | E6 | ɥ | F6 | ÷ |
| 07 | ▪ | 17 | ─ | 27 | ´ | 37 | 7 | 47 | G | 57 | W | 67 | g | 77 | w | 87 | ç | 97 | ù | A7 | º | B7 | ũ | C7 |  | D7 | ▓ | E7 | γ | F7 | ≈ |
| 08 | ◘ | 18 | ┌ | 28 | ( | 38 | 8 | 48 | H | 58 | X | 68 | h | 78 | x | 88 | ê | 98 | ÿ | A8 | ¿ | B8 | Ŋ | C8 | █ | D8 | ∆ | E8 | ɸ | F8 | ○ |
| 09 | ○ | 19 | ┐ | 29 | ) | 39 | 9 | 49 | I | 59 | Y | 69 | i | 79 | y | 89 | Í | 99 | Ö | A9 | ┌ | B9 |  | C9 | ▐ | D9 |  | E9 | Ɵ | F9 | ● |
| 0A | ◙ | 1A | └ | 2A | \* | 3A | : | 4A | J | 5A | Z | 6A | j | 7A | z | 8A | Ó | 9A | Ü | AA | ┐ | BA | ¾ | CA | █ | DA | ω | EA | Ω | FA | · |
| 0B | ♂ | 1B | ┘ | 2B | + | 3B | ; | 4B | K | 5B | [ | 6B | k | 7B | { | 8B | Ú | 9B | ¢ | AB | ½ | BB | ~ | CB | ▒ | DB | ■ | EB | ȴ | FB | √ |
| 0C | ♀ | 1C | X | 2C | , | 3C | < | 4C | L | 5C | \ | 6C | l | 7C | ¦ | 8C | Â | 9C | £ | AC | ¼ | BC | ◊ | CC | ░ | DC | ▄ | EC | ∞ | FC | ⁿ |
| 0D | ♪ | 1D | / | 2D | - | 3D | = | 4D | M | 5D | ] | 6D | m | 7D | } | 8D | Ê | 9D | ¥ | AD | ¡ | BD | ‰ | CD | ▼ | DD | ▌ | ED | ῲ | FD | ² |
| 0E | ♫ | 1E | \ | 2E | . | 3E | > | 4E | N | 5E | ^ | 6E | n | 7E | ~ | 8E | Ô | 9E | ǲ | AE | « | BE | ᴨ | CE | ▲ | DE | ▐ | EE | € | FE | ▌ |
| 0F | ☼ | 1F | ┼ | 2F | / | 3F | ? | 4F | O | 5F | \_ | 6F | o | 7F | ° | 8F | À | 9F | ʄ | AF | » | BF | ɸ | CF | ► | DF | ▀ | EF | ∩ | FF |  |

|  |  |  |
| --- | --- | --- |
| 000 ~ 031 No imprimible |  | 041 ~ 090 Mayúsculas |
| 128 ~ 191 Internacional |  | 097 ~ 122 Minúsculas |
| 192 ~ 255 No imprimible |  | Español |

46

Volvemos a lo mismo: una vez que tenemos el QUE, pasar al COMO esta chupao. Atención:

COGER:

LD A,[HL] ; Cogemos carácter

CP 32

JR C, FIN1 ; Error 1 si es menor de 32

CP ‘A’ ; Lo colocamos en DE sin modificar si menor de 32...

JR C, PONER

CP ‘z’+1 ; ...o mayor o igual que 123

JR NC, PONER

CHKMAY:

CP ‘Z’+1 ; ¿Es mayúscula?

JR NC, CHKMIN ; No: comprobamos si es minúscula

ADD ‘a’-"A" ; Sí: convertimos a mayúscula

JR PONER

CHKMIN:

CP ‘a’ ; ¿Es minúscula?

JR C, PONER ; No: lo colocamos en DE sin modificar

SUB ‘a’-‘A’ ; Sí: convertimos a minúscula

PONER:

LD [DE], A ; Colocamos el carácter, convertido o no, en DE...

INC HL ; ...y seguimos con el bucle

INC DE

DJNZ COGER

De haber empezado a codificar esto sin el esquema previo, probablemente nos habríamos hecho el hacha un lío. Una vez hemos acabado el bucle queda poca cosa por hacer:

LD A,[TERM\_F]

OR A

JR Z, NOTERM

LD A,[TERM] ; Ponemos el carácter de terminación

LD [DE],A ; si la cadena original tenía (TERM\_F=$FF)

NOTERM:

LD A,[LONG]

LD B, A

XOR A ; Terminamos sin error

JR FIN

FIN1:

LD A, 1 ; Finalizaciones con error

JR FIN

FIN2:

LD A, 2

JR FIN

FIN3:

LD A, 3

FIN:

POP DE

POP HL

RET

Observa que, al final del bucle, DE apunta justo después del último carácter de la cadena creada, por lo que podemos poner el carácter de terminación con un simple LD (DE),A. Finalmente cargamos en B la longitud, y terminamos sin error. A continuación colocamos las distintas finalizaciones con error, y después vendrá la subrutina MEDIR y las variables. Añadimos un poco de perejil, y ya tenemos nuestra rutina lista para servir.

Resumiendo...

47

MAYMIN:

; Convierte las mayúsculas de una cadena en minúsculas y viceversa.

; Entrada: HL = Cadena de origen

; DE = Buffer para la cadena de destino

; CY == 0 -> A = Carácter de terminación, B ignorado

; CY == 1 -> B = Longitud de la cadena, A ignorado

; Salida: Cadena convertida en DE, acabada con el mismo carácter que la.

; B = Longitud de la cadena sin contar la terminación

; A = Error: 0 -> No hay error

; 1 -> La cadena contiene códigos de control (<32)

; 2 -> Se han procesado más de 255 carácteres

; 3 -> CY==1 y B==0 a la entrada.

; Registros: C

PUSH HL

PUSH DE

CALL NC, MEDIR

JR NC, FIN2 ; Si MEDIR devuelve CY=0, terminamos con error 2

LD A, B

OR A

JR Z, FIN3 ; Error 3 si la cadena está vacía

LD [LONGIT], A

COGER:

LD A,[HL] ; Cogemos carácter

CP 32

JR C, FIN1 ; Error 1 si es menor de 32

CP ‘A’ ; Lo colocamos en DE sin modificar si menor de 32...

JR C, PONER

CP ‘z’+1 ;...o mayor o igual que 123

JR NC, PONER

CHKMAY:

CP ‘Z’+1 ; ¿Es mayúscula?

JR NC, CHKMIN ; No: comprobamos si es minúscula

AND 11011111b ; Sí: convertimos a mayúscula

JR PONER

CHKMIN:

CP ‘a’ ; ¿Es minúscula?

JR C, PONER ; No: lo colocamos en DE sin modificar

OR 00100000b ; Sí: convertimos a minúscula

PONER:

LD [DE],A ; Colocamos el carácter, convertido o no, en DE...

INC HL ; ...y seguimos con el bucle

INC DE

DJNZ COGER

LD A,[TERM\_F]

OR A

JR Z, NOTERM

LD A,[TERM] ; Ponemos el carácter de terminación

LD [DE], A ; si la cadena original tenía (TERM\_F=#FF)

NOTERM:

LD A,[LONG]

LD B, A

XOR A ; Terminamos sin error

JR FIN

FIN1:

LD A, 1 ; Finalizaciones con error

JR FIN

FIN2:

LD A, 2

JR FIN

FIN3:

LD A, 3

FIN:

POP DE

POP HL

RET

48

MEDIR:

; Devuelve en B la longitud de la cadena HL

; Termina con CY=0 si hay error 2, en caso contrario con CY=1

; Modifica A y C

PUSH HL

LD [TERM], A

LD C, A

LD B, 0

LD A, $FF ; Informamos que CY=0 a la entrada

LD [TERM\_F], A

BUCMED:

LD A,[HL] ; Cogemos carácter

CP C ;¿Es el de terminación?

JR Z, FINMED ; Sí: terminamos

INC B ; No: incrementamos contador...

JR Z, FINM2 ; (¿Error 2?)

INC HL ; ...y pasamos al siguiente carácter

JR BUCMED

FINMED:

POP HL

SCF

RET

FINM2:

POP HL

OR A

RET

; VARIABLES SIEMPRE EN RAM:

ORG $C000 ; Si estamos trabajando con una ROM.

TERM: DS 1

TERM\_F: DS 1

LONGIT: DS 1

Como verás he hecho un par de retoques: he movido la instrucción **LD** [TERM], A al interior de la rutina MEDIR, ya que si no pasamos carácter de terminación no hay por qué usar esta variable; y en cuanto a la conversión mayúscula-minúscula y viceversa con una operación lógica, lo verás claro si observas los códigos ASCII de las mayúsculas y de las minúsculas en formato binario.

¿Qué? ¿Te ha parecido larga la rutina? Je, je, pobrecito... esto no es nada comparado con las rutinas de verdad (es decir, las que sirven para algo) que irás programando a lo largo de tu vida. Pero tú tranquilo, por muy largas que sean no tienes nada que temer si procedes como el Z80 manda, es decir:

- Primero, hay que saber qué es lo que hará exactamente la rutina, y a partir de ahí diseñar la cabecera.

- Debes tener las ideas claras con respecto a las estructuras de datos que vas a manejar, sin escatimar esquemas ayudativos (otra palabra cuya existencia desconozco, pero en RET...).

- A la hora de programar, plantéate antes de cada bloque de instrucciones qué va a hacer ese bloque, sigue un orden y no ahorres comentarios ni variables.

Si así lo haces, con un mucho de práctica (y con información técnica) serás capaz de programar cualquier cosa que te propongas. No, un detector de japonesas no; lo siento, ya lo he intentado yo, pero falta el hard adecuado.

Ya hablaré con Henrik o con Padial...

49

**27 EN EL PROXIMO CAPITULO...** [**Índice**](#INDICE)

Enhorabuena: si has conseguido soportarme hasta este punto, a estas alturas ya eres capaz de programar por tu cuenta, o eso creo. Empieza con cosas sencillitas y ve aumentando la complejidad de tus desarrollos hasta que hayas programado el SD Snatcher 2, entonces avísame, te haré unas cuantas músicas y nos repartimos los beneficios de la venta (a ver si creías que te estaba enseñando ensamblador por gusto...)

Bueno, ahora en serio (o algo menos en broma): aparte de coger práctica, sólo te falta profundizar en la arquitectura del MSX, y eso es lo que haremos a partir del próximo Easymbler. Como aperitivo ahí van un par de rutinas de la BIOS que pueden serte útiles para ir haciendo cosillas:

; BREAKX ($00B7)

; Comprueba la pulsación de CTRL+STOP

; ENTRADA: -

; SALIDA: CF=1 si CTRL+STOP están pulsadas

; MODIFICA: AF

Gracias a esta rutina podrás realizar programas que hagan algo indefinidamente y finalicen al pulsar CTRL+STOP:

START:

...

**CALL** BREAKX

**JR NC**, START

**RET**

Ahí van otras, relacionadas con la lectura del teclado y la salida por pantalla:

; GTSTCK $00D5

; Lee el estado del joystick o los cursores

; ENTRADA: A=Joystick a leer (1 o 2, 0 para los cursores)

; SALIDA: A=Dirección del joystick leída (0 a 8, como STICK del BASIC)

; MODIFICA: Todos los registros

; GTTRIG $00D8

; Lee el estado de los botones del joystick o la barra espaciadora

; ENTRADA: A=Botón a leer (1 o 3: joystick 1, 2 o 4: joystick 2, 0: espacio)

; SALIDA: A=0 -> Botón no pulsado A=$FF -> Botón pulsado

; MODIFICA: AF

; CHGET $009F

; Espera que el usuario pulse una tecla

; ENTRADA: --

; SALIDA: A=Tecla pulsada

; MODIFICA: AF

; CLS $00C3

; Borra la pantalla

; ENTRADA: Z=1

; SALIDA: --

; MODIFICA: AF, BC, DE

; POSIT $00C6

; Posiciona el cursor

ENTRADA: H=Coordenada X L=Coordenada Y

SALIDA: --

MODIFICA: AF

; BEEP $00C0

; Adivina...

; ENTRADA: --

; SALIDA: --

; MODIFICA: Todos los registros

50

Con estas rutinas tienes más que suficiente para realizar programas con un interfaz de usuario básico. No tengas miedo y ponte a investigar, lo peor que te puede pasar (y de hecho te pasará bastante) es colgar al ordenador, pero de algo tiene que comer la tecla RESET, ¿no?

Un par de cosillas más acerca de la grabación de programas en ensamblador para ser usados desde BASIC. Una vez ensamblado (en $A000, habíamos dicho) has de grabar tu programa en formato binario, con $A000 como dirección inicial y de ejecución; como dirección final has de poner la que te dice el ensamblador cuando terminas de ensamblar. Ya puedes usar tus rutinas desde BASIC haciendo lo siguiente:

**CLEAR** 200, &HA000

**BLOAD** "NOMBRE.BIN", R

Para volver a ejecutar una rutina sin tener que volver a cargarla haz:

DEFUSR=&HA000:A=USR(0)

La instrucción **CLEAR** reserva la memoria desde &HA000 hasta el final de la zona de usuario para tus programas en ensamblador, de forma que puedes disponer de esta parte de la memoria con la seguridad de que el intérprete BASIC no la tocará. ¿Y dónde termina la zona de usuario? Depende del ordenador y la cantidad de unidades de disco y programas residentes instalados; de todas formas, si no tocas más allá de $DE00 nunca tendrás problemas.

**28 SE ME ACABO LA CUERDA** [**Índice**](#INDICE)

Ya no tienes excusa. Tienes los conocimientos necesarios para empezar a programar, así que no te escaquees: deja de leer estas chorradas y, ¡¡a machacar el teclado!! ¡¡¡PERO YA!!!

Ahora sí que termino, pero no te hagas ilusiones porque volveréeeee... YIEJ, YIEJ, YIEJ...

TO BE CONTINUED... UNLUCKILY FOR YOU!

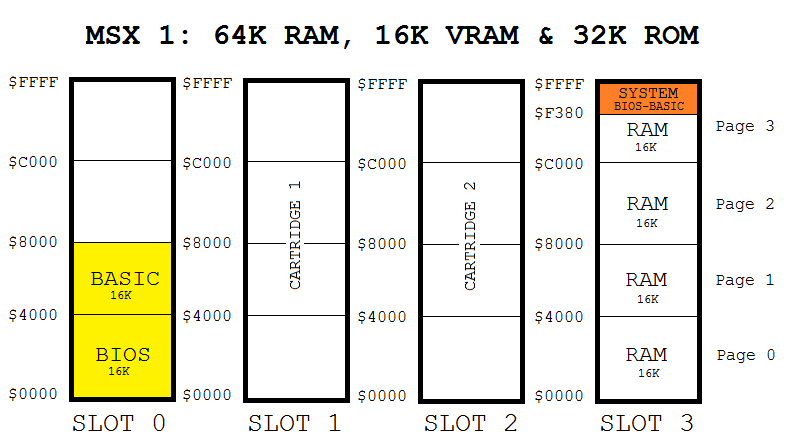
51

**29 CAPERUCITA OBSOLETA** [**Índice**](#INDICE)

Erase que se eran unos señores de esos calvos, con gafas y bata blanca, que saben muuucho de ordenadores y demás parafernalia postmoderna, que hace ya unos cuantos lustros dijeron: "¡Qué aburrimiento! ¿Y si diseñamos una **M**aquinita **S**uper e**X**traña (abreviadamente MSX)?" "¡Fale! Y la basamos en el Z80, que mi primo trabaja en Zilog y me puede conseguir el kilo de Z80 más barato." Y así lo hicieron.

Pero para que la maquinita tardara al menos cinco años y no cinco minutos en quedarse obsoleta, no podían limitarse a conectar una ROM para el BASIC y un poco de RAM al Z80 y ¡hala, a vender! NO. El ordenador debía ser ampliable, tanto internamente (expansiones a elegir por cada fabricante), como externamente (expansiones externas a conectar por el usuario). Además en aquella época el formato más adecuado para programas "serios" era directamente el chip con ROM, que debía poder ser conectada al MSX externamente.

¿Y cómo compaginar todo esto con un pobre Z80 que sólo direcciona 64K de RAM? Pues dándole al coco y diseñando un sistema de bancos de memoria, vulgarmente conocidos como SLOTS. De esta forma, en cada slot puede haber (de hecho suele haber) 64K de algo (RAM, ROM, controlador de periférico) que el Z80 puede conectar a su espacio de direccionamiento y, a partir de ahí, leer y/o escribir tan tranquilamente como quien le estampa una tarta a Bill Gates en el careto.



