Ensamblador para ZX Spectrum

PONG

Ensamblador para ZX Spectrum PONG por Juan Antonio Rubio García

Esta obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional License.

Comentarios al código por Spirax.

Correcciones al texto original realizadas por Joaquín Ferrero.

Contenido

Introducción....4
Herramientas que vamos a usar....5
Editor de texto....5
Emulador de ZX Spectrum....5
Compilador de ensamblador....5
Control de código fuente....5
Enlaces de interés....5
Hola Mundo....6
¿Qué es el Z80?....6
Registros del Z80....6
Memoria del ZX Spectrum....8
Decimal, binario, hexadecimal....9
Etiquetas, variables y constantes....10
ORG y END....10
Instrucciones de carga....11
Instrucciones RST....12
Incrementos y decrementos....12
Operaciones lógicas....14
Cambios de flujo de programa....14
Subrutinas....16
Puertos de entrada y salida....18
Paso 1: dibujando por la pantalla....22
Paso 2: teclas de control....30
Paso 3: palas y línea central....37
Cambiar el color del borde....37
Asignar los atributos de color a la pantalla....38
Dibujar la línea central del campo....41
Dibujar las palas de ambos jugadores....43
Mover las palas hacia arriba y hacia abajo....46
Paso 4: empezamos a mover la bola....55
Paso 5: movemos la bola por la pantalla....65
Paso 6: campo, palas, bola y temporización....75
Paso 7: detección de colisiones....90

Paso 8: partida a dos jugadores y cambio de velocidad de la bola....................100
Paso 9: cambio de dirección/velocidad de la bola al golpear la pala....................123
Paso 10: sonido y optimización....................136
Optimización de ScanKeys....................145
Optimización de Cls....................146
Optimización de MoveBall....................147
Optimización de ReprintLine....................147
Optimización de GetPointSprite....................149
Optimización PrintPoints y ReprintPoints....................151
Bug del golpeo de la bola en la parte baja de la pala....................155
Sonido....................156
Paso 11: optimización parte 2....................165
Optimización de PlaySound....................165
Optimización de ReprintPoints....................166
Paso 12: pantalla de carga....................171
Implementamos nuestro cargador....................171
Añadimos la pantalla de carga....................173
Incluimos nuestro PorompomPong....................174
Apéndice 1: frecuencias y notas....................175
Bibliografía....................179

Introducción

El objetivo del presente tutorial es adquirir las nociones básicas que nos permitirán, más adelante, realizar nuestros propios desarrollos en ensamblador de Z80, para ZX Spectrum.

Para hacer más ameno el proceso, vamos a desarrollar paso a paso una versión de uno de los videojuegos más famosos de todos los tiempos: Pong.

Nuestra versión va a ocupar poco más de 1.5 Kb y es compatible con los modelos de 16, 48 y 128 Kb de ZX Spectrum.

A cada paso veremos algo nuevo, y el resultado será algo funcional, haremos cosas que luego cambiaremos, hasta concluir el desarrollo.

No se pretende hacer un código optimizado, más bien mostrar paso a paso cómo hacer las cosas de distintas maneras.

Herramientas que vamos a usar

A continuación, se detallan las herramientas que vamos a usar para nuestro desarrollo.

Editor de texto

Cualquier editor de texto vale, por sencillo que sea, como es el caso del Bloc de notas de Windows.

Otros editores gratuitos, más potentes y con resaltado de sintaxis son:

- NotePad++
- Visual Studio Code, instalando la extensión Z80 Assembly (imanolea.z80-asm)
- Sublime Text, instalando el paquete z80asm-ti

Emulador de ZX Spectrum

Son muchos los emuladores de ZX Spectrum, siendo quizá los gratuitos los mejores.

Para el tutorial vamos a utilizar ZEsarUX, un emulador de desarrollo español y disponible para Windows, Mac y Linux.

Compilador de ensamblador

Para este tutorial vamos a utilizar PASMO, que es un ensamblador cruzado con versiones para Windows, Mac y Linux, y que genera código objeto ejecutable, para entre otros, el ZX Spectrum.

PASMO funciona por línea de comandos por lo que, si usas Windows, es recomendable incluirlo en la variable Path para que se pueda ejecutar desde cualquier directorio.

Control de código fuente

Es una buena práctica tener algún tipo de control de código fuente, para que en caso de que algo deje de funcionar poder ver como estaba en una versión anterior.

En nuestro caso he optado por Git, creando un repositorio local e instalando en Visual Studio Code la extensión Git Graph (mhutchie.git-graph).

Este no es un requisito obligatorio, aunque es muy aconsejable.

Enlaces de interés

- Notepad++
- Visual Studio Code
- Sublime Text
- ZEsarUX
- Pasmo
- Git
- Curso de ensamblador Z80 de Compiler Software

Hola Mundo

Antes de entrar de lleno con el desarrollo de PorompomPong, vamos a hacer lo que se hace casi cada vez que se inicia el aprendizaje de un lenguaje de programación, vamos a implementar un “Hola Mundo”.

La implementación de nuestro “Hola Mundo” nos va a servir para adquirir los conocimientos necesarios, para el posterior desarrollo de nuestro PorompomPong.