**30 KORE WA DOO TABERU KA** [**Índice**](#INDICE)

Uséase: ¿esto cómo se come? (si lo he escrito mal -lo más probable- acepto patadas dentales). Pues muy fácil. Pero antes recordemos una vez más (por si te habías dormido) que el MSX divide el espacio de direccionamiento del Z80 en cuatro páginas:

Página 0: $0000 - $3FFF

Página 1: $4000 - $7FFF

Página 2: $8000 - $BFFF

Página 3: $C000 - $FFFF

Todo MSX dispone de cuatro slots, uno de los cuales ha de ser externo (suelen ser dos) para permitir al usuario conectar el Salamander o el F1 Spirit y picarse con el vecino cosa mala; el resto son internos y contienen diversas RAMs y ROMs necesarias para que el MSX sirva para algo más que decorar el armario. Los slots se numeran de 0 a 3.

Cada slot se divide también en cuatro páginas, igual que el Z80. ¿Casualidad? No hijo, no: en todo momento y circunstancia cada página del Z80 está conectada con la página afín (¡del mismo número coño!) de un slot. Pero, ¿de qué slot? Esto se controla mediante el puerto $A8, de la manera siguiente:

52

- Los bits 0 y 1 del valor que contiene el registro $A8 indican el slot conectado a la página 0 del Z80.

- Los bits 2 y 3, lo ídem de la página 1.

- Los bits 4 y 5, lo mismo en la página 2.

- Los bits 6 y 7, que sí que te oigo de la página 3.

Por ejemplo, si haces **OUT** $A8, 10011100b establecerás la siguiente configuración:

- Página 0 del slot 0 en la página 0 del Z80 (xx xx xx 00)

- Página 1 del slot 3 en la página 1 del Z80 (xx xx 11 xx)

- Página 2 del slot 1 en la página 2 del Z80 (xx 01 xx xx)

- Página 3 del slot 2 en la página 3 del Z80 (10 xx xx xx)

¡Y yastá! ¿Ves qué facil? "Pues yo lo he probado y se me cuelga el MSX" ¡¡Claro, burro, no puedes cambiar los slots así a lo loco sin saber lo que haces!! Anda, trae el hacha...

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| PPI (A) Port In/Out $A8 Configura los SLOTS | | | | | | | |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Page 3  $C000 - $FFFF | | Page 2  $8000 - $BFFF | | Page 1  $4000 - $7FFF | | Page 0  $0000 - $3FFF | |
| nº de slot | | nº de slot | | nº de slot | | nº de slot | |

El puerto $A8 es de lectura y escritura. Uséase, para obtener la configuración de slots vigente no tienes más que leer el puerto $A8. O al menos sería así de no ser por los subslots. A ello vamos:

**31 ¡HA GANADO UN SUBSLOT!** [**Índice**](#INDICE)

"Pues mi vecino tiene un expansor de 8 slots y el capullo este va y dice que un MSX sólo puede tener 4." Punto uno: tu padre. Punto dos: no he terminado. Me reitero en que un MSX sólo tiene cuatro slots, pero es que mediante una cabriola hardwarística de no te menees, cada uno de estos cuatro se puede expandir, uséase convertir en otros cuatro (¡los tipos de la bata blanca eran realmente listos!). Entonces no hablamos ya de "el slot X" sino de "el slot X, subslot Y", con Y también de 0 a 3. O, más abreviadamente, de los slots X-0, X-1, X-2 y X-3. Ojo: los subslots NO se añaden al slot primario, sino que lo sustituyen.

Y tú, mente tremendamente inquieta, te preguntarás, ¿quién ganará el mundial de *furgol*?, en cuyo caso te puedes ir directamente a plantar bonsáis en medio del Sáhara. Pero si por casualidad te has equivocado al formular la pregunta y lo que te ha salido es: ¿cómo puedo averiguar si un slot está expandido?, la respuesta es: no lo sé, y no creo que mucha gente lo sepa, pero si preguntas en la zona de trabajo del MSX igual te dicen algo. Sí, parece que la BIOS ya se encarga de tales averiguaciones al inicializar el sistema, y si quieres averiguar si un slot está expandido no tienes más que consultar el bit 7 de las direcciones $FCC1 a $FCC4 para los slots 0 a 3, respectivamente. Si el susodicho bit está a 1, enhorabuena, el slot ha tenido cuatro preciosos subslots.

¿Que cómo se conecta un slot expandido? Bien, siéntate que esto tiene su miga (aunque no tanta), y más de un programa comercial ha dado problemas de incompatibilidad por culpa de un erróneo manejo de los subslots.

Hemos visto que el puerto $A8 controla la conexión de los slots primarios. En el caso de que el slot esté expandido esto no cambia: el puerto $A8 indica entonces el número de slot primario. ¿Y cómo indicamos el subslot? Pues mediante la dirección $FFFF del slot correspondiente. Esta dirección actúa como un puerto, y contiene la siguiente información, similar en cuanto al formato a la del puerto $A8:

- Los bits 0 y 1 contienen el subslot que será conectado en caso de que el slot primario correspondiente sea conectado en la página 0 del Z80 mediante el puerto $A8.

- Los bits 2 y 3, ídem página 1.

- Los bits 4 y 5, ídem página 2.

- Los bits 6 y 7, ídem página 3.

53

¿¿¿ QUEEEEEE ???

Esto... sí, vale, acabo de releer lo que llevo escrito y no me ha quedado muy claro; tienes razón al agarrar de nuevo el hacha, aunque naturalmente no me voy a dejar atrapar (uno es tonto pero no tanto). Así pues vamos a lo importante: ¿cuál es el procedimiento correcto para conectar el slot X-Y? Pues tal que este:

- Lo primero al realizar cambios de slot SIEMPRE es deshabilitar las interrupciones. Si no, no respondo de cuelgues y demás windowerías.

- Leemos el puerto $A8 y variamos los bits necesarios del valor obtenido para conectar el slot X en la página deseada.

En caso de que el slot no esté expandido (consulta el bit 7 de $FCC1+X) escribimos el nuevo valor en el puerto $A8 y ya hemos acabado. Si no...

- Averiguamos qué slot hay conectado a la página 3 mediante la lectura de los bits 6 y 7 del valor obtenido del puerto $A8, y guardamos esta información.

- Conectamos el slot X en la página 3. "¿¡Qué!? ¿Pero esto no era pecado mortal?" Sí, pero como hemos deshabilitado las interrupciones nadie se va a enterar. Venga, sin miedo: pones X en los bits 6 y 7 y mandas el nuevo valor al puerto $A8.

- Leemos el valor de la posición $FFFF (-1 pa los amigos), y lo complementamos. "¿Mande?" Sí, es que para diferenciar esta posición, que en realidad es un puerto, de las direcciones normales, al MSX no se le ocurre nada mejor que complementar (cerar los unos y unar los ceros [¡Has vuelto a fallar! ¡Eres la deshonra del club de hachas!]) el valor del puerto antes de entregarlo. Pero nada, tú plantificas un **CPL** después del **LD** A,[-1] y tan fresco.

- Establecemos los bits necesarios del valor leído (según la página) con Y, y mandamos el nuevo valor a -1 con un simple **LD** [-1],A.

- Restablecemos el slot original de la página 3 (¡¡sobre todo que no se te olvide esto!!)

- Actualizamos la zona de trabajo (hablaremos de esto más adelante)

Y yasta toa. Si simplemente quieres saber qué slot-subslot X-Y hay conectado en una determinada página, la cosa se simplifica un pixel:

- **DI**

- Lectura de $A8, cuyo valor guardamos.

- Averiguamos qué slot hay conectado en la página deseada: ya tenemos X.

- Consultamos el bit 7 de $FCC1+X: si es cero el slot no está expandido, y ya hemos acabado.

- Establecemos X en los bits 6 y 7 (página 3) del valor anteriormente leído de $A8, y lo mandamos de nuevo al mismo sitio. Es decir, conectamos X en la página 3.

- Leemos la dirección -1, la complementamos y de aquí sacamos el valor Y.

- Restauramos el valor original de $A8 para dejar los slots como estaban.

No voy a poner las rutinas necesarias para hacer estas operaciones, por dos razones. Una: será un buen ejercicio que las hagas tú mismo. ¿Ya está? ¿Funcionan? Vale, perfecto. Dos: sonríe a la cámara...

¡¡¡INOCENTEEEE!!!

¡Has picado! Todo esto que te he contado es cierto, pero no es necesario romperse tanto la cabeza, porque ¡la BIOS ya tiene rutinas para realizar cambios de slot! ¡¡JUA JUA!! Ay, qué gracia... eh... ¿no te ríes? ¿No te ha hecho gracia? Bueno, al menos veo que has soltado el hacha... pero ¿qué es eso que coges ahora? Parece una sierra mecánica... oye cálmate... no...

¡¡SOCORROOO!!

Suerte que tenía el Twingo a mano para salir por ruedas. Bien, te preguntarás el porqué de esta putada. No es tal putada: ahora tienes un conocimiento profundo del funcionamiento de los slots de los MSX, lo cual siempre es útil y te puede ayudar más de 34 veces a depurar tus programas. Además, ¿y si por cualquier razón no puedes/no quieres/no \*.\* usar la BIOS? Pues ya conoces los procedimientos correctos para slotear a tus anchas.

54

Bueno, ahora que te has cansado de correr aprovecho para describirte qué croquetas pone la BIOS a tu disposición para manejar los slots:

\* RSLREG $0138 - Lee el registro selector de slots primaro en A.

\* WSLREG $013B - Escribe A en el registro selector de slots primario.

Estas rutinas no tienen mucha gracia, porque lo único que hacen es leer o escribir directamente el puerto $A8 (también llamado en círculos reducidos el registro selector de slots primario). No modifican ningún registro.

Esta otra rutina es bastante más interesante:

\* ENASLT $0024 - Conecta el slot A en la página indicada en HL.

En A debes indicar el slot-subslot a conectar, con el siguiente formato:

E000YYXXb

XX: Número de slot primario

YY: Número de subslot

E: 1 si el slot está expandido

En HL debes establecer cualquier dirección contenida en la página que quieres conectar; o, dicho de una forma más liosa: la página a conectar se indica mediante los bits 6 y 7 del registro H. No te compliques la vida: simplemente usa H=$40 para cambiar la página 1 y H=$80 para la página 2, por ejemplo, y *yastá*.

Ah, que modifica todos los registros.

Además de conectarlos, con los slots se pueden hacer más cosas. ¡Me refiero a leer y escribir su contenido, malpensado! Para ello tenemos a...

\* RDSLT $000C - Lee el contenido de una dirección de un slot.

Entrada: A = Slot, mismo formato que ENASLT

HL = Dirección a leer

Salida: A = Dato leído

Modifica: AF, BC, DE

\* WRSLT $0014 - Escribe en una dirección de un slot.

Entrada: A = Slot, mismo formato que ENASLT

HL = Dirección a escribir

E = Dato a escribir

Salida: ¿Qué salida quieres? ¡Ninguna, hombre!

Modifica: AF, BC, D

Por supuesto, WRSLT sólo funciona si el slot se deja escribir, es decir, si no es ROM o equiparable. Sólo faltaría...

¡No se vayan todavía, aún hay más! (el Saver creía que se decía "una y más", cosa que no entendía y le producía interminables dolores de cabeza. Que se joda, por cabrón). ¿Qué te parecería poder hacer llamadas a rutinas contenidas en otro slot? ¡Con BIOS ya puedes! ¡¡Aprovecha la oferta!! (Este anuncio es de una maquinita obsoleta. No usar en caso de querer mantener una reputación. En caso de cuelgue consulte con Konami Man. Ver código fuente.)

\* CALSLT $001C - Llamada a una rutina en otro slot

Entrada: IYh = Slot, mismo formato que ENASLT

IX = Dirección de la rutina a llamar

Salida: Según la rutina llamada

Modifica: Según la rutina llamada

No, no te esfuerces: si IX es superior a #BFFF la rutina será llamada, pero sin realizar ningún cambio de slot. ¡Que la BIOS no es tonta!

55

Esta otra es bastante curiosa:

CALLF $0030 - Llamada a una rutina en otro slot

Entrada: El slot y la dirección se han de indicar tras la llamada a CALLF, de la siguiente forma:

CALL CALLF

DB Slot

DW Dirección

Esto es bastante útil para llamadas a una rutina situada en una dirección fija de un slot concreto. Si le echas un vistazo a #FD9F, el gancho de la interrupción del reloj, igual te encuentras una llamada al slot de la controladora de disco con este mismo formato. Falen, no me mires así, sólo era una curiosidad...

**32 LA ZONA DE WORKEO** [**Índice**](#INDICE)

Ahora vamos a *boreham* qué nos ofrece la zona de trabajo en cuanto a slots se refiere, que la teníamos un poco olvidada a la pobre...

\* EXPTBL ($FCC1, 4 bytes)

Ya la conoces: indica si cada slot está expandido o no, mediante el bit 7: el slot S está expandido si ($FCC1+S) tiene el bit 7 a uno.

Además, la dirección $FCC1 tiene truki: indica el slot de la BIOS, con el mismo formato que ENASLT. "¿Pero no era siempre el 0?" Hoy por hoy yo diría que sí, pero quién sabe... esta dirección más bien parece una futura aplicación.

\* SLTTBL ($FCC5, 4 bytes)

Contiene una copia de la dirección -1 (el registro de selección de subslot) para cada slot. Así, si quieres saber qué slot hay conectado en determinada página sin pasar por toda la parafernalia que te he explicado antes, sólo tienes que...

- Leer $A8.

- Averiguar qué slot S hay conectado en la página deseada, consultando los bits adecuados (0 y 1 para la página 0, etc.)

- Averiguar qué subslot hay conectado consultando los mismos bits de $FCC5+S.

Mucho más fácil, ¿verdad? Esta es la zona de trabajo que debes actualizar si quieres cambiar de slot "a mano". Por supuesto ENASLT la actualiza automáticamente.

Hay más, pero contienen información más específica sobre el contenido de cada slot que no creo que te haga falta, al menos ahora que estás empezando; además me sé de cierto redactor jefe que me agradecerá este ahorro de líneas, ejem...

¿Y LA SUBROM QUE?

Ah sí, calla calla... no me acordaba de que los MSX2 y superiores disponen de una BIOS secundaria para funciones extendidas, en un slot aparte. Esto lo liquidamos pronto.

Primero: ¿en qué slot está la SUBROM? Pues donde diga EXBRSA $FAF8.

Segundo: ¿cómo se usa? Pues con un par de rutinas de la BIOS:

\* EXTROM $015F - Llamada a una rutina de la SUB-ROM

Entrada: IX = Dirección de la rutina a llamar

Salida: Según la rutina llamada

Modifica: Según la rutina llamada. Los registros alternativos y IY nunca son modificados.

\* SUBROM $015C - Hace lo mismo que EXTROM, pero al acabar hace un POP IX, por lo que antes de llamarla has de hacer un PUSH IX. Mira, se ve que los de ASCII se aburrían...

56

"Y en la SUBROM, ¿qué hay de interesante?" Pues por ejemplo CHGMOD $00D1, que establece el modo gráfico que le metas en A. Desde la BIOS principal sólo puedes establecer SCREEN 0 a 3.

¡¿ Y EL MSX-DOS QUE?! ¡¿EH?!

¡Otia te has dado cuenta! Si en modo MSX-DOS la página 0 está conectada a RAM no se puede usar la BIOS, y si no se puede usar la BIOS no se puede cambiar de slot (de forma facil) ni leer/escribir otros slots ni hacer llamadas a otros slots... ¡Que no panda el *cúnico*! Cuando el modo MSX-DOS está activo, los primeros 256 bytes de RAM (direcciones $0000 a $01FF) están reservados para el sistema, y ¡albricias: algunas de las rutinas de la BIOS para manejo de slots están disponibles! Concretamente:

\* RDSLT $000C

\* WRSLT $0014

\* CALSLT $001C

\* ENASLT $0024

\* CALLF $0030

La dirección de estas rutinas y su funcionamiento es idéntico al de sus congéneres BIOSeras, asín que ya podemos pasar a la última croqueta:

¿QUE ES ESO DEL POKE -1,0/170/255?

¡Ah, caramba! Así que tú también te estabas preguntando desde hace años por qué hay que ponerle un **POKE** -1,algo a ciertas conversiones konamieras y a ciertos juegos de cinta de hace un par de lustros, ¿eh? Pues nada, ya que viene a cuento paso a explicarlo.