Con “Hola Mundo” vamos a descubrir:

- Características del microprocesador Zilog Z80 y de sus registros.
- La distribución de la memoria del ZX Spectrum.
- Números en distintas notaciones.
- Etiquetas, variables y constantes en ensamblador.
- Directivas ORG y END.
- Instrucciones de carga.
- Instrucciones RST.
- Incrementos y decrementos.
- Operaciones lógicas.
- Cambios de flujo de programa.
- Subrutinas.
- Puertos de entrada y salida.

¿Qué es el Z80?

El Z80 es un microprocesador que salió al mercado en 1976 de la mano de Zilog. Es el microprocesador que lleva el ZX Spectrum, en todos sus modelos.

El Z80 es una CPU de tipo “Little Endian”. Una CPU de este tipo, cuando almacena en memoria valores de 16 bits, almacena en la primera posición el byte menos significativo, y en la siguiente el más significativo: al cargar el valor $CCFF en la posición $8000, almacena en la posición $8000 el valor $FF y en la $8001 el valor $CC.

Otra característica del Z80 es que no es un microprocesador ortogonal, lo que hace que no todas las operaciones entre registros estén permitidas.

Registros del Z80

Los registros son memoria de alta velocidad y baja capacidad, y están integrados en el microprocesador.

El Z80 dispone de registros de 8 y 16 bits.

Registros de 8 bits

- **A**: acumulador. Es el destino de las operaciones aritméticas, lógicas y de comparación de 8 bits. Es el byte más significativo del registro de 16 bits AF.
- **F**: flags (indicadores). Conjunto de indicadores que dan información de las operaciones que se están realizando. Es el byte menos significativo del registro de 16 bits AF.
- **B**: registro de propósito general que se suele usar en bucles; la instrucción DJNZ lo usa como contador. Es el byte más significativo del registro de 16 bits BC.

- **C**: registro de propósito general. Es el byte menos significativo del registro de 16 bits BC.
- **D**: registro de propósito general. Es el byte más significativo del registro de 16 bits DE.
- **E**: registro de propósito general. Es el byte menos significativo del registro de 16 bits DE.
- **H**: registro de propósito general. Es el byte más significativo del registro de 16 bits HL.
- **L**: registro de propósito general. Es el byte menos significativo del registro de 16 bits HL.
- **I**: registro de interrupción. Permite manejar 128 interrupciones distintas.
- **R**: registro de refresco de memoria. Manejado por el Z80, cambia los bits del 0 al 6. Se puede usar para generar números pseudo aleatorios entre 0 y 127.

Registros alternativos

Los registros alternativos sirven para hacer una copia temporal de los registros de 8 bits:

- **A':** registro alternativo de A.
- **F':** registro alternativo de F.
- **B':** registro alternativo de B.
- **C':** registro alternativo de C.
- **D':** registro alternativo de D.
- **E':** registro alternativo de E.
- **H':** registro alternativo de H.
- **L':** registro alternativo de L.

Registros de 16 bits

- **AF:** formado por el registro A como byte más significativo y el F como byte menos significativo.
- **BC:** formado por el registro B como byte más significativo y el C como byte menos significativo. Se usa como contador en operaciones como LDIR, LDDR, etcétera.
- **DE:** formado por el registro D como byte más significativo y el E como byte menos significativo. Se usa, generalmente, para leer y escribir en una operación única, así como registro de destino en operaciones LDIR, LDDR, etcétera.
- **HL:** formado por el registro H como byte más significativo y el L como byte menos significativo. Se usa, generalmente, para leer y escribir en una operación única, así como registro de origen en operaciones como LDIR, LDDR, etcétera. El registro HL es el registro acumulador en operaciones de 16 bits.
- **IX:** acceso a memoria de forma indexada, LD (IX + desplazamiento), pudiendo ser el desplazamiento un valor entre -128 y 127.
- **IY:** acceso a memoria de forma indexada, LD (IY + desplazamiento), pudiendo ser el desplazamiento un valor entre -128 y 127.
- **SP:** puntero de pila. Apunta a la posición actual de la cabeza de la pila.
- **PC:** contador de programa. Contiene la dirección de la instrucción actual a ejecutar.

Códigos de operación de los registros (opcodes)

- **0:** B
- **1:** C
- **2:** D
- **3:** E

- **4:** H
- **5:** L
- **6:** (HL)
- **7:** A

Estos códigos de operación se utilizan para calcular el código de operación de las instrucciones en las que el parámetro es un registro.

- **LD A, r**: 0x78 + rb
- **LD C, r**: 0x48 + rb

Siendo rb el código de operación de los registros que se cargan, en este caso en A o en B.

Registro F

Cada bit del registro F, indicadores, tiene un significado propio que cambia automáticamente según el resultado de las operaciones que se realizan:

- **Bit 0:** flag C (acarreo). Se pone a 1 si el resultado de la operación anterior necesita un bit extra para representarse (me llevo una). Ese bit, flag de acarreo, es el bit extra que se necesita.
- **Bit 1:** flag N (resta). Se pone a 1 si la última operación fue una resta.
- **Bit 2:** flag P/V (paridad/desbordamiento). En operaciones que modifican el bit de paridad, se pone a 1 cuando el número de bits a 1 del resultado es par. En operaciones que modifican el bit de desbordamiento, se pone a 1 cuando el resultado de la operación necesita más de 8 bits para representarse.
- **Bit 3:** no se usa.
- **Bit 4:** flag H (acarreo BCD). Se pone a 1 cuando en operaciones BCD existe un acarreo del bit 3 al 4.
- **Bit 5:** no se usa.
- **Bit 6:** flag Z (cero). Se pone a 1 si el resultado de la operación anterior es 0. Muy útil en bucles.
- **Bit 7:** flag S (signo). Se pone a 1 si el resultado de la operación en complemento a dos es negativo.

No se puede acceder directamente al registro F, y no todas las operaciones le afectan.

Registro F – indicadores de flags								
Bit	7	6	5	4	3	2	1	0
	S	Z	F5	H	F3	P/V	N	C

Memoria del ZX Spectrum

La memoria está divida en dos, en los modelos 16K, o cuatro, en los modelos 48K, bloques de 16 KiB cada uno (16384 bytes):

- **Primer bloque**: de la posición $0000 a la $3FFF (0 a 16383). Se corresponde con la ROM y es de solo lectura.
- **Segundo bloque**: de la posición $4000 a la $7FFF (16384 a 32767). En este bloque se encuentran el área de pantalla, el buffer de impresora, las variables de sistema, etcétera, dejando aproximadamente 9 KiB para los programas, en los modelos 16K.

Los siguientes bloques de memoria sólo se encuentran en los modelos 48K:

- **Tercer bloque**: de la posición $8000 a la $BFFF (32768 a 49151). Es memoria RAM de propósito general.
- **Cuarto bloque**: de la posición $C000 a la $FFFF (49152 a 65535). Es memoria RAM de propósito general.

La distribución del segundo bloque, muy por encima, es la siguiente:

- **$4000 - $57FF (16384 a 22527)**: área de los píxeles de la pantalla. La pantalla del ZX Spectrum tiene una resolución de 256*192 píxeles. Cada byte de este rango de memoria representa ocho píxeles (256*192/8 = 6144 bytes).
- **$5800 - $5AFF (22528 a 23295)**: área de los atributos de color de la pantalla. La resolución en este caso es de 32*24 caracteres. Cada byte especifica el color de una zona de 8*8 píxeles, definiendo en los bits del 0 al 2 el color de tinta (de 0 a 7), en los bits del 3 al 5 el color del fondo (de 0 a 7), en el bit 6 el brillo (de 0 a 1) y en el bit 7 el parpadeo (de 0 a 1). El área ocupa un total de 768 bytes (32*24).
- **$5B00 a $5BFF (23296 a 23551)**: búfer de impresora. 256 bytes que se pueden usar si no tenemos impresora, o si no la usa el programa.
- **$5C00 - $5CB5 (23552 a 23733)**: variables de sistema.
- **$7FFF**: puntero de la pila. Suele apuntar a esta dirección y decrece según se ponen cosas en ella.

Decimal, binario, hexadecimal

La representación decimal es en la que estamos acostumbrados a ver los números, en una secuencia de dígitos en los que cada uno puede tener un valor entre 0 y 9. Esta notación también se conoce como decimal o en base 10.

En informática es distinto ya que los ordenadores trabajan con dos valores: 0 y 1. Estos números se conocen como binarios o en base 2.

En ensamblador, la forma más común de representar los números es en base 16 (notación hexadecimal). En hexadecimal, cada dígito puede representar un valor del 0 al 15; a partir del 9 se usan letras:

- A = 10
- B = 11
- C = 12
- D = 13
- E = 14
- F = 15

Un dígito hexadecimal representa 4 bits, por lo que a simple vista sabemos de cuántos bits se compone, siendo lo normal hablar de múltiplos de 8 (8, 16, 32, 64…).

Sin una calculadora a mano, la conversión de números entre distintas bases puede llegar a ser muy tediosa. Resulta de gran ayuda saber el valor de cada bit; en el caso del Z80, números de 8 y 16 bits.

Vamos a usar la siguiente tabla, en la que se muestran los valores de cada bit, para guiarnos en las conversiones:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
32718	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1

Según se ve en esta tabla, solo tenemos que sumar para convertir números de una base a otra, como se puede observar en el ejemplo siguiente:

```
5FA0h ✉ 0101 1111 1010 0000 ✉ 32 + 128 + 256 + 512 + 1024 + 2048 + 4096 + 16384 = 24480
```

Como se puede ver, la conversión hexadecimal/binario es directa, de cuatro en cuatro bits.

```
F0h ✉ 1111 0000         3Ah ✉ 0011 1010         CCh ✉ 1100 1100         78h ✉ 0111 1000

0001 0000 ✉ 10h         0100 0101 ✉ 45h         1010 1010 ✉ AAh         0010 0011 ✉ 23h
```

Etiquetas, variables y constantes

Las etiquetas nos permiten hacer referencia a posiciones de memoria a través de ellas, en lugar de tener que calcular y memorizar las direcciones. El programa ensamblador se encarga de sustituir las etiquetas por las direcciones de memoria correctas; este proceso lo realiza al crear el código objeto.