Supongamos un juego de cinta de, digamos, el año 87. El juego es para MSX1 con 64K de RAM, y necesita conectar el slot de la ídem en las páginas 0 y/o 1. Como por aquel entonces los subslots no estaban muy de moda (creo que ningún MSX1 tiene), el procedimiento para conectar la RAM usado era este:

- **DI**

- Averiguar qué slot hay conectado en la página 3

- Conectar dicho slot en la página 0, o 1, o ambas, mediante el puerto $A8

Y eso era todo: tenía que funcionar y funcionaba. Pero entonces llegaron los MSX2 que, más chulos que nadie, tenían la RAM en un slot expandido. Pongamos el caso del Philips 8245, que la tenía en el slot 3-2. El juego, al cargarse, conectaba el slot 3 en la página 0 o 1. Pero, ¿qué subslot? Pues depende del valor de la dirección -1 en ese momento, ya que dicha dirección no es modificada. ¿Y qué valor tiene? Pues vete a saber. Cuando el MSX arranca en modo MSX-BASIC conecta el slot de ROM y el BASIC en las páginas 0 y 1, por lo que el valor del selector de subslots del slot de RAM para estas páginas no tiene por qué ser establecido. Uséase: valor aleatorio. Resultado: viene el juego y conecta el slot de RAM pero no se preocupa del subslot, se conecta un ídem aleatorio y ya la hemos liado.

¿La solución? Establecer adecuadamente los 4 bits bajos de la dirección -1 del slot de RAM ANTES de conectar el slot. Esto lo podemos hacer desde BASIC con un simple POKE, dado que lo que hay conectado en las páginas 0 y 1 en ese momento es ROM. ¿Y cómo se establece? Basta ver qué subslot hay conectado en la página 2 o 3 (será el mismo) y establecer este valor para las páginas 0 y 1. Uséase, copiar los 4 bits altos de -1 en los 4 bits bajos.

Ejemplo: en el 8245, la RAM está en el slot 3-2. Entonces el valor leído de -1 es 1010xxxxb, y hay que establecerlo en 10101010b: de aquí el POKE -1,170. En los Sony F700 el slot es el 3-3, y hay que hacer un POKE -1,255. Lo que no todo el mundo sabe es que hay un POKE universal que nos evita todo este guirigai:

POKE -1,((PEEK(-1)XOR&HFF)AND&HF0)\*1.0625

Recuerda que hay que complementar el valor tras leerlo de -1: de aquí el XOR &HFF.

Y con esta duda existencial despejada ya puedes ir a dormir tranquilo, así que...

57

**33 ...ACABOSE** [**Índice**](#INDICE)

"¿Qué? ¡¿Ya?!" Sí, sí; se ha hecho corto, ¿eh? "Hooombreee..." Aceptaré eso como un "Sí" (hay que ver, lo que hace la envidia...) Bueno, esta entrega ha sido más corta porque en caso contrario el jefe habría cogido el enfado que ha superado al enfado del jefe enfadado (Copyright del retorcimiento: Dragon Ball team), y porque ya no incluyo listados. Esto es debido a que si has leído los anteriores Easymbleres (cosa que ni por un ciclo de reloj dudo) ya tienes el nivel suficiente para programar tú solito (como la canción, "cuando yo era pequeñito me hacían las rutinitas, ahora que soy mayorcito las cuelgo yo solito...") y yo ya sólo puedo darte información técnica y cuatro consejos (por ejemplo: envía un generoso donativo a Konami Man, el chaval se lo merece).

En la próxima entrega os machacaré con otro tema de candente actualidad: la memoria mapeada. Y esto va para el Saver: ¡¡CABRON!! (Nótese que ya no se merece la R mayúscula final. ¡Es que encima ya no viene a las rus de Barcelona pa poder irse a Austria! ¡¡Pa matarlo!!). En fin, se despide el envidioso mayor del reino.

///// KONAMI MAN \\\\\

NOTA para despistados y/o no inmersos en la secta: el Saver tiene desde hace tres meses una novia japonesa -residente en Austria-, cosa que ha disparado el nivel de envidia del menda y que le convierte en un CABRON. Desde aquí le deseo sinceramente que se muera.

**34 ¿COMO ESTAN USTEDEEEES?** [**Índice**](#INDICE)

¡Lo que hace el aburrimiento! No hace ni una semana que he entregado la cuarta entrega (valga la rimbombancia) de Easymbler a la autoridad competente, y heme aquí escribiendo ya la quinta. En parte también es una medida de autoprotección, ya que de esta forma te pillo desprevenido y no te doy tiempo a desenfundar y/o afilar el hacha.

Habiendo machacado y convenientemente digerido (supongo) el tema de los slots, pasamos a adentrarnos en otro de los aspectos clave del hardware mesxesiano: la memoria mapeada.

**35 MAIN RAM: 128K** [**Índice**](#INDICE)

Erase que se era un pobre iluso que se leía todos los Easymbler creyendo que así aprendería algo sobre los aspectos más intrínsecos y profundos de su maquinit aobsoleta (pobre...). El caso es que tras meterse entre frente y nuca el rollazo sobre los slots empezó a cavilar tal que así:

"Al encender mi MSX me dice que tiene 128K de RAM. Como en cada slot sólo caben 64K, eso es que debo tener dos slots internos llenos de RAM. Pero mi vecino del trigesimocuarto tiene 512K, por lo que necesita 8 slots; eso debe ser que algún circuito raro los junta todos en uno conectando alguna línea indocumentada del Z80 o algo así... sí, debe ser eso."

En este momento es cuando yo digo que tengo una ampliación de 4 megas, te das cuenta de que para ello son necesarios 64 slots, lo cual es imposible; piensas que te he estado engañando, coges el hacha y ya la hemos vuelto a liar. Como siempre, me vas a permitir que empiece a correr y por el camino te cuento un cuento.

¿LA MEMORIA QUEEE?

Mapeada. O el memory mapper, como dicen los mayores. Se trata de otra pirueta harwarística que permite meter hasta 4 megas (sí, sí: 4096K) de RAM en un mismo slot.

¿Y esto cómo se hace? Pos muy facil. Un slot con memoria RAM mapeada divide esta memoria en segmentos de 16K. Para conectar uno de estos segmentos al espacio de direccionamiento del Z80 lo primero que hay que hacer es conectar el slot a la página deseada (lo cual por otra parte es bastante lógico). Ipso facto seguido se selecciona el segmento por medio de un puerto que el MSX tiene a tal efecto. Bueno, en realidad son cuatro: del $FC al $FF, uno para cada página. Una vez enviado el número de segmento a $FC+P, donde P es la página a la que hemos conectado el slot, ya podemos leer y/o escribir el segmento tranquilamente desde esa página.

Como siempre, la explicación a saco no se entiende (yo al menos no la veo muy clara), así que vamos a ver un ejemplillo. ¿Que queremos conectar el segmento S del slot X en la página 2? Pues hacemos tal que esto:

- Conectamos el slot X en la página 2.

- Conectamos el segmento S a la página 2 tal que así:

LD A, S

OUT [$FE], A

- Ya tenemos el segmento S conectado a las direcciones $8000-$BFFF y listo para leer y escribir.

Como tratamos con puertos de 8 bits el número máximo de segmentos manejables con este sistema es de 256. Si multiplicas este número por las 16K que tiene cada segmento obtendrás los 4096K por slot de máxima.

**36 ¿ASI DE FACIL?** [**Índice**](#INDICE)

Sí... y no. Bueno, de hecho no. El uso incorrecto del mapeador de memoria tiene la culpa de todos los problemas de incompatibilidad de los programas antiguos de DOS 1 cuando corren bajo DOS 2 y/o bajo Turbo-R.

Para empezar, y esto es algo que poca gente sabía hasta hace unos años (yo por ejemplo no lo sabía), los puertos del mapeador (los $FC a $FF) son de sólo escritura. Es decir, oficialmente no puedes averiguar qué segmento hay conectado en una página determinada del Z80 con un simple IN. Por tanto, el procedimiento correcto pasa por guardar en memoria una copia del valor enviado al mapeador cada vez que cambiamos de segmento, y consultar este valor en vez de leer el puerto cuando sea necesario:

; Rutinas para conectar el segmento A en una página

(se supone el slot de RAM ya conectado)

PUT\_P0: LD (STORE\_P0),A

OUT ($FC),A

RET

PUT\_P1: (Idem con $FD y STORE\_P1)

PUT\_P2: (Idem con $FE y STORE\_P2)

PUT\_P3: RET

; Rutinas para averiguar qué segmento hay conectado en una página

GET\_P0: LD A,(STORE\_P0)

RET

GET\_P1: (Idem con STORE\_P1)

GET\_P2: (Idem con STORE\_P2)

GET\_P3: (Idem con STORE\_P3)

; Copia en memoria del valor de los puertos del mapeador

STORE\_P0: .DB 3

STORE\_P1: .DB 2

STORE\_P2: .DB 1

STORE\_P3: .DB 0

Un par de detalles importantes. Uno: el valor inicial de los puertos del mapeador al inicializarse el sistema (¡qué miedo, suena a Windows!) es siempre de 3, 2, 1 y 0 para las páginas 0, 1, 2 y 3, respectivamente. Dos: evidentemente, si habíamos quedado en que no puedes cambiar el slot de la página 3 porque contiene la zona de trabajo, tampoco puedes cambiar su segmento; de ahí el RET en PUT\_P3.

Pero como ya he dicho antes esta restricción de los puertos no se conocía, o sí se conocía pero no se hacía caso. ¿Por qué? Pues porque en todos los MSX2 europeos pasa una cosa bastante curiosa: cuando lees los registros del mapeador obtienes un valor que te permite, mediante una sencilla operación, matar dos Bill Gates de un tiro: averiguar qué segmento hay conectado y cuantos segmentos tiene el slot, es decir, la cantidad de RAM del mismo. Miel sobre japonesas, así que todo el mundo usó este sistema (incluido yo mismo en mis inicios, lo confieso...). Problema: llegan los MSX2+ de Panasonic y los Turbo-R y resulta que el valor leído de los puertos del mapeador cuando haces un IN en estos ordenadores no cumple esta regla "universal" de los MSX europeos (y están en su derecho)... y el asunto explota (uséase, se cuelga) en cuanto ejecutas un programa que lee los puertos para averiguar qué segmento hay conectado: el valor obtenido no es correcto, y al intentar volver a conectar ese segmento, ¡¡boum!!

**37 ¿CUANTA RAM HAS DICHO?** [**Índice**](#INDICE)

A estas alturas tú, persona inquieta y curiosa donde las haya (y nerviosa, no sé dónde te dieron la licencia de hachas), ya te habrás preguntado algo así como esto: *falen*, leer los puertos del mapeador es un deporte de riesgo; entonces, ¿cómo puedo averiguar cuánta RAM mapeada hay en un determinado slot? Pues bien, ¡albricias y alborozo!: eso es precisamente lo que iba a explicar ahora.

Los puertos del mapeador de memoria tienen una característica bastante curiosa (y significativa) que nos vendrá de perlas para satisfacer nuestra curiosidad: son cíclicos. Explicome: si el slot tiene S segmentos de RAM (0 a S-1, recuerda) y mandamos al puerto correspondiente el valor S, lo que obtendremos será no una explosión nuclear ni nada por el estilo, sino el segmento 0 conectado. Si mandamos el valor S+1, tendremos conectado el segmento 1, y así sucesivamente. Ah, esta característica sí es oficial, respira tranquilo.

Vale, ¿Y cómo nos montamos un test de memoria con esto? Pues muy fácil, tal que así:

- Seleccionamos una página, la que más nos guste, por ejemplo la 2, y le conectamos el slot a investigar.

- Hacemos un bucle con S de 0 a 255 que haga tal que esto:

\* Conectar el segmento S a la página 2

\* Escribimos el valor S en el primer byte del segmento, es decir en la dirección $8000

- Conectamos el segmento 0 en la página 2 y leemos el valor de su primer byte(dirección $8000), llamémosle B.

Pues bien, el número de segmentos disponibles en ese mapeador es precisamente 256-B. ¿Sorprendido? ¿Extasiado quizás? ¡Es lógico: Konami Man es insuperable! Y muy modesto, todo hay que decirlo.

¿Cómo? ¿No lo ves claro? "Lo veo clarísimo: Konami Man es un fantasmón de mucho cuidado." Grftx... me refería al test de memoria, so hachoso. Es fácil: como hemos visto antes, el mapeador es cíclico, lo cual significa que el segmento S equivale al 0; pero también el 2\*S, el 3\*S... y así hasta el (N-1)\*S, donde N es el resultado de dividir 256 entre S (aprovecho para recordarte que S siempre es una potencia de dos, por tanto esta división siempre será exacta). Entonces, como siempre escribimos en #8000 y vamos conectando todos los segmentos cíclicamente, el último dato que escribamos en el segmento 0 será precisamente (N-1)\*S, y:

256-B = 256-((N-1)\*S) = 256-(((256/S)-1)\*S) = S

Veo tanto humo salir de tu cabeza como de tus manos, por la enorme presión ejercida sobre el mango del hacha. ¡Tranquilo, que ahora iba a poner un ejemplo clarificador! Supongamos S=4 (64K de RAM, el mapeador más pequeño posible); entonces:

S 2\*S (N-1)\*S

Dato enviado al puerto: 0 1 2 3 4 5 6 7 8 9 10 . . . 250 251 252 253 254 255

Segmento conectado: 0 1 2 3 0 1 2 3 0 1 2 . . . 2 3 0 1 2 3

Así, al conectar el segmento 0 y leer de #8000, obtendríamos el valor 252, y casualmente 256-252=4. ¿Ves como no era tan difícil? ¡Neuras, que eres un neuras!

Atención: este test de memoria es destructivo, es decir, nos cargamos el primer byte de cada segmento. Para hacer las cosas bien, prepara un bufer de 256 bytes (que no ha de estar en la página 2, por supuesto) y ve guardando en él este primer byte de cada segmento antes de machacarlo, para reponerlo al final con un bucle.

**38 DOS 2 RULEZ** [**Índice**](#INDICE)

Ciertamente, la obsolescencia profunda es un estado que confiere una gran paz interior y un alto prestigio en círculos sectario-mesxesianos. Sin embargo, de vez en cuando conviene modernizarse un poco, manque nos pese (¿o era "mal que"? Nunca me acuerdo...)

No, no pretendo que adquieras un hacha electrónica ni nada por el estilo. Estoy hablando de esa maravilla de la operativa sistemática (disciplina encargada de los sistemas operativos (supongo)) que es el MSX-DOS 2. A estas alturas, tú, asaz incrédulo debido a mis desvaríos anteriores, seguramente estarás pensando algo como "¿Eins? Vale, el DOS2 me deja hacer subdirectorios, ¿pero qué tiene que ver esto con la memoria mapeada?" Suerte que estoy aquí para sacarte del ostracismo en el que vives (la frase es de Mato#34).

Resulta que las mejoras que el DOS 2 incorpora no se limitan a la fachada, uséase, ¡no sólo de subdirectorios vive el hombre obsoleto! El DOS 2 también está pensado para facilitar la vida del programador ensambladero (esto ya parece el teletienda). Y resulta que ¡oh, maravillas de la técnica!, tienes a tu disposición una serie de rutinas destinadas a la gestión de la memoria mapeada. De esta forma, igual que gracias al DOS no es necesario conocer los puertos de la disketera para abrir un fichero, lo mismo sucede ahora con la memoria. Y todos tan contentos como el Gili Gates al enterarse de que acaba de ganar ultracientos millones más (¿para qué querrá tantos? Y, si gana tanto vendiendo un sistema operativo que falla por todas partes, ¿cuánto ganaría si vendiera uno que funcionara? Tiemblo sólo del frio que hace).

**39 MEMORIZANDO** [**Índice**](#INDICE)

Basta de autocomplacencia, que veo que tu hacha empieza a cobrar vida de forma inusitada (si es que se escribe asín). Bajo DOS 2 la memoria mapeada se sigue dividiendo en segmentos de 16K (esto es una característica inamovible -qué bien suena esto- del hardware), pero ahora estos segmentos pueden estar en uno de estos tres estados:

- Libre, o sea, que no está siendo usada por ningún programa.

- Reservada en modo usuario, es decir en uso por el programa en ejecución. Cuando el programa termina y el DOS vuelve a tomar el control, el segmento es liberado.

- Reservada en modo sistema: aunque el programa en curso termine, el segmento sigue reservado, y sólo es liberado cuando el usuario lo hace de forma explícita. Este modo es útil para programas residentes, siendo usado por el propio RAMdisk del DOS.

Así pues, el método correcto para usar la memoria mapeada bajo DOS sigue los siguientes pasos:

- Petición de un segmento al DOS.

- Si hay algún segmento libre, el DOS lo marca como reservado, y nos devuelve el número del mismo y su slot: ya podemos usarlo.

- Conexión del segmento al espacio de direccionamiento, lo cual ahora no se hace accediendo directamente a los puertos sino mediante llamadas al sistema específicas. También es posible acceder a un segmento sin conectarlo. En ambos casos, primero hay que conectar el slot que contiene el segmento, lo cual ya sabes hacer gracias al fantástico Easymbler#4.

- Aunque teóricamente no es necesario, es recomendable liberar los segmentos reservados en modo usuario antes de que el programa termine.