Si no pudiéramos utilizar etiquetas, al modificar alguna parte del código habría que recalcular las direcciones de memoria para todos los JR, JP o CALL. El ensamblador sustituye las etiquetas por las direcciones de memoria de las instrucciones que siguen a las mismas.

Las etiquetas sirven para definir rutinas y datos; en el caso de los datos pueden ser numéricos o texto, y constantes o variables.

Los datos se definen usando las siguientes directivas:

- **EQU**: define constantes `nombre EQU valor`
- **DB/DEFB**: define bytes `nombre DB 1, $FF, %10101010`
- **DM/DEFM**: define message `nombre DEFM "Hola Mundo"`
- **DW/DEFW**: define word `nombre DW $0040`
- **DS/DEFS**: define space `nombre DEFS $08`

DB, DEFB, DM, DEFM, DW, DEFW, DS o DEFS no se ensamblan, por lo que es recomendable ponerlas al final del código, ya que se ejecutarán como si fueran instrucciones del Z80. Si el código empezara con:

```
DB $CD, $00, $00
```

Al no ensamblarse la directiva DB, esta línea haría un reset; DB $CD, $00, $00 es CALL $0000.

ORG y END

ORG y END son dos de las directivas más importantes de las que vamos a usar. Con ORG especificamos la dirección de memoria donde cargar el código, pudiéndose poner varios ORG para cargar partes del código en distintas direcciones de memoria.

END sirve para indicar dónde finaliza el programa, y una dirección de autoinicio para PASMO.

Con lo que hemos visto hasta ahora, podemos desarrollar nuestro primer programa; no olvidéis abrir el editor de texto para escribir estas líneas:

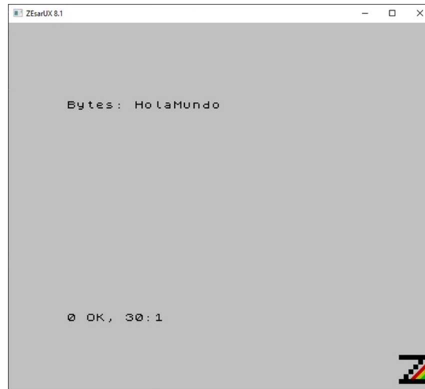
```
org   $8000
ret
end   $8000
```

Grabamos el archivo como “holamundo.asm” y compilamos con PASMO:

```
pasmo --name HolaMundo --tapbas holamundo.asm holamundo.tap --public
```

Este comando (pasmo…) lo vamos a usar siempre para compilar nuestros programas.

Ahora podemos abrir el archivo holamundo.tap con un emulador de ZX Spectrum y vemos que se ejecuta, aunque lo único que hace es salir, pero al menos no hemos roto nada.



Instrucciones de carga

Estas instrucciones se utilizan para cargar un valor en un registro, copiar el valor de un registro en otro, cargar un valor en memoria, cargar un registro en memoria y cargar un valor de memoria en un registro.

La sintaxis de las instrucciones de carga es la siguiente:

```
LD destino, origen
```

Destino puede ser un registro o una posición de memoria, mientras que el origen puede ser un registro, una posición de memoria o un valor de 8 o 16 bits.

Estas instrucciones no afectan al registro F, a excepción de **LD A, I** y **LD A, R**.

Este es el momento de volver a nuestro primer programa donde, justo debajo de ORG, vamos agregar las siguientes líneas:

```
ld hl, $4000

ld (hl), $ff
```

Con estas líneas activamos los 8 bits de la primera dirección de memoria de la pantalla, en adelante VideoRAM. Compilamos con PASMO y cargamos en el emulador:

```
pasmo --name HolaMundo --tapbas holamundo.asm holamundo.tap --public
```

Instrucciones RST

Estas instrucciones son utilizadas para saltar a una dirección concreta a través de una instrucción de un solo código de operación (opcode).

Existen varias instrucciones RST, aunque solo vamos a usar RST $10 (RST 16), que imprime el ASCII correspondiente al valor que tiene el registro A.

Recuperamos el archivo HolaMundo.asm, quitamos las dos líneas que habíamos añadido y escribimos las siguientes:

```
ld     a, 'H'
rst    $10
```

Compilamos y cargamos en el emulador. La letra H se debe imprimir en la pantalla.

Incrementos y decrementos

Sirven para incrementar (INC), o decrementar (DEC), en una unidad el contenido de determinados registros o posiciones de memoria (apuntadas por los registros HL, IX o IY).