De esta forma suprimimos el ciclo vital de dos aves con una única eyección de plomo: ganamos facilidad de uso (se acabaron las engorrosas rutinas para saber cuánta memoria tenemos, y los problemas derivados del acceso directo a los puertos), y gracias al sistema de reserva evitamos conflictos entre programas residentes, ya que ahora es imposible que usen la misma memoria. Pero claro, el problema es que aún corren por ahí muchos programas que no son para DOS 2 y siguen haciendo lo que les da la gana con la memoria, pero en fin, esto ya no es culpa mía...

**40 POSOLOGIA** [**Índice**](#INDICE)

Frena tu impaciencia, oh noble hachero, pues voy a explicarte los detalles para hacer todo esto tal que ya.

El DOS 2 pone a tu disposición dos recursos para gestionar la memoria mapeada: una tabla con información sobre la misma, y una serie de rutinas (llamadas "rutinas de soporte del mapeador", nombre francamante bonito) para reservarla/liberarla y conectarla, entre otras cosas. Veamos primero la tabla, que resulta que tiene tal que este formato:

+0: Slot del mapeador, en el formato típico ExxxSSPP

+1: Número total de segmentos

+2: Número de segmentos libres

+3: Número de segmentos reservados en modo sistema

+4: Número de segmentos reservados en modo usuario

+5..+7: Sin usar

Como mínimo hay una de estas tablas, que es la correspondiente al mapeador primario. Y como creo que antes no lo he hecho, ahora cojo y te explico lo que es el mapeador primario: es el que contiene la página 3, y el que está siempre conectado por defecto en las otras páginas cuando tu programa BASIC o DOS inicia su ejecución. En los MSX2 y 2+, cuando hay más de un slot con RAM mapeada, se establece como mapeador primario el de mayor capacidad (en caso de empate se elige el que tiene el menor número de slot, ¡está todo previsto!). En los Turbo-R, el mapeador primario siempre es el interno, el del slot 3-0.

Hay una tabla como la que has visto para cada mapeador, y están todas colocadas consecutivamente. Por tanto, no tienes más que mirar a partir de la posición +8 de la primera, para encontrarte la siguiente; si lo que te encuentras es el valor 0, es que ya no hay más (porque por muy obsoleto que sea, ningún MSX tiene la RAM en el slot 0. Bueno, creo que el Sony F500 sí, pero este ordenador es mu raro y no se deja ampliar. Allá él... tú no te preocupes, que los papis del DOS 2 sabían lo que se hacían).

¡Las rutinas! Pues están disponibles a través de una tabla de salto muy bonita que es la que tienes a continuación. Además, como son rutinas oficiales y caballeras y todo eso, pues cada una tiene un nombre muy bonito que también incluyo, y gratix (de nada). He aquí la susodicha:

+#0 ALL\_SEG Reserva un segmento

+#3 FRE\_SEG Libera un segmento

+#6 RD\_SEG Lee un byte de un segmento

+#9 WR\_SEG Escribe un byte en un segmento

+#C CAL\_SEG Llama a una rutina en un segmento, pasando la dirección en IYh:IX

+#F CALLS Llama a una rutina en un segmento, pasando la dirección en el propio listado del programa

+#12 PUT\_HL Conecta un segmento en la página indicada por la dirección contenida en HL

+#15 GET\_HL Obtiene el segmento conectado en la página indicada por la dirección contenida en HL

+#18 PUT\_P0 Conecta un segmento en la página 0

+#1B GET\_P0 Obtiene el segmento conectado en la página 0

+#1E PUT\_P1 Conecta un segmento en la página 1

+#21 GET\_P1 Obtiene el segmento conectado en la página 1

+#24 PUT\_P2 Conecta un segmento en la página 2

+#27 GET\_P2 Obtiene el segmento conectado en la página 2

+#2A PUT\_P3 Debería conectar un segmento en la página 3, pero como eso es pecado, pues no hace nada

+#2D GET\_P3 Obtiene el segmento conectado en la página 3

¿Has visto qué bonito? Pues Vicente, bésame la frente; Andrés, bésame los pies; Angulo... ¿dónde vas tan corriendo?

**41 "DONDE ESTAS LOBATON, OIGO TU COMPILAR..."** [**Índice**](#INDICE)

Puedo leer en tu cara, reflejada en tu hacha, algo como esto: "Vale, tenemos dos tablas muy bonitas, pero ¿en qué dirección de memoria están ubicadas? Si no lo sabemos, ¿cómo podemos usarlas?" (Macho, es que tienes una cara complicadilla...)

La respuesta a la primera pregunta es: no lo sé, de hecho las direcciones exactas pueden variar para cada versión del DOS, y sólo puedo asegurarte que estarán en la página 3. Y no creas que me vas a atrapar tan facilmente, pues ya estoy acostumbrado a correr delante de tu hacha y ya voy más rápido que aquel que corría tan rápido (esto de los deportes no es lo mío), pero por si acaso voy a ir respondiéndote a la segunda: es posible averiguar la dirección de inicio de ambas tablas mediante un par de llamadas a la BIOS extendida.

(Entre el párrafo anterior y el corriente han (he) pasado tres días encamado con una amigdalitis tamaño XL y 34+5 de fiebre. ¡Bonita manera de empezar unas vacaciones!)

Por dónde íbamos... ¡ah, la cara de tonto que se te ha quedado con lo de la BIOS extendida! (es broma, no me haches). Explicacioncilla rápida de lo que es el engendro este: como su propio nombre permite barruntar, es un invento que permitía a los desarrolladores (qué raro suena esto, pero si escribes "developpers" quedas como un rey, no es justo...) del sistema sacarse de la manga extensiones de la BIOS (normalmente para manejar extensiones del hard, por ejemplo el RS-232) sin tener que modificar la ROM. ¿Cómo se deglute esto? Fácil: si bien la BIOS clásica se llama a través de un conjunto de direcciones, una para cada rutina, ahora tenemos una sola dirección (concretamente #FFCA, en bonito es EXTBIO), y a través de DE le pasamos el identificador de la rutina que queremos ejecutar. Y como todo el tinglado se gestiona desde RAM y no desde ROM como la BIOS normal, pues es fácilmente actualizable y configurable (toma ya, ni el Güindous al instalarle una rata).

¿Que no has entendido nada? Da igual, sólo era para ponerte en situación y no entra en examen... lo importante viene ahora: existen dos rutinas, accesibles por medio de la BIOS extendida, que nos devuelven las direcciones de las dos tablas aquellas: la de datos sobre el mapeador, y la de la tabla de salto de las rutinas para su manejo. Verbigratia:

\* Obtención de la dirección de la tabla de datos del mapeador:

Entrada: A = 0

DE = #0401

Y tras un CALL #FFCA...

Salida: A = Slot del mapeador primario

HL = Dirección de inicio de la tabla

\* Obtención de la dirección de las rutinas de soporte del mapeador:

Entrada: A = 0

DE = #0402

Y tras un CALL #FFCA...

Salida: A = Número total de segmentos

B = Slot del mapeador primario

C = Número de segmentos libres en el mapeador primario

HL = Dirección de inicio de la tabla de salto

Observa que lo realmente importante en ambos casos es el valor que obtenemos en HL, el resto son simplemente croquetas. A través del valor devuelto en A podemos averiguar si realmente existen las tablas y rutinas de soporte del mapeador: simplemente comprueba si A es distinto de 0 tras el CALL #FFCA, ya

que en caso contrario no hay tales rutinas (seguramente porque no tienes DOS 2), y habrá que volver a la parafernalia del principio del articulillo este...

Así pues parece que la cosa está facilona: haces un par de llamadas a la BIOS extendida, guardas el valor de HL en ambos casos, y ya puedes mapear a gusto:

XOR A

LD DE,#0401

CALL #FFCA

LD (MAP\_TABLA),HL

OR A

JP Z,ERROR ;Para comprobar si realmente tienes las tablas

XOR A

LD DE,#0402

CALL #FFCA

LD (MAP\_RUT),HL

Una vez hecho esto, si por ejemplo quieres averiguar cuántos segmentos libres hay en el mapeador primario, no tienes más que:

LD IX,(MAP\_TABLA)

LD A,(IX+2)

y yastá. Y si por ejemplo quieres obtener el segmento conectado en la página 0, pues... eeeeh... hum...

**42 EL TRUKI DE LAS TRES TREINTA** [**Índice**](#INDICE)

Tá complicadilla la cosa: se trata de hacer un CALL a una dirección que no conocemos a priori (uséase cuando escribimos el programa). ¿Cómo lo hacemos? Hay instrucciones tipo JP (IX) por ahí, pero recordando que en todo el cosmos rige desde hace mucho tiempo la Ley Universal del Mínimo Esfuerzo, nos vamos a sacar de la manga un truquillo que nos permitirá usar las rutinas de soporte con CALLs de los de toda la vida.

Vamos allá, que es fácil. Si haces algo más que afilar tu hacha mientras yo voy vertiendo conocimientos, te habrás fijado en que las rutinas de soporte están separadas entre sí tres bytes. Tienes razón: es imposible meter una rutina en tres bytes. Lo que en realidad tenemos es una tabla de JPs (recuerda que la he llamado "tabla de saltos" un par de veces) a las verdaderas rutinas. Es decir, que lo que obtenemos en HL al llamar a EXTBIO con DE=#0402 es una dirección en la que hay tal que esto:

JP &Hxxxx

JP &Hyyyy

JP &Hzzzz

...

donde &Hxxxx es la verdadera dirección de inicio de ALL\_SEG, &Hyyyy la de FRE\_SEG, &Hzzzz la de RD\_SEG, etc... así pues, tenemos 16 rutinas, a 25 pesetas... perdón, a 3 bytes cada una, 48 bytes en total: a efectos prácticos, estos 48 bytes son para nosotros todas las rutinas de soporte.

Sabiendo esto, deducimos fácilmente (supongo) el truquillo: una vez sabemos dónde está esta tabla de 48 bytes, la copiamos a saco en una dirección conocida, con lo cual ya podremos CALLear a gusto y sin más intermediarios.

¿Que esto tampoco lo has entendido? Macho, entonces definitivamente tu futuro está en Microsoft, por mucho que te duela aceptarlo. Pero tranquilo, para que admires una vez más mi magnanimidad, a continuación tienes el truco este traducido a nemónicos de aplicación directa (consulte a su tecladólogo):

; Obtención de las rutinas de soporte del mapeador

; Versión 2 (truquillo inside)

; Una vez asesinada, digo ejecutada, se puede hacer CALL ALL\_SEG,

; CALL FRE\_SEG, etc, con la tranquilidad de un Vil Gates cualquiera

; absorbiendo empresas cual vampiro cibernético.

XOR A

LD DE,#0402

CALL #FFCA

LD DE,ALL\_SEG

LD BC,48

LDIR

RET

ALL\_SEG: DS 3

FRE\_SEG: DS 3

RD\_SEG: DS 3

WR\_SEG: DS 3

CAL\_SEG: DS 3

CALLS: DS 3

PUT\_PH: DS 3

GET\_PH: DS 3

PUT\_P0: DS 3

GET\_P0: DS 3

PUT\_P1: DS 3

GET\_P1: DS 3

PUT\_P2: DS 3

GET\_P2: DS 3

PUT\_P3: DS 3

GET\_P3: DS 3

**43 CONCRETIZACION DE LAS CROQUETAS** (¿a que no lo dices con la boca llena?)[**Índice**](#INDICE)

El jefe me encargó un texto de 20K y esa longitud ya ha sido sobrepasada, pero aún queda por exponer un aspecto fundamental sobre las rutinas de soporte: sus especificaciones de entrada-salida detalladas, que por supuesto no puedes adivinar ano ser que seas un "Rappelente" (¡Juo juo! ¡Pobre del que tenga que traducir esto !Anda, si es el jefe... esteeee... no, que yo decía que... ¡Mira, un Windows que no se cuelga!) Ahora que está distraído, aprovecho:

\* ALL\_SEG: Reserva de un segmento

Entrada: A = 0 -> Reserva un segmento en modo usuario

A <>0 -> Reserva un segmento en modo sistema

B = 0 -> Reserva un segmento del mapeador primario

B<> 0 -> Reserva un segmento del slot &B ExxxSSPP, donde

PP = slot primario, SS = secundario, E = extendido

(igual que en las rutinas de la BIOS)

El campo "xxx" de B nos permite especificar las preferencias a la hora de elegir el slot del mapeador en el que estará el segmento que queremos reservar, de la forma siguiente:

xxx = 000: Buscar segmentos libres sólo en el slot especificado.

xxx = 001: Buscar segmentos libres sólo en slots que no sean el especificado.

xxx = 010: Buscar segmentos libres en el slot especificado, y si no hay ninguno, buscar en los otros slots.

xxx = 011: Buscar segmentos libres en slots que no sean el especificado, y si no hay ninguno, buscar en el slot especificado.

Recuerda que un valor de 0 para E, SS y PP indica el slot del mapeador primario.

Salida: Cy = 1 si no hay segmentos libres

(según el método de reserva especificado)

Cy = 0 si se ha podido reservar el segmento

A = Número del segmento reservado

B = Slot del mapeador que contiene el segmento reservado

(0, si se especificó E, SS y PP = 0)

\* FRE\_SEG: Libera un segmento previamente reservado

Entrada: A = Número del segmento a liberar

B = Slot del segmento a liberar, &B E000SSPP

(0 para el mapeador primario)

Salida: Cy = 1 si ha habido error

Cy = 0 si el segmento ha sido liberado sin error

\* RD\_SEG: Lee un byte de un segmento

Entrada: A = Número de segmento

HL = Dirección

Salida: A = Valor leído

Los demás registros preservados

Unos cuantos detallitos a tener en cuenta al usar esta rutina:

- Lógicamente, al ser los segmentos de 16K, sólo son válidas las direcciones #0000-#3FFF. Pero si especificas direcciones superiores también son consideradas válidas: simplemente, los rangos de direcciones #4000-#7FFF, #8000-#BFFF y #C000-#FFFF equivalen a #0000-#3FFF.

- El slot del mapeador que contiene el segmento ha de estar conectado en la página 2 previamente. Esta rutina no hace ningún cambio de slot.

- La pila no ha de estar en la página 2 al llamar a esta rutina, suponiendo claro que no te gustan los cuelgues fulminantes.

- Esta rutina deshabilita las interrupciones.

\* WR\_SEG: Escribe un byte en un segmento

Entrada: A = Número de segmento

HL = Dirección

E = Dato a escribir

Salida: Todos los registros preservados excepto A

Échale un ojo a las explicaciones sobre RD\_SEG, que también son aplicables aquí.

\* CAL\_SEG: Llamada a una rutina contenida en un segmento

Entrada: IYh = Número de segmento

IX = Dirección de la rutina

AF, BC, DE, HL serán pasados tal cual a la rutina

Salida: AF, BC, DE, HL, IX, IY tal como los deje la rutina llamada

Detallitos dos, la venganza:

- Ahora cualquier dirección es válida, pero cuidado: el slot del mapeador ha de estar conectado previamente en la página correspondiente a la dirección especificada, ya que CAL\_SEG sólo hace cambios de segmento, no de slot.

- Lógicamente, si la dirección especificada es #C000 o superior, no se realizará ningún cambio de segmento (recuerda que cambiar el slot o segmento de la página 3 es pecado), y CAL\_SEG equivaldrá entonces a un simple CALL.

- Si vas a llamar a una rutina desde la página 0, cuidadito con las interrupciones (al hacer un cambio de segmento en la página 0, perderás el gancho de la interrupción de #0038: cópialo en el segmento a llamar, o bien haz un DI antes de la llamada, si no quieres quedarte ligeramente colgado).

- CAL\_SEG no modifica el estado de las interrupciones.

\* CAL\_SEG: Llamada a una rutina contenida en un segmento,

con especificación del segmento y dirección

mediante incrustación directa en el código

(en inglés se dice "inline", que queda más bonito)

Entrada: AF, BC, DE, HL serán pasados tal cual a la rutina

La secuencia de llamada es tal que así:

CALL CALLS

DB Segmento

DW Dirección

Salida: AF, BC, DE, HL, IX, IY tal como los deje la rutina llamada

En este punto, leerte las explicaciones de CAL\_SEG es beneficioso para la salud de tus programas, y ayuda a regular el nivel de silicio.

\* PUT\_PH: Conecta un segmento en la página correspondiente a la dirección especificada en HL

Entrada: HL = Cualquier dirección de la página deseada

A = Segmento a conectar

Salida: Todos los segmentos preservados

O sea, que esto es justo lo que parece: si HL está entre #0000 y #3FFF, el segmento A será conectado en la página 0; si entre #4000 y #7FFF, pues en la página 1; si entre #8000 y #BFFF, pues en la página 2; y si entre #C000 y #FFFF... pues no, listillo, ni por esas nos vas a pillar ni a mí ni al DOS. En este último caso, PUT\_PH no hace absolutamente nada, ya que el cambio del segmento de la página 3 significaría un descenso directo a los infiernos.

Ah, y que como ya viene siendo habitual, el slot del mapeador correspondiente ha de estar previamente conectado en la página adecuada.

\* GET\_PH: Obtiene el segmento conectado en la página correspondiente a la dirección especificada en HL

Entrada: HL = Cualquier dirección de la página deseada

Salida: A = Segmento conectado

Todos los demás segmentos preservados

El valor devuelto no es leído directamente de los puertos del mapeador, sino que se saca de una tabla interna que tiene el DOS en memoria. De esta forma se evitan problemas con mapeadores rarillos y/o ligeramente incompatibles.

Y que te acuerdes de vigilar las interrupciones cuando manosees la página 0...

\* PUT\_Pn: Conecta un segmento en una página específica (n = 0, 1, 2)

Entrada: A = Segmento a conectar

Salida: Todos los segmentos preservados

Lo de siempre: "Cuando el segmento de tu vecino veas cambiar, pon antes el slot de tu página correspondiente a plantificar".

Y que no, pesao, ¡¡que PUT\_P3 no existe!! Bueno, sí existe, pero no hace nada (ganas de perder el tiempo...)

\* GET\_Pn: Obtiene el slot conectado en una página específica (n = 0, 1, 2, 3)

Entrada: Ninguna (en cómodos plazos)

Salida: A = Segmento conectado

Todos los demás segmentos preservados

En un arrebato de generosidad, resulta que GET\_P3 sí existe, mira tú.

**44 ACABANDO, QUE YA LLEVAMOS 28K** [**Índice**](#INDICE)

Pues eso, resumiendo, que para manejar el la memoria mapeada hay que usar las rutinas de soporte del DOS cuando haberlas haigalas, y cuando no, pues cebolla y agua (o como se diga), y hay que usar rutinas propias como las desparramadas al principio de esta secuencia de códigos ASCII interpretables como un fichero de texto. Recuerda que para averiguar si hay rutinas de soporte del mapeador, hay que comprobar si A=0 tras un CALL $FFCA con A=0 y DE=$0401 o $0402.

Y eso, que así son las cosas obsoletas y así se las hemos incrustado, y que en la próxima entrega de este instrumento de cultura popular igual conoceréis los secretos más profundos de la caligrafía pictográfica en la Patagonia, porque la verdad es que sobre introducción al ensamblador MSX ya no tengo ni pendona idea de sobre qué apabullaros... en fin, ya pensaré algo (una vez al año no hace demasiado daño), manque se admiten sugerencias. ¡Hasta incluso, y feliz navidad/verano! (según cuándo se publique esto).

**45** **CONVERSIÓN DE NUMEROS A ASCII** [**Índice**](#INDICE)

**46** **OPERACIONES CON ENTEROS** [**Índice**](#INDICE)

**47** **OPERACIONES CON PUNTO FIJO** [**Índice**](#INDICE)

**48** **COMPRESIÓN DE DATOS** [**Índice**](#INDICE)

**49** **UN JUEGO SENCILLO** [**Índice**](#INDICE)

**50** **TABLAS Z80** [**Índice**](#INDICE)

\* **CPL**: negación. NOT 0 = 1, NOT 1 = 0, cambia todos los bits.

\* **NEG**: Complemento a dos niega todos los bits y les suma uno. (CPL + INC)

\* **AND**: producto lógico, vale uno sólo si los dos bits son uno:

0 AND 0 = 0

0 AND 1 = 0

1 AND 0 = 0

1 AND 1 = 1

\* **OR**: suma lógica, vale uno si al menos uno de los dos bits vale uno.

0 OR 0 = 0

0 OR 1 = 1

1 OR 0 = 1

1 OR 1 = 1

\* **XOR**: OR exclusiva. El resultado es uno si sólo uno de los bits es uno.

0 XOR 0 = 0

0 XOR 1 = 1

1 XOR 0 = 1

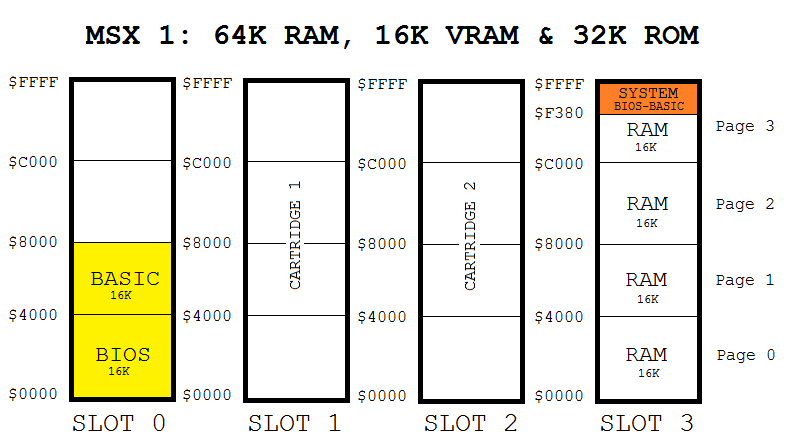
1 XOR 1 = 0

**Los Registros del Z80**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | F | <- **EX** AF,AF' -> | A' | F' | Acumulador y Flags (banderas) |
| B | C |  | B' | C' |  |
| D | E | <- **EXX** -> | D' | E' |  |
| H | L |  | H' | L' |  |
| I | R | Interrupción y Refresco de Memoria |  |  |  |
| IX | | <- Con instrucciones -> | IXh | IXl | Registros de Indexación de 16 bits Ejemp: **LD** H,(IX+n) |
| IY | |  | IYh | IYl |
| SP | | Stack Pointer (puntero de pila) |  |  |  |
| PC | | Program Counter (contador de programa) |  |  |  |

**Las banderas del registro F**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **bit 7** | **bit 6** | **bit 5** | **bit 4** | **bit 3** | **bit 2** | **bit 1** | **bit 0** |
| **S** | **Z** | **5** | **H** | **3** | **P/V** | **N** | **C** |
| signo | cero | bit 5 | acarreo  mitad | bit 3 | paridad  desbord. | resta | acarreo |

**CF** = 1 **SCF** **CF** = 0 **OR** A **ZF** = 1 **CP** A, A **ZF** = 0 **AND** 1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **D** | **H** | **Bin** | **Nombre** | **R** | **G** | **B** |
| 0 | 0 | 0000 | Transp. | 0 | 0 | 0 |
| 1 | 1 | 0001 | Negro | 0 | 0 | 0 |
| 2 | 2 | 0010 | Verde | 33 | 200 | 66 |
| 3 | 3 | 0011 | Verde  Claro | 94 | 220 | 120 |
| 4 | 4 | 0100 | Azul  Oscuro | 84 | 85 | 237 |
| 5 | 5 | 0101 | Azul  Claro | 125 | 118 | 252 |
| 6 | 6 | 0110 | Rojo  Oscuro | 212 | 82 | 77 |
| 7 | 7 | 0111 | Azul  Cielo | 66 | 235 | 245 |
| 8 | 8 | 1000 | Rojo | 252 | 85 | 84 |
| 9 | 9 | 1001 | Rojo  Claro | 255 | 121 | 120 |
| 10 | A | 1010 | Amarillo | 212 | 193 | 84 |
| 11 | B | 1011 | Amarillo  Claro | 230 | 206 | 128 |
| 12 | C | 1100 | Verdo  Oscuro | 33 | 179 | 59 |
| 13 | D | 1101 | Púrpura  Magenta | 201 | 91 | 186 |
| 14 | E | 1110 | Gris | 204 | 204 | 204 |
| 15 | F | 1111 | Blanco | 255 | 255 | 255 |

****

xx

**8-bit Load**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Nemónico** | **Operación** | **Banderas** | | | | | | **B** | **T** | **Comentarios** |
| S | Z | H | PV | N | C |
| LD r, r | r = r | ● | ● | ● | ● | ● | ● | 1 | 4 | r = A,B,C,D,E,H y L |
| LD p,p’ | p = p’ | ● | ● | ● | ● | ● | ● | 2 | 8 | p = A,B,C,D,E,IXh y IXl |
| LD q,q’ | q = q’ | ● | ● | ● | ● | ● | ● | 2 | 8 | q = A,B,C,D,E,IYh y IYl |
| LD r,n | r = n | ● | ● | ● | ● | ● | ● | 2 | 7 | n = número 8 bits |
| LD r,[HL] | r = [HL] | ● | ● | ● | ● | ● | ● | 1 | 7 |  |
| LD [HL],r | [HL] = r | ● | ● | ● | ● | ● | ● | 1 | 7 |  |
| LD A,[BC] | A = [BC] | ● | ● | ● | ● | ● | ● | 1 | 7 |  |
| LD A,[DE] | A = [DE] | ● | ● | ● | ● | ● | ● | 1 | 7 |  |
| LD [BC],A | [BC] = A | ● | ● | ● | ● | ● | ● | 1 | 7 |  |
| LD [DE],A | [DE] = A | ● | ● | ● | ● | ● | ● | 1 | 7 |  |
| LD p,n | p = n | ● | ● | ● | ● | ● | ● | 3 | 11 |  |
| LD q,n | q = n | ● | ● | ● | ● | ● | ● | 3 | 11 |  |
| LD [HL],n | [HL] = n | ● | ● | ● | ● | ● | ● | 2 | 11 |  |
| LD A,[nn] | A = [nn] | ● | ● | ● | ● | ● | ● | 3 | 13 |  |
| LD [nn],A | [nn] = A | ● | ● | ● | ● | ● | ● | 3 | 13 |  |
| LD r,[IX+d] | r = [IX+d] | ● | ● | ● | ● | ● | ● | 3 | 19 |  |
| LD r,[IY+d] | r =[IY+d] | ● | ● | ● | ● | ● | ● | 3 | 19 |  |
| LD [IX+d],r | [IX+d] = r | ● | ● | ● | ● | ● | ● | 3 | 19 |  |
| LD [YX+d],r | [YX+d] = r | ● | ● | ● | ● | ● | ● | 3 | 19 |  |
| LD [IX+d],n | [IX+d] = n | ● | ● | ● | ● | ● | ● | 4 | 19 |  |
| LD [IY+d],n | [IY+d] = n | ● | ● | ● | ● | ● | ● | 4 | 19 | ● No toca nada |
| LD A,I | A = I | ↕ | ↕ | 0 | ? | 0 | ↕ | 2 | 9 | 0 Siempre a 0 |
| LD A,R | A = R | ↕ | ↕ | 0 | ? | 0 | ↕ | 2 | 9 | 1 Siempre a 1 |
| LD I,A | I = A | ● | ● | ● | ● | ● | ● | 2 | 9 | ↕ Cambio normal |
| LD R,A | R = A | ● | ● | ● | ● | ● | ● | 2 | 9 | ? Ni idea, no importa |

**16-bit Load**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Nemónico** | **Operación** | **Banderas** | | | | | | **B** | **T** | **Comentarios** |
| S | Z | H | PV | N | C |
| LD dd, nn | dd = nn | ● | ● | ● | ● | ● | ● | 3 | 10 | dd = BC,DE,HL y SP |
| LD IX, nn | IX = nn | ● | ● | ● | ● | ● | ● | 4 | 14 | nn = número 16 bits |
| LD IY, nn | IY = nn | ● | ● | ● | ● | ● | ● | 4 | 14 |  |
| LD HL, [nn] | H=[nn+1] L=nn | ● | ● | ● | ● | ● | ● | 3 | 16 |  |
| LD dd, [nn] | ddh=[nn+1] ddl=[nn] | ● | ● | ● | ● | ● | ● | 4 | 20 |  |
| LD IX, [nn] | IXh=[nn+1] IXl=[nn] | ● | ● | ● | ● | ● | ● | 4 | 20 |  |
| LD IY, [nn] | IYh=[nn+1] IYl=[nn] | ● | ● | ● | ● | ● | ● | 4 | 20 |  |
| LD [nn], HL | [nn+1]=H [nn]=L | ● | ● | ● | ● | ● | ● | 3 | 16 |  |
| LD [nn], dd | [nn+1]=ddh [nn]=ddl | ● | ● | ● | ● | ● | ● | 4 | 20 |  |
| LD [nn], IX | [nn+1]=IXh [nn]=IXl | ● | ● | ● | ● | ● | ● | 4 | 20 | ● No toca nada |
| LD [nn], IY | [nn+1]=IYh [nn]=IYl | ● | ● | ● | ● | ● | ● | 4 | 20 | 0 Siempre a 0 |
| LD SP, HL | SP = HL | ● | ● | ● | ● | ● | ● | 1 | 6 | 1 Siempre a 1 |
| LD SP, IX | SP = IX | ● | ● | ● | ● | ● | ● | 2 | 10 | ↕ Cambio normal |
| LD SP, IY | SP = IY | ● | ● | ● | ● | ● | ● | 2 | 10 | ? Ni idea, no importa |

**Push / Pop**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Nemónico** | **Operación** | **Banderas** | | | | | | **B** | **T** | **Comentarios** |
| S | Z | H | PV | N | C |
| PUSH qq | [SP-2]=qql [SP-1]=qqh | ● | ● | ● | ● | ● | ● | 1 | 11 | qq = BC,DE,HL y AF |
| PUSH IX | [SP-2]=IXl [SP-1]=IXh | ● | ● | ● | ● | ● | ● | 2 | 15 |  |
| PUSH IY | [SP-2]=IYl [SP-1]=IYh | ● | ● | ● | ● | ● | ● | 2 | 15 |  |
| POP qq | qqh=[SP+1] qql=[SP] | ● | ● | ● | ● | ● | ● | 1 | 10 |  |
| POP IX | IXh=[SP+1] IXl=[SP] | ● | ● | ● | ● | ● | ● | 2 | 14 |  |
| POP IY | IYh=[SP+1] IYl=[SP] | ● | ● | ● | ● | ● | ● | 2 | 14 |  |

xx

**EXchange, Block Transfer & Search**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Nemónico** | **Operación** | **Banderas** | | | | | | **B** | **T** | **Comentarios** |
| S | Z | H | PV | N | C |
| EX DE, HL | DE <-> HL cambia | ● | ● | ● | ● | ● | ● | 1 | 4 |  |
| EX AF, AF’ | AF <-> AF’ cambia | ● | ● | ● | ● | ● | ● | 1 | 4 |  |
| EXX | BC,DE,HL<->BC’,DE’HL’ | ● | ● | ● | ● | ● | ● | 1 | 4 |  |
| EX [SP],HL | H<->[SP+1] L<->[SP] | ● | ● | ● | ● | ● | ● | 1 | 19 |  |
| EX [SP],IX | IXh<->[SP+1] IXl<->[SP] | ● | ● | ● | ● | ● | ● | 2 | 23 |  |
| EX [SP],IY | IYh<->[SP+1] IYl<->[SP] | ● | ● | ● | ● | ● | ● | 2 | 23 |  |
| LDI | [DE]=[HL] copia  DE+=1; HL+=1; BC-=1 | ● | ● | 0 | ♠ | 0 | ● | 2 | 16 | ♠ P/V=0 si BC-1==0 |
| LDIR | [DE]=[HL]  DE+=1; HL+=1; BC-=1  hasta BC==0 | ● | ● | 0 | ♣ | 0 | ● | 2 | 21  16 | 21T si BC!=0  16T si BC==0  ♣ P/V=0 al acabar |
| LDD | [DE]=[HL] copia  DE-=1; HL-=1; BC-=1 | ● | ● | 0 | ♠ | 0 | ● | 2 | 16 |  |
| LDDR | [DE]=[HL] copia  DE-=1; HL-=1; BC-=1  hasta BC==0 | ● | ● | 0 | ♣ | 0 | ● | 2 | 21  16 | 21T si BC!=0  16T si BC==0 |
| CPI | (A-[HL]) busca  HL+=1; BC-=1 | ? | ♥ | ? | ♠ | 1 | ● | 2 | 16 | ♥ Z=1 si A==[HL] |
| CPDIR | (A-[HL]) busca  HL+=1; BC-=1  hasta (A==[HL])||(BC==0) | ? | ♥ | ? | ♠ | 1 | ● | 2 | 2116 | 21T si (BC!=0)&&(A!=[HL])  16T si (BC==0)||(A==[HL]) |
| CPD | (A-[HL]) busca  HL-=1; BC-=1 | ? | ♥ | ? | ♠ | 1 | ● | 2 | 16 |  |
| CPDR | (A-[HL]) busca  HL-=1; BC-=1  hasta (A==[HL])||(BC==0) | ? | ♥ | ? | ♠ | 1 | ● | 2 | 21  16 | 21T si (BC!=0)&&(A!=[HL])  16T si (BC==0)||(A==[HL]) |