Las operaciones permitidas son:

```
INC r                         DEC r
INC rr                        DEC rr
INC (HL)                      DEC (HL)
INC (IX + n)                  DEC (IX + n)
INC (IY + n)                  DEC (IY + n)
```

Estas operaciones, cuando se realizan sobre registros de 16 bits no afectan al registro F, mientras que, si se realizan sobre registros de 8 bits afectan de distintas maneras:

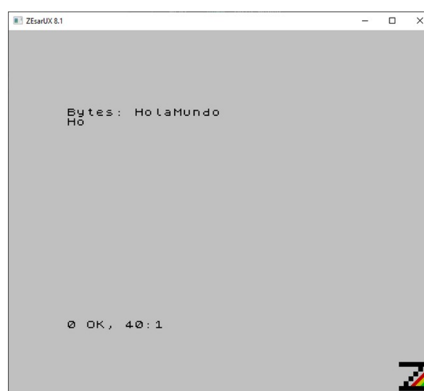
	Flags					
Instrucción	**S**	**Z**	**H**	**P**	**N**	**C**
INC r	*	*	*	V	0	-
INC (HL)	*	*	*	V	0	-
INC (ri + n)	*	*	*	V	0	-
INC rr	-	-	-	-	-	-
DEC r	*	*	*	V	1	-
DEC (HL)	*	*	*	V	1	-
DEC (ri + n)	*	*	*	V	1	-
DEC rr	-	-	-	-	-	-

- = no afecta, * = afecta, 0 = se pone a 0, 1 = se pone a 1, V = overflow

Recuperamos el archivo HolaMundo.asm, y lo dejamos tal y como sigue:

```
org   $8000        ; Dirección donde carga el programa

ld    hl, msg      ; Carga en HL la dirección de memoria del mensaje
ld    a, (hl)      ; Carga en A el primer carácter
rst   $10          ; Imprime el carácter
inc   hl           ; Apunta HL al carácter siguiente
ld    a, (hl)      ; Carga el carácter en A
rst   $10          ; Imprime el carácter

ret

msg:  defm 'Hola ensamblador ZX Spectrum'

end   $8000
```

Compilamos y cargamos en el emulador. Ahora veremos "Ho" impreso en pantalla.



Operaciones lógicas

Las operaciones lógicas se realizan a nivel de bit, comparando dos bits. Hay tres tipos de operaciones lógicas:

- **AND**: multiplicación lógica. El resultado solo es 1 si los dos bits están a 1.
- **OR**: suma lógica. Si alguno de los dos bits está a 1, el resultado es 1, de lo contrario el resultado es 0.
- **XOR**: or exclusivo. Si los dos bits son iguales, el resultado es 0, de lo contrario el resultado es 1.

En la siguiente tabla se muestran los posibles resultados de las operaciones lógicas:

Bit 1	Bit 2	AND	OR	XOR
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Resultados de las instrucciones lógicas

El formato de las operaciones lógicas es el siguiente:

```
AND   origen
OR    origen
XOR   origen
```

En las operaciones lógicas, el origen puede ser cualquiera de los registros de 8 bits (a excepción del F), un valor, una posición de memoria apuntada por (HL) o por los registros índice, (IX + n) o (IY + n). El destino siempre es el registro A; las operaciones lógicas se hacen sobre el valor que contiene el registro A, y el resultado se deja en este mismo registro.

Las operaciones lógicas afectan al registro A de la siguiente manera:

	Flags					
Instrucción	**S**	**Z**	**H**	**P**	**N**	**C**
AND s	*	*	*	P	0	0
OR s	*	*	*	P	0	0
XOR s	*	*	*	P	0	0

- = no afecta, * = afecta, 0 = se pone a 0, 1 = se pone a 1, P = paridad

Cambios de flujo de programa

Cambian el flujo del programa (salta), con o sin condiciones, de manera absoluta (JP) o relativa (JR). Estas instrucciones no afectan al registro F.

Los saltos absolutos pueden ser:

- **JP nn**: salta a la dirección de memoria **nn**, que puede ser una etiqueta (en los siguientes casos también).
- **JP (HL)**: salta a la dirección de memoria del valor que tiene HL; al valor de HL (16 bits), no al valor de la dirección apuntada por HL (8 bits).
- **JP (registro índice)**: salta a la dirección de memoria del valor que tiene IX o IY.
- **JP NZ, nn**: salta a la dirección **nn**, si el flag Z está a cero; el resultado de la última operación no es cero.

- **JP Z, nn**: salta a la dirección de memoria **nn** si el flag Z está a uno; el resultado de la última operación es cero.
- **JP NC, nn**: salta a la dirección de memoria **nn** si el flag C está a cero; no hay acarreo.
- **JP C, nn**: salta a la dirección de memoria **nn** si el flag C está a uno; hay acarreo.
- **JP PO, nn**: salta a la dirección de memoria **nn** si el flag P/V está a cero; no hay paridad/desbordamiento.
- **JP PE, nn**: salta a la dirección de memoria **nn** si el flag P/V está a uno; hay paridad/desbordamiento.
- **JP P, nn**: salta a la dirección de memoria **nn** si el flag S está a cero; el resultado de la última operación es positivo.
- **JP M, nn**: salta a la dirección de memoria **nn** si el flag S está a uno; el resultado de la última operación es negativo.