**8-Bit Logical**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Nemónico** | **Operación** | **Banderas** | | | | | | **B** | **T** | **Comentarios** |
| S | Z | H | PV | N | C |
| AND r | A&=r | ↕ | ↕ | 1 | P | 0 | 0 | 1 | 4 | r = A,B,C,D,E,H y L |
| AND p | A&=p | ↕ | ↕ | 1 | P | 0 | 0 | 2 | 8 | p = A,B,C,D,E,IXh y IXl |
| AND q | A&=q | ↕ | ↕ | 1 | P | 0 | 0 | 2 | 8 | q = A,B,C,D,E,IYh y IYl |
| AND n | A&=n | ↕ | ↕ | 1 | P | 0 | 0 | 2 | 7 | n = numero 8 bits |
| AND [HL] | A&=[HL] | ↕ | ↕ | 1 | P | 0 | 0 | 1 | 7 |  |
| AND [IX+d] | A&=[IX+d] | ↕ | ↕ | 1 | P | 0 | 0 | 3 | 19 |  |
| AND [IY+d] | A&=[IY+d] | ↕ | ↕ | 1 | P | 0 | 0 | 3 | 19 |  |
| OR r | A|=r | ↕ | ↕ | 0 | P | 0 | 0 | 1 | 4 |  |
| OR p | A|=p | ↕ | ↕ | 0 | P | 0 | 0 | 2 | 8 |  |
| OR q | A|=q | ↕ | ↕ | 0 | P | 0 | 0 | 2 | 8 |  |
| OR n | A|=n | ↕ | ↕ | 0 | P | 0 | 0 | 2 | 7 |  |
| OR [HL] | A|=[HL] | ↕ | ↕ | 0 | P | 0 | 0 | 1 | 7 |  |
| OR [IX+d] | A|=[IX+d] | ↕ | ↕ | 0 | P | 0 | 0 | 3 | 19 |  |
| OR [IY+d] | A|=[IY+d] | ↕ | ↕ | 0 | P | 0 | 0 | 3 | 19 |  |
| XOR r | A^=r | ↕ | ↕ | 0 | P | 0 | 0 | 1 | 4 |  |
| XOR p | A^=p | ↕ | ↕ | 0 | P | 0 | 0 | 2 | 8 |  |
| XOR q | A^=q | ↕ | ↕ | 0 | P | 0 | 0 | 2 | 8 | ● No toca nada |
| XOR n | A^=n | ↕ | ↕ | 0 | P | 0 | 0 | 2 | 7 | 0 Siempre a 0 |
| XOR [HL] | A^=[HL] | ↕ | ↕ | 0 | P | 0 | 0 | 1 | 7 | 1 Siempre a 1 |
| XOR [IX+d] | A^=[IX+d] | ↕ | ↕ | 0 | P | 0 | 0 | 3 | 19 | ↕ Cambio normal |
| XOR [IY+d] | A^=[IY+d] | ↕ | ↕ | 0 | P | 0 | 0 | 3 | 19 | ? Ni idea, no importa |
| CPL | !A NOT(A) | ● | ● | 1 | ● | 1 | ● | 1 | 4 |  |
| NEG | ~A A=-A | ↕ | ↕ | ↕ | V | 1 | ↕ | 2 | 8 |  |
| CCF | !CF NOT(CF) | ● | ● | ♠ | ● | 0 | ↕ | 1 | 4 | ♠ HF is like CF before |
| SCF | CF=1 | ● | ● | 0 | ● | 0 | 1 | 1 | 4 |  |
| DAA | *mejor no usar* | ↕ | ↕ | ↕ | P | ● | ↕ | 1 | 4 | Decimal adjust accumulator |

xx

**8-Bit Add, Substract, Inc & Dec**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Nemónico** | **Operación** | **Banderas** | | | | | | **B** | **T** | **Comentarios** |
| S | Z | H | PV | N | C |
| ADD A, r | A+=r | ↕ | ↕ | ↕ | V | 0 | ↕ | 1 | 4 | r = A,B,C,D,E,H y L |
| ADD A, p | A+=p | ↕ | ↕ | ↕ | V | 0 | ↕ | 2 | 8 | p = A,B,C,D,E,IXh y IXl |
| ADD A, q | A+=q | ↕ | ↕ | ↕ | V | 0 | ↕ | 2 | 8 | q = A,B,C,D,E,IYh y IYl |
| ADD A, n | A+=n | ↕ | ↕ | ↕ | V | 0 | ↕ | 2 | 7 | n = numero 8 bits |
| ADD A, [HL] | A+=[HL] | ↕ | ↕ | ↕ | V | 0 | ↕ | 1 | 7 |  |
| ADD A,[IX+d] | A+=[IX+d] | ↕ | ↕ | ↕ | V | 0 | ↕ | 3 | 19 |  |
| ADD A,[IY+d] | A+=[IY+d] | ↕ | ↕ | ↕ | V | 0 | ↕ | 3 | 19 |  |
| ADC A, r | A+=r+CF | ↕ | ↕ | ↕ | V | 0 | ↕ | 1 | 4 |  |
| ADC A, p | A+=p+CF | ↕ | ↕ | ↕ | V | 0 | ↕ | 2 | 8 |  |
| ADC A, q | A+=q+CF | ↕ | ↕ | ↕ | V | 0 | ↕ | 2 | 8 |  |
| ADC A, n | A+=n+CF | ↕ | ↕ | ↕ | V | 0 | ↕ | 2 | 7 |  |
| ADC A, [HL] | A+=[HL]+CF | ↕ | ↕ | ↕ | V | 0 | ↕ | 1 | 7 |  |
| ADC A,[IX+d] | A+=[IX+d]+CF | ↕ | ↕ | ↕ | V | 0 | ↕ | 3 | 19 |  |
| ADC A,[IY+d] | A+=[IY+d]+CF | ↕ | ↕ | ↕ | V | 0 | ↕ | 3 | 19 |  |
| SUB A, r | A-=r | ↕ | ↕ | ↕ | V | 1 | ↕ | 1 | 4 |  |
| SUB A, p | A-=p | ↕ | ↕ | ↕ | V | 1 | ↕ | 2 | 8 |  |
| SUB A, q | A-=q | ↕ | ↕ | ↕ | V | 1 | ↕ | 2 | 8 |  |
| SUB A, n | A-=n | ↕ | ↕ | ↕ | V | 1 | ↕ | 2 | 7 |  |
| SUB A, [HL] | A-=[HL] | ↕ | ↕ | ↕ | V | 1 | ↕ | 1 | 7 |  |
| SUB A,[IX+d] | A-=[IX+d] | ↕ | ↕ | ↕ | V | 1 | ↕ | 3 | 19 |  |
| SUB A,[IY+d] | A-=[IY+d] | ↕ | ↕ | ↕ | V | 1 | ↕ | 3 | 19 |  |
| SBC A, r | A-=r-CF | ↕ | ↕ | ↕ | V | 1 | ↕ | 1 | 4 |  |
| SBC A, p | A-=p-CF | ↕ | ↕ | ↕ | V | 1 | ↕ | 2 | 8 |  |
| SBC A, q | A-=q-CF | ↕ | ↕ | ↕ | V | 1 | ↕ | 2 | 8 |  |
| SBC A, n | A-=n-CF | ↕ | ↕ | ↕ | V | 1 | ↕ | 2 | 7 |  |
| SBC A, [HL] | A-=[HL]-CF | ↕ | ↕ | ↕ | V | 1 | ↕ | 1 | 7 |  |
| SBC A,[IX+d] | A-=[IX+d]-CF | ↕ | ↕ | ↕ | V | 1 | ↕ | 3 | 19 |  |
| SBC A,[IY+d] | A-=[IY+d]-CF | ↕ | ↕ | ↕ | V | 1 | ↕ | 3 | 19 |  |
| INC r | ++r | ↕ | ↕ | ↕ | V | 0 | ● | 1 | 4 | r = A,B,C,D,E,H y L |
| INC p | ++p | ↕ | ↕ | ↕ | V | 0 | ● | 2 | 8 | p = A,B,C,D,E,IXh y IXl |
| INC q | ++q | ↕ | ↕ | ↕ | V | 0 | ● | 2 | 8 | q = A,B,C,D,E,IYh y IYl |
| INC [HL] | ++[HL] | ↕ | ↕ | ↕ | V | 0 | ● | 1 | 11 | n = numero 8 bits |
| INC [IX+d] | ++[IX+d] | ↕ | ↕ | ↕ | V | 0 | ● | 3 | 23 |  |
| INC [IY+d] | ++[IY+d] | ↕ | ↕ | ↕ | V | 0 | ● | 3 | 23 |  |
| DEC r | --r | ↕ | ↕ | ↕ | V | 1 | ● | 1 | 4 |  |
| DEC p | --p | ↕ | ↕ | ↕ | V | 1 | ● | 2 | 8 |  |
| DEC q | --q | ↕ | ↕ | ↕ | V | 1 | ● | 2 | 8 |  |
| DEC [HL] | --[HL] | ↕ | ↕ | ↕ | V | 1 | ● | 1 | 11 |  |
| DEC [IX+d] | --[IX+d] | ↕ | ↕ | ↕ | V | 1 | ● | 3 | 23 |  |
| DEC [IY+d] | --[IY+d] | ↕ | ↕ | ↕ | V | 1 | ● | 3 | 23 |  |

**16-Bit Add, Substract, Inc & Dec**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Nemónico** | **Operación** | **Banderas** | | | | | | **B** | **T** | **Comentarios** |
| S | Z | H | PV | N | C |
| ADD HL, ss | HL+=ss | ● | ● | ↕ | ● | 0 | ↕ | 1 | 11 | ss = BC, DE, HL y SP |
| ADD IX, pp | IX+=pp | ● | ● | ↕ | ● | 0 | ↕ | 2 | 15 | pp = BC, DE, IX y SP |
| ADD IY, qq | IY+=qq | ● | ● | ↕ | ● | 0 | ↕ | 2 | 15 | qq = BC, DE, Iy y SP |
| ADC HL, ss | HL+=ss+CF | ↕ | ↕ | ↕ | V | 0 | ↕ | 2 | 15 |  |
| SBC HL, ss | HL-=ss-CF | ↕ | ↕ | ↕ | V | 0 | ↕ | 2 | 15 |  |
| INC ss | ++ss | ● | ● | ● | ● | ● | ● | 1 | 6 |  |
| INC IX | ++IX | ● | ● | ● | ● | ● | ● | 2 | 10 | ● No toca nada |
| INC IY | ++IY | ● | ● | ● | ● | ● | ● | 2 | 10 | 0 Siempre a 0 |
| DEC ss | ++ss | ● | ● | ● | ● | ● | ● | 1 | 6 | 1 Siempre a 1 |
| DEC IX | ++IX | ● | ● | ● | ● | ● | ● | 2 | 10 | ↕ Cambio normal |
| DEC IY | ++IY | ● | ● | ● | ● | ● | ● | 2 | 10 | ? Ni idea, no importa |

xx

**Jump**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Nemónico** | **Operación** | **Banderas** | | | | | | **B** | **T** | **Comentarios** |
| S | Z | H | PV | N | C |
| CP A, r | (A-r) compara | ↕ | ↕ | ↕ | V | 1 | ↕ | 1 | 4 | si 0 Z=1 C=0 |
| CP A, p | (A-p) | ↕ | ↕ | ↕ | V | 1 | ↕ | 2 | 8 | si >0 Z=0 C=0 |
| CP A, q | (A-q) | ↕ | ↕ | ↕ | V | 1 | ↕ | 2 | 8 | si <0 Z=0 C=1 |
| CP A, n | (A-n) | ↕ | ↕ | ↕ | V | 1 | ↕ | 2 | 7 | n = numero 8 bits |
| CP A, [HL] | (A-[HL]) | ↕ | ↕ | ↕ | V | 1 | ↕ | 1 | 7 | nn = numero 16 bits |
| CP A,[IX+d] | (A-[IX+d]) | ↕ | ↕ | ↕ | V | 1 | ↕ | 3 | 19 | cc=Z,NZ,C,NC,PO,PE,P,M |
| CP A,[IY+d] | (A-[IY+d]) | ↕ | ↕ | ↕ | V | 1 | ↕ | 3 | 19 | **Z** Cero **NZ** No Cero |
| JP nn | goto nn; PC=nn | ● | ● | ● | ● | ● | ● | 3 | 10 | **C** Acarreo **NC** No Acarreo |
| JP [HL] | goto [HL] | ● | ● | ● | ● | ● | ● | 1 | 4 | **PE** Paridad Par |
| JP [IX] | goto [IX] | ● | ● | ● | ● | ● | ● | 2 | 8 | **PO** Paridad Impar |
| JP [IY] | goto [IY] | ● | ● | ● | ● | ● | ● | 2 | 8 | **P** Positivo |
| JP cc, nn | if(cc) goto nn | ● | ● | ● | ● | ● | ● | 3 | 10 | **M** Negativo |
| JR e | goto PC+e; PC+=e | ● | ● | ● | ● | ● | ● | 2 | 12 | e = 129 a -127 |
| JR ss, e | if(ss) goto PC+e | ● | ● | ● | ● | ● | ● | 2 | 127 | ss = C, NC, Z y NZ  12T si (ss==true) |
| DJNZ e | --B;  if(B!=0) goto PC+e | ● | ● | ● | ● | ● | ● | 2 | 13  8 | 13T si (B!=0)  8T si (B==0) |
| CALL nn | funcion(nn) | ● | ● | ● | ● | ● | ● | 3 | 17 | nn = numero 16 bits |
| CALL cc, nn | if(cc) funcion(nn) | ● | ● | ● | ● | ● | ● | 3 | 17  10 | 17T si (cc==true)  10T si (cc==false) |
| RET | return | ● | ● | ● | ● | ● | ● | 1 | 10 |  |
| RET cc | if(cc) return | ● | ● | ● | ● | ● | ● | 1 | 11  5 | 11T si (cc==true)  5T si (cc==false) |
| RST p | funcion(p) | ● | ● | ● | ● | ● | ● | 1 | 11 | p=$0,$8,$10,$18,$20,$28,$30,$38 |
| NOP | nada | ● | ● | ● | ● | ● | ● | 1 | 4 |  |
| HALT | espera interrupción | ● | ● | ● | ● | ● | ● | 1 | 4 | HALT + DI = CUELGUE |
| DI | desactiva interrup. | ● | ● | ● | ● | ● | ● | 1 | 4 | ganamos velocidad |
| EI | activa interrup. | ● | ● | ● | ● | ● | ● | 1 | 4 |  |
| IM 0 | *no usado por MSX* | ● | ● | ● | ● | ● | ● | 2 | 8 | *no usado por el MSX* |
| IM 1 | interrup. tipo 1 | ● | ● | ● | ● | ● | ● | 2 | 8 | salta a $0038 |
| IM 2 | *no usado por MSX* | ● | ● | ● | ● | ● | ● | 2 | 8 | *no usado por el MSX* |
| RETI | return interrup. | ● | ● | ● | ● | ● | ● | 2 | 14 | retorna de interrupción |
| RETN | *no usado por MSX* | ● | ● | ● | ● | ● | ● | 2 | 14 | *no usado por el MSX* |

**Bit Set, Reset & Test**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Nemónico** | **Operación** | **Banderas** | | | | | | **B** | **T** | **Comentarios** |
| S | Z | H | PV | N | C |
| BIT b, r | ZF=!(rb) | ↕ | ↕ | 1 | ↕ | 0 | ● | 2 | 8 | r=A,B,C,D,E,H,L y A |
| BIT b, [HL] | ZF=!(HLb) | ↕ | ↕ | 1 | ↕ | 0 | ● | 2 | 12 | b = bit  000 0  001 1  010 2  011 3  100 4  101 5  110 6  111 7 |
| BIT b,[IX+d] | ZF=!(IX+d)b | ↕ | ↕ | 1 | ↕ | 0 | ● | 4 | 20 |
| BIT b,[IY+d] | ZF=!(IY+d)b | ↕ | ↕ | 1 | ↕ | 0 | ● | 4 | 20 |
| SET b, r | rb=1 | ● | ● | ● | ● | ● | ● | 2 | 8 |
| SET b, [HL] | [HL]b=1 | ● | ● | ● | ● | ● | ● | 2 | 15 |
| SET b,[IX+d] | [IX+d]b=1 | ● | ● | ● | ● | ● | ● | 4 | 23 |
| SET b,[IY+d] | [IY+d]b=1 | ● | ● | ● | ● | ● | ● | 4 | 23 |
| SET b,[IX+d],r | r=[IX+d]  rb=1; [IX+d]=r | ● | ● | ● | ● | ● | ● | 4 | 23 |
| SET b,[IY+d],r | r=[IY+d]  rb=1; [IY+d]=r | ● | ● | ● | ● | ● | ● | 4 | 23 |
| RES b, r | rb=0 | ● | ● | ● | ● | ● | ● | 2 | 8 | ● No toca nada  0 Siempre a 0  1 Siempre a 1  ↕ Cambio normal  ? Ni idea, no importa |
| RES b, [HL] | [HL]b=0 | ● | ● | ● | ● | ● | ● | 2 | 15 |
| RES b,[IX+d] | [IX+d]b=0 | ● | ● | ● | ● | ● | ● | 4 | 23 |
| RES b,[IY+d] | [IY+d]b=0 | ● | ● | ● | ● | ● | ● | 4 | 23 |
| RES b,[IX+d],r | r=[IX+d]  rb=0; [IX+d]=r | ● | ● | ● | ● | ● | ● | 4 | 23 |
| RES b,[IY+d],r | r=[IY+d]  rb=0; [IY+d]=r | ● | ● | ● | ● | ● | ● | 4 | 23 |

xx

**Input & Output**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Nemónico** | **Operación** | **Banderas** | | | | | | **B** | **T** | **Comentarios** |
| S | Z | H | PV | N | C |
| IN A, [n] | A=[n] | ● | ● | ● | ● | ● | ● | 2 | 11 | r=A,B,C,D,E,H,L y A |
| IN r, [C] | r=[C] | ↕ | ↕ | 0 | P | 0 | ● | 2 | 12 |  |
| IN F, [n] | <-[C] | ↕ | ↕ | 0 | P | 0 | ● | 2 | 12 |  |
| INI | [HL]=[C]; ++HL; --B | ♠ | ♠ | ? | ? | ♣ | ? | 2 | 16 | 1 ♠ igual que en DEC B |
| INIR | [HL]=[C]; ++HL; --B  hasta (B==0) | 0 | 1 | ? | ? | ♣ | ? | 2 | 21  16 | T21 si (B!=0)  T15 si (B==0) |
| IND | [HL]=[C];  --HL; --B | ♠ | ♠ | ? | ? | ♣ | ? | 2 | 16 | 2 ♣ copia del bit 7 |
| INDR | [HL]=[C]; --HL; --B  hasta (B==0) | 0 | 1 | ? | ? | ♣ | ? | 2 | 21  16 | T21 si (B!=0)  T16 si (B==0) |
| OUT [n], A | [n]=A | ● | ● | ● | ● | ● | ● | 2 | 11 |  |
| OUT [C], r | [C]=r | ● | ● | ● | ● | ● | ● | 2 | 12 |  |
| OUT [C], 0 | [C]=0 | ● | ● | ● | ● | ● | ● | 2 | 12 |  |
| OUTI | [C]=[HL]; ++HL; --B | ♠ | ♠ | ? | ? | ♣ | ? | 2 | 16 |  |
| OTIR | [C]=[HL]; ++HL; --B  hasta (B==0) | 0 | 1 | ? | ? | ♣ | ? | 2 | 21  16 | T21 si (B!=0)  T16 si (B==0) |
| OUTD | [C]=[HL]; --HL; --B | ♠ | ♠ | ? | ? | ♣ | ? | 2 | 16 |  |
| OTDR | [C]=[HL]; --HL; --B  hasta (B==0) | 0 | 1 | ? | ? | ♣ | ? | 2 | 21  16 | T21 si (B!=0)  T16 si (B==0) |

**Rotate and Shift**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Nemónico** | **Operación** | **Banderas** | | | | | | **B** | **T** | **Comentarios** |
| S | Z | H | PV | N | C |
| SLA m |  | ↕ | ↕ | 0 | P | 0 | ↕ | 2 | 8 | r=A,B,C,D,E,H,L y A |
| SLL m |  | ↕ | ↕ | 0 | P | 0 | ↕ | 2 | 8 | m=r,[HL],[IX+d],[IY+d] |
| SRA m |  | ↕ | ↕ | 0 | P | 0 | ↕ | 2 | 8 |  |
| SRL m |  | ↕ | ↕ | 0 | P | 0 | ↕ | 2 | 8 |  |
| RLCA |  | ● | ● | 0 | ● | 0 | ↕ | 1 | 4 |  |
| RLC r |  | ↕ | ↕ | 0 | P | 0 | ↕ | 2 | 8 |  |
| RLC [HL] |  | ↕ | ↕ | 0 | P | 0 | ↕ | 2 | 15 |  |
| RLC [IX+d] |  | ↕ | ↕ | 0 | P | 0 | ↕ | 4 | 23 |  |
| RLC [IY+d] |  | ↕ | ↕ | 0 | P | 0 | ↕ | 4 | 23 |  |
| RLC [IX+d],r | r=[IX+d]; RLC r  [IX+d]=r | ↕ | ↕ | 0 | P | 0 | ↕ | 4 | 23 |  |
| RLC [IY+d],r | r=[IY+d]; RLC r  [IY+d]=r | ↕ | ↕ | 0 | P | 0 | ↕ | 4 | 23 |  |
| RRCA |  | ● | ● | 0 | ● | 0 | ↕ | 1 | 4 |  |
| RRC m |  | ↕ | ↕ | 0 | P | 0 | ↕ | 2 | 8 |  |
| RLA |  | ● | ● | 0 | ● | 0 | ↕ | 1 | 4 |  |
| RL m |  | ↕ | ↕ | 0 | P | 0 | ↕ | 2 | 8 | ● No toca nada |
| RRA |  | ● | ● | 0 | ● | 0 | ↕ | 1 | 4 | 0 Siempre a 0 |
| RR m |  | ↕ | ↕ | 0 | P | 0 | ↕ | 2 | 8 | 1 Siempre a 1 |
| RLD |  | ↕ | ↕ | 0 | P | 0 | ● | 2 | 18 | ↕ Cambio normal |
| RRD |  | ↕ | ↕ | 0 | P | 0 | ● | 2 | 18 | ? Ni idea, no importa |

xx

**51 VARIABLES BIOS** [**Índice**](#INDICE)

\* Subroutines for read/write calls of the inter-slot

RDPRIM (F380H, 5)

contents: read from basic slot

WRPRIM (F385H, 7)

contents: write to basic slot

CLPRIM (F38CH, 14)

contents: basic slot call

\* Starting address of assembly language program of USR function, text screen

USRTAB (F39AH, 20)

initial value: FCERR

contents: starting address of assembly language program of USR function

(0 to 9); the value before defining assembly language program

points to FCERR (475AH).

LINL40 (F3AEH, 1)

initial value: 39

contents: screen width per line at SCREEN 0 (set by WIDTH statement

at SCREEN 0)

LINL32 (F3AfH, 1)

initial value: 32

contents: screen width per line at SCREEN 1 (set by WIDTH statement

at SCREEN 1)

LINLEN (F3B0H, 1)

initial value: 29

contents: current screen width per line

CRTCNT (F3B1H, 1)

initial value: 24

contents: number of lines of current screen

CLMLST (F3B2H, 1)

initial value: 14

contents: horizontal location in the case that items are divided by

commas in PRINT statement

\* Work for initialisation

- SCREEN 0

TXTNAM (F3B3H, 2)

initial value: 0000H

contents: pattern name table

TXTCOL (F3B5H, 2)

contents: unused

TXTCGP (F3B7H, 2)

initial value: 0800H

contents: pattern generator table

TXTATR (F3B9H, 2)

contents: unused

TXTPAT (F3BBH, 2)

contents: unused

- SCREEN 1

T32NAM (F3BDH, 2)

initial value: 1800H

contents: pattern name table

T32COL (F3BFH, 2)

initial value: 2000H

contents: colour table

T32CGP (F3C1H, 2)

initial value: 0000H

contents: pattern generator table

T32ATR (F3C3H, 2)

initial value: 1B00H

contents: sprite attribute table

T32PAT (F3C5H, 2)

initial value: 3800H

contents: sprite generator table

- SCREEN 2

GRPNAM (F3C7H, 2)

initial value: 1800H

contents: pattern name table

GRPCOL (F3C9H, 2)

initial value: 2000H

contents: colour table

GRPCGP (F3CBH, 2)

initial value: 0000H

contents: pattern generator table

GRPATR (F3CDH, 2)

initial value: 1B00H

contents: sprite attribute table

GRPPAT (F3CFH, 2)

initial value: 3800H

contents: sprite generator table

- SCREEN 3

MLTNAM (F3D1H, 2)

initial value: 0800H

contents: pattern name table

MLTCOL (F3D3H, 2)

contents: unused

MLTCGP (F3D5H, 2)

initial value: 0000H

contents: pattern generator table

MLTATR (F3D7H, 2)

initial value: 1B00H

contents: sprite attribute table

MLTPAT (F3D9H, 2)

initial value: 3800H

contents: sprite generator table

\* Other screen settings

CLIKSW (F3DBH, 1)

initial value: 1

contents: key click switch (0 = OFF, otherwise = ON), set by

<key click switch> of SCREEN statement

CSRY (F3DCH, 1)

initial value: 1

contents: Y-coordinate of cursor

CSRX (F3DDH, 1)

initial value: 1

contents: X-coordinate of cursor

CNSDFG (F3DEH, 1)

initial value: 0

contents: function key display switch (0 = display, otherwise = no

display), set by KEY ON/OFF statement

\* Area to save VDP registers

RG0SAV (F3DFH, 1)

initial value: 0

RG1SAV (F3E0H, 1)

initial value: E0H

RG2SAV (F3E1H, 1)

initial value: 0

RG3SAV (F3E2H, 1)

initial value: 0

RG4SAV (F3E3H, 1)

initial value: 0

RG5SAV (F3E4H, 1)

initial value: 0

RG6SAV (F3E5H, 1)

initial value: 0

RG7SAV (F3E6H, 1)

initial value: 0

STATFL (F3E7H, 1)

initial value: 0

contents: stores VDP status (contents of status register 0, in MSX2)

TRGFLG (F3E8H, 1)

initial value: FFH

contents: stores trigger button status of joystick

FORCLR (F3E9H, 1)

initial value: 15

contents: foreground colour; set by colour statement

BAKCLR (F3EAH, 1)

initial value: 4

contents: background colour; set by colour statement

BDRCLR (F3EBH, 1)

initial value: 7

contents: border colour; set by colour statement

MAXUPD (F3ECH, 3)

initial value: JP 0000H (C3H, 00H, 00H)

contents: used by CIRCLE statement internally

MINUPD (F3EFH, 3)

initial value: JP 0000H (C3H, 00H, 00H)

contents: used by CIRCLE statement internally

ATRBYT (F3F2H, 1)

initial value: 15

contents: colour code in using graphics

\* Work area for PLAY statement

QUEUES (F3F3H, 2)

initial value: QUETAB (F959H)

contents: points to queue table at the execution of PLAY statement

FRCNEW (F3F5H), 1)

initial value: 255

contents: used by BASIC interpreter internally

\* Work area for key input

SCNCNT (F3F6H, 1)

initial value: 1

contents: interval for the key scan

REPCNT (F3F7H, 1)

initial value: 50

contents: delay until the auto-repeat of the key begins

PUTPNT (F3F8H, 2)

initial value: KEYBUF (FBF0H)

contents: points to address to write in the key buffer

GETPNT (F3FAH, 2)

initial value: KEYBUF (FBF0H)

contents: points to address to read from key buffer

CLOC (F92AH, 2)

contents: used internally by graphic routine

**52 PPI POR BIOS** [**Índice**](#INDICE)

**53 PSG POR BIOS** [**Índice**](#INDICE)

**54 VDP POR BIOS** [**Índice**](#INDICE)

**55 PPI POR PUERTOS** [**Índice**](#INDICE)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| PPI (A) Port In/Out $A8 Configura los SLOTS | | | | | | | |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Page 3  $C000 - $FFFF | | Page 2  $8000 - $BFFF | | Page 1  $4000 - $7FFF | | Page 0  $0000 - $3FFF | |
| nº de slot | | nº de slot | | nº de slot | | nº de slot | |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | PPI (B) Port In $A9 Keyboard Column | | | | | | | | |
| PPI (C)  Port Out  $AA  Keyboard  Row |  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0000 | &  7 | ^  6 | %  5 | $  4 | #  3 | @  2 | !  1 | )  0 |
| 0001 | :  ; | }  ] | {  [ | |  \ | +  = | - | (  9 | \*  8 |
| 0010 | B | A | ACENT | ?  / | >  . | <  , | ~  ` | “  ‘ |
| 0011 | J | I | H | G | F | E | D | C |
| 0100 | R | Q | P | O | N | M | L | K |
| 0101 | Z | Y | X | W | V | U | T | S |
| 0110 | F3 | F2 | F1 | CODE | CAPS | GRAPH | CTRL | SHIFT |
| 0111 | RETURN | SELECT | BS | STOP | TAB | ESC | F5 | F4 |
| 1000 | → | ↓ | ↑ | ← | DEL | INS | HOME  CLS | SPACE |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| PPI (C) Port Out $AA | | | | | | | |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Key Click | Cap  LED | Cas  Out | Cas  Motor | Keyboard Row | | | |
| 0=On  1=Off | 0=On  1=Off | 0=On  1=Off | 0=On  1=Off | [ 0000 ~ 1000 ] | | | |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| PPI Mode Port $ABH | | | | | | | |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | Mode A &  C upper four bits | | A $A8  Dir | C $AA  Dir upper four bits | Mode B &  C lower four bits | B $A9  Dir | C $AA  Dir lower four bits |
| 00=Normal Mode  01=Strobed Mode  10=Bidirectional | | 0=Output  1=Input | 0=Output  1=Input | 0=Normal Mode 1=Strobed Mode | 0=output  1=Input | 0=Output  1=Input |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| PPI Mode Port $ABH | | | | | | | |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | Not used | | | Bit Number | | | Set |
|  | | | 00=0  01=1  10=2  11=3 | | | 1=Set  0=Reset |

**56 PSG POR PUERTOS** [**Índice**](#INDICE)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| REGISTRO / BIT | | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | PERIODO CANAL A | AJUSTE FINO | | | | | | | |
| 1 |  |  |  |  | AJUSTE GRUESO | | | |
| 2 | PERIODO CANAL B | AJUSTE FINO | | | | | | | |
| 3 |  |  |  |  | AJUSTE GRUESO | | | |
| 4 | PERIODO CANAL C | AJUSTE FINO | | | | | | | |
| 5 |  |  |  |  | AJUSTE GRUESO | | | |
| 6 | PERIODO RUIDO |  |  |  | 5 BITS AJUSTE (0~31) | | | | |
| 7 | ACTIVACIÓN | E/S | | RUIDO | | | TONO | | |
| IOB | IOA | C | B | A | C | B | A |
| 8 | VOLUMEN A |  |  |  | M | 4 BITS VOLUMEN (0~15) | | | |
| 9 | VOLUMEN B |  |  |  | M | 4 BITS VOLUMEN (0~15) | | | |
| 10 | VOLUMEN C |  |  |  | M | 4 BITS VOLUMEN (0~15) | | | |
| 11 | PERIODO  ENVOLVENTE | 8 BITS AJUSTE FINO | | | | | | | |
| 12 | 8 BITS AJUSTE GRUESO | | | | | | | |
| 13 | FORMA ENVOLVENTE |  |  |  |  | CONT | ATT | ALT | HOLD |
| 14 | PUERTO A JOYSTICK | CAS | KDB | B | A | → | ← | ↓ | ↑ |
| 15 | PUERTO B JOYSTICK | INP | MODE | B | A | → | ← | ↓ | ↑ |

Frecuencia = ( 1789772.5 Hz ) / 16 \* ( 256 \* Grueso + Fino )

X = INT( 1789772.5 / ( 16 \* Frecuencia )

Fino = INT( X / 256 )

Grueso = X – Fino

HOLD: Sostenimiento o mantenimiento, determina si la envolvente mantiene o no, el valor final del primer ciclo.

0 - La envolvente se repetirá periódicamente.

1 - El volumen del sonido se sostendrá.

ALT: Alternativamente, si está a uno se sucederán alternativamente rampas ascendentes y descendentes que determinarán una envolvente triangular.

ATT: Es ataque es el periodo inicial de la producción del sonido

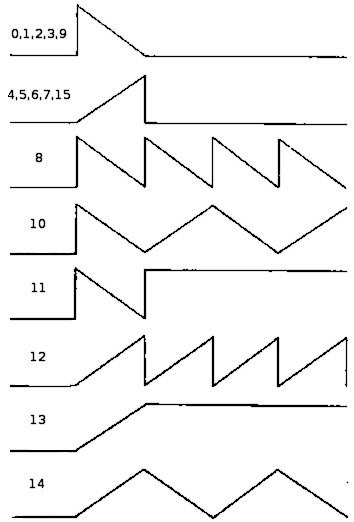
0 – Ascendente

1 - Descendente

CONT: La envolvente se repite periódicamente o no después del primer ciclo.

0 – Al terminar el primer ciclo el volumen caerá a cero.