Los saltos relativos, son relativos a la instrucción actual y saltan un número de bytes que van desde -128 a 127. Las rutinas con saltos relativos son reubicables, pues no afecta la posición de memoria en la que se cargan. Los saltos relativos pueden ser:

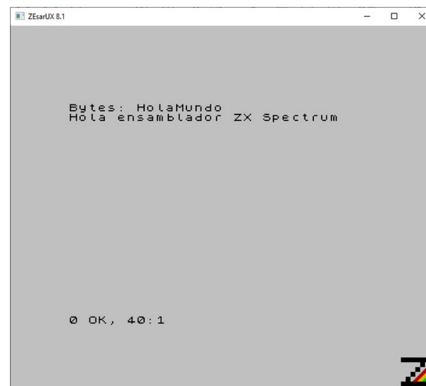
- **JR n**: salta a la dirección de memoria que está a **n** bytes; **n** puede ser una etiqueta (en los siguientes casos también).
- **JR NZ, n**: salta a la dirección de memoria que está a **n** bytes si el flag Z está a cero; el resultado de la última operación no es cero.
- **JP Z, n**: salta a la dirección de memoria que está a **n** bytes si el flag Z está a uno; el resultado de la última operación es cero.
- **JR NC, n**: salta a la dirección de memoria que está a **n** bytes si el flag C está a cero; no hay acarreo.
- **JR C, n**: salta a la dirección de memoria que está a **n** bytes si el flag C está a uno; hay acarreo.

Recuperamos el fichero HolaMundo.asm y vamos a utilizar las operaciones lógicas, y los cambios de flujo, para imprimir todo el mensaje:

```
org   $8000       ; Dirección donde se carga el programa

ld    hl, msg     ; Carga en HL la dirección de memoria del mensaje

Bucle:
ld    a, (hl)     ; Carga un carácter de la cadena
or    a           ; Comprueba si A es 0. A or A = 0 solo si A = 0
jr    z, Fin      ; Si A = 0, salta a la etiqueta Fin
rst   $10         ; Imprime el carácter
inc   hl          ; Apunta HL al siguiente carácter
jr    Bucle       ; Vuelve al principio del bucle

Fin:
ret               ; Sale del programa
```

```
msg:  defm 'Hola ensamblador ZX Spectrum', $00
            ; Cadena terminada en 0 = null

end   $8000
```

Compilamos con PASMO, cargamos en el emulador y vemos los resultados:



Subrutinas

Las subrutinas son bloques de código, que hacen una acción concreta, y al que se puede llamar en ocasiones múltiples; se usa CALL para saltar a una subrutina y RET para salir y volver al lugar desde el que se ha llamado.

CALL es parecido a JP, pero antes de saltar hace un PUSH de PC para guardar por dónde va el programa. Al hacer RET, se hace POP de PC y el programa vuelve por donde iba.

Se pueden realizar CALL y RET condicionales, al igual que se ha visto con JP y JR:

```
CALL  nn                                              RET
CALL  NZ, nn                                          RET   NZ
CALL  Z, nn                                           RET   Z
CALL  NC, nn                                          RET   NC
CALL  C, nn                                           RET   C
CALL  PO, nn                                          RET   PO
CALL  PE, nn                                          RET   PE
CALL  P, nn                                           RET   P
CALL  M, nn                                           RET   M
```

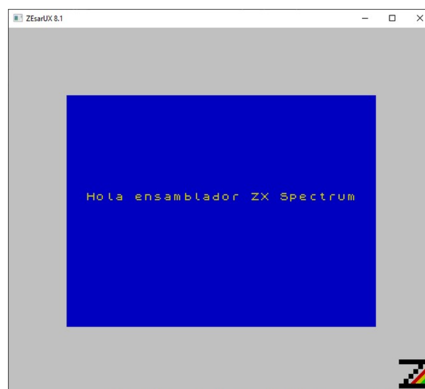
Recuperamos HolaMundo.asm, y gracias a CALL vamos a llamar a alguna rutina de la ROM, para hacer que los resultados sean algo más vistosos:

```
org   $8000       ; Dirección donde se carga el programa

; Variable de sistema donde están los atributos permanentes
; de la pantalla 1. La pantalla 1 es la principal.
```

```
; El formato es Flash, Bright, Paper, Ink (FBPPPIII).
ATTR_S:     equ   $5c8d

; Variable de sistema donde está el atributo actual (FBPPPIII).
ATTR_T:     equ   $5c8f

; -------------------------------------------------------------
; Rutina de la ROM similar al AT de Basic
; Posiciona el cursor en las coordenadas especificadas.
; Entrada:  B ✉ Coordenada Y.
;           C ✉ Coordenada X.
; Para esta rutina, la esquina superior izquierda de la pantalla
; es (24, 33).
; Altera el valor de los registros A, DE y HL.
; -------------------------------------------------------------
LOCATE:     equ   $0dd9

; -------------------------------------------------------------
; Rutina de la ROM semejante al CLS de Basic.
; Borra la pantalla usando los atributos cargados en la
; variable de sistema ATTR_S.
; Altera el valor de los registros AF, BC, DE y HL.
; -------------------------------------------------------------
CLS:  equ   $0daf

Inicio:
ld    a, $0e      ; Carga en A los atributos de color
ld    hl, ATTR_T  ; Carga en HL la dirección de memoria donde se
                  ; encuentran los atributos actuales
ld    (hl), a     ; Carga en memoria los atributos actuales
ld    hl, ATTR_S  ; Carga en HL la dirección de memoria donde
                  ; se encuentran los atributos permanentes
ld    (hl), a     ; Carga en memoria los atributos permanentes

call  CLS         ; Limpia la pantalla usando los atributos de ATTR_S
```

```
ld    b, $18-$0a   ; Carga la coordenada Y en B
ld    c, $21-$02   ; Carga la coordenada X en C
call  LOCATE       ; Posiciona el cursor

ld    hl, msg      ; Carga en HL la dirección de memoria del mensaje

Bucle:
ld    a, (hl)      ; Carga un carácter de la cadena
or    a            ; Comprueba si A es 0.
jr    z, Fin       ; Salta a la etiqueta fin si A = 0
rst   $10          ; Imprime el carácter
inc   hl           ; Apunta HL al siguiente carácter
jr    Bucle        ; Bucle hasta que A = 0

Fin:
jr    Fin          ; Bucle infinito

msg:  defm  'Hola ensamblador ZX Spectrum', $00
; Cadena terminada en 0 = null

end   $8000
```