1 – La envolvente se repite.



**ESCRIBIR**

**LD** A, n ; n. registro (0~15)

**OUT** [$A0], A ; al PSG

**LD** A, v ; valor registro (0~255)

**OUT** [$A1], A ;

**LEER**

**LD** A, n ; n. registro (0~15)

**OUT** [$A0], A ; al PSG

**IN** A,[$A2] ; lee el valor devuelto

**LD** [v],A ; guardamos el valor

**57 VDP POR PUERTOS** [**Índice**](#INDICE)

**Puerto de Datos (I/O $98) “Data Port”**

* El Puerto de Datos se utiliza para leer o escribir bytes individuales a la VRAM.
* El VDP posee un Registro de Dirección interno que apunta a una localización en la VRAM.
* Leyendo o escribiendo el Puerto de Datos accedemos al byte de la VRAM indicado por el Registro de Dirección.
* Después de una lectura o escritura el Registro de Direcciones se incrementa automáticamente para que apunte a la siguiente posición de VRAM.
* Bytes secuenciales pueden ser accedidos simplemente leyendo o escribiendo continuamente el Puerto de Datos.

**Puerto de Comandos (I/O $99) “Command Port”**

El Puerto de Comandos se utiliza para tres propósitos:

1. Para configurar el Registro de Direcciones del Puerto de Datos.
2. Para leer el Registro de Estado del VDP

**IN** A, [$99] ;

1. Para escribir en uno de los Registros de Modo del VDP

**El Registro de Direcciones:**

* El Registro de Dirección del Puerto de Datos debe estar configurado de diferentes maneras dependiendo de si el acceso posterior va a ser una lectura o una escritura.
* El Registro de Dirección puede ajustarse a cualquier valor de 0000H a 3FFFH escribiendo primero el LSB (byte menos significativo) y luego el MSB (byte más significativo) en el Puerto de Comandos.
* Los bits 6 y 7 del MSB son utilizados por el VDP para determinar si el Registro de Dirección está configurado para lectura o escritura de la siguiente manera:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| LECTURA | LSB  $00-$FF | | | | | | | | 0 | 0 | MSB  $00-$3F | | | | | |
| ESCRITURA | 0 | 1 |

Es importante que no se produzcan otros accesos al VDP entre la escritura del LSB y del MSB, ya que esto alteraría su sincronización. El Gestor de Interrupciones de la ROM BIOS está leyendo continuamente el estado del Registro de Estado como una tarea en segundo plano por lo que las interrupciones se deben desactivar y activar cuando sea necesario. Para esto contamos con las instrucciones **DI** y **EI** respectivamente.

Escribir: Leer:

**IN** A, [$99] **IN** A, [$99] ; Registro de Estado (para sincronizar)

**LD** A, LSB **LD** A, LSB ; Dirección LSB de la VRAM

**OUT** [$99], A **OUT** [$99], A ; Puerto de Comandos

**LD** A, MSB + 64 **LD** A, MSB ; Dirección MSB de la VRAM

**OUT** [$99], A **OUT** [$99], A ; Puerto de Comandos

**LD** A, DATO

**OUT** [$98], A **IN** A, [$98] ; Puerto de Datos

**El Registro de Estado:**

Leyendo el Puerto de Comandos tenemos acceso al Registro de Estado que contiene la siguiente información: **IN** A, [$99]

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| F  Frame Int. | 5S  5º Sprite | C  Colisión | Número del 5º Sprite (0 a 31) | | | | |

* La bandera de (F) Frame está normalmente a 0, pero se pone a 1 al llegar al final de la última línea activa de video. Si la frecuencia es de 50Hz y esto ocurre cada 20ms, y si es de 60Hz cada 16,666ms. Leyendo el Registro de Estado esta bandera se pone a 0.
* La bandera de (5S) 5º Sprite se activa a 1 cuando hay más de cuatro Sprites en la misma línea de video. Leyendo el Registro de Estado esta bandera se pone a 0.
* La bandera de (C) Colisión se activa cuando los Sprites tienen un o más pixels superpuestos, si los Sprites en movimiento son tan rápidos que pasan por encima uno del otro entre los Frames no se detectará ninguna colisión.
* Los bits del 0 al 4 contienen el número del Sprite que ha desencadenado la bandera de 5º Sprite.
* Leyendo el Registro de Estado las banderas se ponen a 0.

**Registros de Modo:**

El VDP tiene ocho Registros de Modo que son de sólo escritura numerados del 0 al 7. Un Registro en particular se escribe mandando un byte de datos en el Puerto de Datos y a continuación un byte de selección de Registro al Puerto de Comandos. El byte de selección de Registro contiene el número de registro en los tres bits más bajos y su bit más alto debe ser 1.

**LD** A, DATO ; El DATO a escribir en la Registro

**OUT** [$98], A ; Mandamos al Puerto de Datos

**LD** A, núm\_reg + 128 ; Número de Registro de 0 a 7 + 128

**OUT** [$99], A ; Seleccionamos un registro

**Registro de Modo 0:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | M3 | EV |

* El bit de señal de video Externa VDP determina si la entrada del VDP externa debe ser activada o desactivada:
  + 0 = Desactivado,
  + 1 = Habilitado.
* El bit M3 se usa en unión con el M1 y el M2 del Registro de Modo 1.

**Registro de Modo 1:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 4/16K | Blank | IE | M1 | M2 | 0 | Size | Mag |

* La bandera de (Mag) Magnificación duplica el tamaño de los Sprites:
  + 0=Normal
  + 1=Doble
* La bandera (Size) de Tamaño determina el tamaño de los Sprites:
  + 0=8x8 bits
  + 1=16x16 bits
* Las banderas M1, M2 y M3 determinan el modo de video:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| M1 | M2 | M3 | Nº | DESCRIPCION | PATRONES  MODELOS | NOMBRES  CARACTER | TABLA  COLOR | SPRITES  PATRONES | SPRITES  ATRIBUTOS |
| 1 | 0 | 0 | 0 | Texto 40x24 | 256x8=  2048 bytes | 40x24=  960 bytes | 2 colores | NO | NO |
| $0800  $0FFF | $0000  $03BF | Registro Modo 7 |
| 0 | 0 | 0 | 1 | Texto 32x24 | 256x8=  2048 bytes | 32x24=  768 bytes | 16 colores  (4x3)x3=  32 bytes | 2028 bytes | 128 bytes |
| $0000  $07FF | $1800  $1AFF | $2000  $201F | $3800  $3FFF | $1B00  $1B7F |
| 0 | 0 | 1 | 2 | Gráficos 32x24  (256x192 pixels) | 768x8=  6144 bytes | 32x24=  768 bytes | 16 colores  768x8=  6144 bytes | 2028 bytes | 128 bytes |
| $0000  $17FF | $1800  $1AFF | $2000  $37FF | $3800  $3FFF | $1B00  $1B7F |
| 0 | 1 | 0 | 3 | Modo Multicolor  (64x24 pixels) | 1563 bytes | 768 bytes | 16 colores | 2028 bytes | 128 bytes |
| $0000  $05FF | $0800  $0AFF | Incluida en Patrones | $3800  $3FFF | $1B00  $1B7F |
| Color del Borde: Registro de Modo 7 (azul) | | | | | | | | | |

* La bandera (IE) Activa o Desactiva la Interrupción del VDP cuando llega a la última línea dibujable, el VDP saltará automáticamente a la dirección de memoria ROM $38, si está activada la BIOS esta permite un gancho en la dirección de memoria $FD9A a $FD9E donde puedes poner CALL mi\_rutina; RET; RET; y a funcionar:
  + 0=Desactiva
  + 1=Activa
* La bandera (Blank) Activa o Desactiva el video en pantalla, cuando esta desactivado se muestra el color del borde, no le veo mucha utilidad asi que déjalo siempre a 1:
  + 0=Desactiva
  + 1=Activa
* La bandera (4/16K) selecciona el tipo de chip de memoria del VDP es mejor que lo dejes siempre a 1:
  + 0=4K
  + 1=16K

**Registro de Modo 2:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 0 | Name Table Base | | | |

Este registro tiene un valor de 0 a 15, que al multiplicarse por $400, da la dirección de comienzo en la VRAM de la Tabla de Nombres.

**Registro de Modo 3:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Colour Table Base | | | | | | | |

Este registro tiene un valor entre 0 y 255, que al multiplicarse por $40 nos da la dirección de comienzo en la VRAM de la Tabla de Colores.

**Registro de Modo 4:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | Character Pattern | | |

Este registro tiene un valor entre 0 y 7, que al multiplicarse por $800 nos da la dirección de comienzo en la VRAM de la Tabla de Patrones.

Los registros 3 y 4 en el modo gráfico de alta resolución 256x192 se comportan diferente, ver tabla anexa:

**Tablas Registro 3 TABLA COLOR Registro 4 TABLA PATRONES**

1 $1F - $0000, $0000, $0000 $00 - $0000, $0000, $0000

2 $3F - $0000, $0800, $0000 $01 - $0000, $0800, $0000

2 $5F - $0000, $0000, $1000 $02 - $0000, $0000, $1000

3 $7F - $0000, $0800, $1000 $03 - $0000, $0800, $1000

1 $9F - $2000, $2000, $2000 $04 - $2000, $2000, $2000

2 $BF - $2000, $2800, $2000 $05 - $2000, $2800, $2000

2 $DF - $2000, $2000, $3000 $06 - $2000, $2000, $3000

3 $FF - $2000, $2800, $3000 $07 - $2000, $2800, $3000

**Registro de Modo 5:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | Sprite Attribute Base | | | | | | |

Este registro tiene un valor entre 0 y 127, que al multiplicarse por $80 nos da la dirección de comienzo en la VRAM de la Tabla de Atributos de Sprites.

**Registro de Modo 6:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | Sprite Pattern | | |

Este registro tiene un valor entre 0 y 7, que al multiplicarse por $800 nos da la dirección de comienzo en la VRAM de la Tabla de Patrones de Sprites.

**Registro de Modo 7:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Color del Texto u Tinta  En Modo 0 bits a 1 | | | | Color del Borde  En Modo 0 bits a 0 | | | |

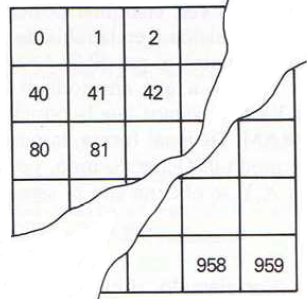
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Num.** | **Nombre** | **R** | **G** | **B** |
| 0 | Transparente | 0 | 0 | 0 |
| 1 | Negro | 0 | 0 | 0 |
| 2 | Verde | 33 | 200 | 66 |
| 3 | Verde Claro | 94 | 220 | 120 |
| 4 | Azul Oscuro | 84 | 85 | 237 |
| 5 | Azul Claro | 125 | 118 | 252 |
| 6 | Rojo Oscuro | 212 | 82 | 77 |
| 7 | Azul Cielo / Ciano | 66 | 235 | 245 |
| 8 | Rojo | 252 | 85 | 84 |
| 9 | Rojo Brillante | 255 | 121 | 120 |
| 10 | Amarillo | 212 | 193 | 84 |
| 11 | Amarillo Claro | 230 | 206 | 128 |
| 12 | Verdo Oscuro | 33 | 179 | 59 |
| 13 | Púrpura / Magenta | 201 | 91 | 186 |
| 14 | Gris | 204 | 204 | 204 |
| 15 | Blanco | 255 | 255 | 255 |

**\* SCREEN 0 TEXT / 2 Colores / (40x24 chars) / ASCII**

0000H - 03BFH --> Nombres 40 x 24 = 960 bytes

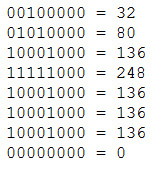
0800H - 0FFFH --> Patrones 256 \* 8 = 2048 bytes

Los colores de los bits 0 y 1 en este modo son definidos por el Registro de Modo 7.



La tabla de nombres corresponde con los códigos ASCII

VRAM = Y \* 40 + X

La Tabla de Patrones 256 caracteres \* 8 bytes = 2048 bytes.

VRAM = (ASCII \* 8) + 2048

Esta tabla está en la ROM a partir de la dirección $1BBF

\* SCREEN 0 (WIDTH 80) / TEXT

0000H - 077FH --> Nombres

0800H - 08EFH --> Blink table (24 lines mode)

- 090DH (26.5 lines mode)

1000H - 17FFH --> Patrones

**\* SCREEN 1 GRAPHIC / 16 colores / (32x24 chars) / ASCII**

0000H - 07FFH --> Patrones 256 chars \* 8 bytes = 2048

1800H - 1AFFH --> Nombres 32 x 24 = 768 bytes

2000H - 201FH --> Color 256 / 8 = 32 bytes

El color del borde exterior de la pantalla en este modo se define por los 4 bits bajos del Registro de Modo 7.

1B00H - 1B7FH --> Sprite Atributos

3800H - 3FFFH --> Sprite Patrones

**\* SCREEN 2 GRAPHIC / 16 colores / (32x24 chars) / (256x192 pixels) / 3 Bloques**

0000H - 07FFH --> Patrones 1 2048 bytes \* 3 = 6144 bytes

0800H - 0FFFH --> Patrones 2 256 chars \* 8 bytes = 2048 bytes

1000H - 17FFH --> Patrones 3

1800H - 18FFH --> Nombres 1 256 bytes \* 3 = 768 bytes

1900H - 19FFH --> Nombres 2 256 indices

1A00H - 1AFFH --> Nombres 3

2000H - 27FFH --> Color 1 2048 bytes \* 3 = 6144 bytes

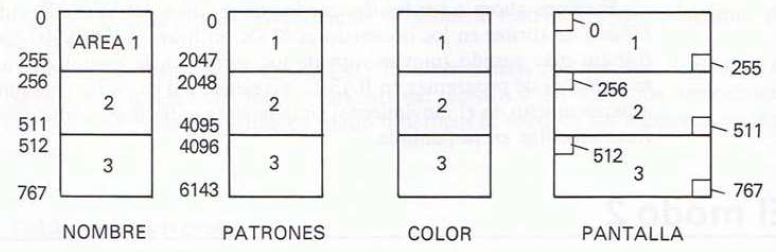
2800H - 2FFFH --> Color 2 256 chars \* 8 bytes = 2048 bytes

3000H - 37FFH --> Color 3 4 bits tinta, 4 bits fondo.

El color del borde exterior de la pantalla en este modo se define por los 4 bits bajos del Registro de Modo 7.

1B00H - 1B7FH --> Sprite Atributos 128 bytes

3800H - 3FFFH --> Sprite Patrones 2048 bytes



**\* SCREEN 3 MULTI COLOUR / 16 colores / (32x24 chars) / (64x24 pixels)**

0000H - 07FFH --> Patrones 768 \* 2 bytes = 1536 bytes incluye los colores

0800H - 0AFFH --> Nombres 32 \* 24 = 768 bytes

El color del borde exterior de la pantalla en este modo se define por los 4 bits bajos del Registro de Modo 7.

1B00H - 1B7FH --> Sprite Atributos 128 bytes

3800H - 3FFFH --> Sprite Patrones 2048 bytes

**Sprites**

Menos en el modo 0, el VDP puede manejar 32 sprites en todos los modos y el funcionamiento es idéntico.

La Tabla de Atributos ocupa 128 bytes de VRAM, contiene 32 bloques de 4 bytes para cada sprite. El primer bloque es en número cero y es el sprite más al frente. El 31 es el ultimo sprite y el más al fondo.

El fotmato de cada bloque es el siguiente:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bits | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Byte 0 | Posición Vertical de -32 ~ 191 | | | | | | | |
| Byte 1 | Posición Horizontal 0 ~ 255 | | | | | | | |
| Byte 2 | Número de Patrón | | | | | | | |
| Byte 0 | EC | 0 | 0 | 0 | Código de Color | | | |

La posicion se cuenta desde el tope izquierdo de la pantalla.

Y = 208 Hace que todos los sprites de menor prioridad desaparezcan de la pantalla.

Y = 209 Desaparece el sprite.

EC (Early Clock Bit) si se pone a 1 desplaza el sprite 32 unidades a la izquierda.

La Tabla de Patrones de Sprite contiene 256 bloques de 8x8 pixels o 64 bloques de 16x16 pixels. No se pueden mezclar estos dos tipos.

* Registro de Modo 1 bit 0: Magnificación duplica el tamaño de los Sprites:
  + 0=Normal
  + 1=Doble
* Registro de Modo 1 bit 1: Tamaño determina el tamaño de los Sprites:
  + 0=8x8 bits
  + 1=16x16 bits

|  |  |  |  |
| --- | --- | --- | --- |
| 8 Bytes |  | 16 Bytes | |
| Block A | A | C |
| Block B |
| Block C | B | D |
| Block D |

**58 CRÉDITOS** [**Índice**](#INDICE)