Compilamos con PASMO, cargamos en el emulador y vemos los resultados:



Puertos de entrada y salida

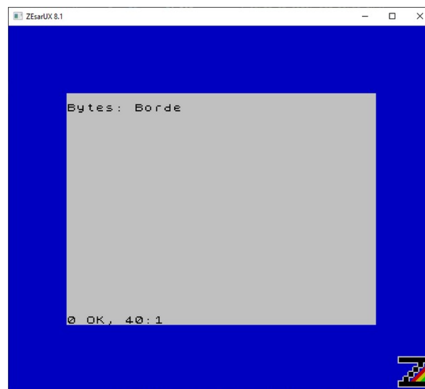
Los puertos de entrada y salida se usan, entre otras cosas, para leer el teclado, el joystick, etcétera.

En nuestro caso, por ahora, solo lo vamos a usar para cambiar el color del borde de la pantalla, usando la instrucción OUT y el puerto $FE.

Vamos a realizar un pequeño programa para ver cómo se cambia el borde:

```
org   $8000        ; Dirección donde se carga el programa
ld    a, $01       ; Carga el color del borde en A
out   ($fe), a     ; Cambia el color del borde
ret
end   $8000
```

Compilamos con PASMO, cargamos en el emulador y vemos el resultado:



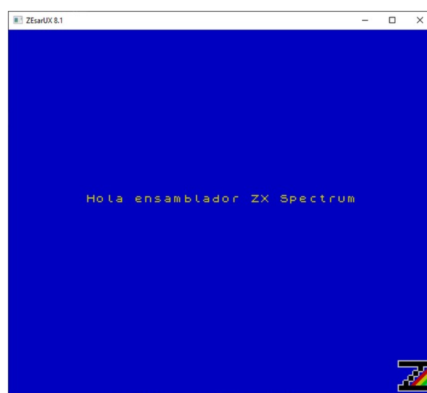
Con esto ya podemos finalizar nuestro primer programa en ensamblador para ZX Spectrum. Recuperamos el archivo HolaMundo.asm y añadimos las líneas para cambiar el color del borde, justo antes de la llamada a CLS:

```
org   $8000        ; Dirección donde se carga el programa

; Variable de sistema donde están los atributos permanentes
; de la pantalla 1. La pantalla 1 es la principal.
; El formato es Flash, Bright, Paper, Ink (FBPPPIII).
ATTR_S:      equ   $5c8d

; Variable de sistema donde está el atributo actual (FBPPPIII).
ATTR_T:      equ   $5c8f

; ------------------------------------------------------------
; Rutina de la ROM similar al AT de Basic
; Posiciona el cursor en las coordenadas especificadas.
; Entrada:   B ✉ Coordenada Y.
;            C ✉ Coordenada X.
; Para esta rutina, la esquina superior izquierda de la pantalla
; es (24, 33).
; Altera el valor de los registros A, DE y HL.
; ------------------------------------------------------------
```

```
LOCATE:     equ   $0dd9

; ---------------------------------------------------------------
; Rutina de la ROM semejante al CLS de Basic.
; Borra la pantalla usando los atributos cargados en la
; variable de sistema ATTR_S.
; Altera el valor de los registros AF, BC, DE y HL.
; ---------------------------------------------------------------
CLS:  equ   $0daf

Inicio:
ld    a, $0e      ; Carga en A los atributos de color
ld    hl, ATTR_T  ; Carga en HL la dirección de memoria donde se
                  ; encuentran los atributos actuales
ld    (hl), a     ; Carga en memoria los atributos actuales
ld    hl, ATTR_S  ; Carga en HL la dirección de memoria donde
                  ; se encuentran los atributos permanentes
ld    (hl), a     ; Carga en memoria los atributos permanentes

ld    a, $01      ; Carga en A el color del borde
out   ($fe), a    ; Cambia el color del borde

call  CLS         ; Limpia la pantalla usando los atributos de ATTR_S

ld    b, $18-$0a  ; Carga la coordenada Y en B
ld    c, $21-$02  ; Carga la coordenada X en C
call  LOCATE      ; Posiciona el cursor

ld    hl, msg     ; Carga en HL la dirección de memoria del mensaje

Bucle:
ld    a, (hl)     ; Carga un carácter de la cadena
or    a           ; Comprueba si A es 0.
jr    z, Fin      ; Salta a la etiqueta fin si A = 0
rst   $10         ; Imprime el carácter
inc   hl          ; Apunta HL al siguiente carácter
```

```
jr    Bucle       ; Bucle hasta que A = 0


Fin:
jr    Fin         ; Bucle infinito


msg:  defm  'Hola ensamblador ZX Spectrum', $00
; Cadena terminada en 0 = null


end   $8000
```

Compilamos con PASMO, cargamos en el emulador y vemos los resultados:



Ya hemos desarrollado nuestro primer programa en ensamblador para ZX Spectrum. A partir de aquí empezamos con el desarrollo de nuestro PorompomPong.

Paso 1: dibujando por la pantalla

La pantalla del ZX Spectrum está situada, el área de los píxeles, desde la dirección de memoria $4000 a la $57FF, ambas inclusive, lo que hace un total de 6144 bytes, o lo que es lo mismo 256x192 píxeles, 32 columnas y 24 líneas.

El ZX Spectrum divide la pantalla en tres tercios, de ocho líneas cada uno, con ocho scanlines (línea horizontal de la pantalla de un píxel de alto) por línea. Las direcciones de memoria que referencian a cada byte de la pantalla (área de píxeles), se codifican de la siguiente manera:

```
010T TSSS LLLC CCCC
```

Donde TT es el tercio (de 0 a 2), SSS es el scanline (de 0 a 7), LLL es la línea (de 0 a 7) y CCCCC es la columna (de 0 a 31).

En este primer paso vamos a aprender cómo dibujar por la pantalla, y vamos a ver dos rutinas que usaremos en nuestro Pong, y muy posiblemente en nuestros próximos desarrollos.

Lo primero que vamos a hacer es crear una carpeta llamada Pong, y dentro de la misma vamos a añadir otra carpeta a la que vamos a llamar Paso01. Dentro de esta última carpeta vamos a crear los archivos Main.asm y Video.asm.

Las dos rutinas que vamos a añadir al archivo Video.asm, NextScan y PreviousScan, han sido tomadas del Curso de ensamblador Z80 de Compiler Software de Santiago Romero, que podemos encontrar en El wiki de speccy.org, y calculan el scanline siguiente y anterior a una posición dada.

Ambas rutinas reciben en HL la posición de la VideoRAM desde la que se quiere calcular el siguiente o anterior scanline, y devuelve dicha posición en el mismo registro. También alteran el valor de AF.

Pasamos a ver la rutina NextScan:

```
NextScan:
inc   h
ld    a, h
and   $07
ret   nz
```

En la primera instrucción incrementamos el scanline, `INC H`, que se encuentra en los bits 0 a 2 de H. Acto seguido cargamos el valor de H en A, `LD A, H`, y nos quedamos solo con el valor de los bits del scanline, `AND $07`.

Si el valor de la operación anterior no es 0, el scanline tiene un valor entre 1 y 7, no es necesario hacer ningún cálculo más y salimos de la rutina, `RET NZ`.

Si el valor es 0, el scanline antes de incrementar H era 7:

```
0100 0111
```

Al sumarle uno, deja los bits del scanline a 0 e incrementa en 1 los bits del tercio:

```
0100 1000
```

Lo siguiente que hace la rutina es:

```
ld    a, l
add   a, $20
ld    l, a
ret   c
```

Cargamos en A el valor de L, `LD A, L`, que contiene la línea dentro del tercio y la columna. Le sumamos 1 a la línea, `ADD A, $20`:

```
$20   =     0010 0000   = LLLC CCCC
```

Luego cargamos el resultado en L, `LD L, A`, y si hay acarreo salimos, `RET C`.

Si hay acarreo, la línea antes de añadirle $20 era 7. Al añadirle 1, la línea pasa a 0 y hay que incrementar el tercio, que ya se incrementó al incrementar el scanline.

Por último, si seguimos adelante es porque seguimos dentro del mismo tercio, por lo que hay que decrementarlo para dejarlo como estaba. Al llegar a este punto, al incrementar el scanline hemos cambiado de línea, y al incrementar la línea no hemos cambiado de tercio.

```
ld    a, h
sub   $08
ld    h, a
ret
```

Cargamos el valor de H en A, `LD A, H`, tercio y scanline. A este valor le restamos $08 para decrementar en uno el tercio, `SUB $08`, y dejarlo como estaba:

```
$08   =     0000 1000   =     010T TSSS
```

Cargamos el resultado de la operación en H, `LD H, A`, y salimos de la rutina, `RET`.

El código completo de la rutina es:

```
; ------------------------------------------------------------------------------
; NextScan. https://wiki.speccy.org/cursos/ensamblador/gfx2_direccionamiento
; Obtiene la posición de memoria correspondiente al scanline siguiente al indicado.
; 010T TSSS LLLC CCCC
; Entrada:  HL -> scanline actual.
; Salida:   HL -> scanline siguiente.
; Altera el valor de los registros AF y HL.
; ------------------------------------------------------------------------------
NextScan:
inc     h                     ; Incrementa H para incrementar el scanline
ld      a, h                  ; Carga el valor en A
and     $07                   ; Se queda con los bits del scanline
ret     nz                    ; Si el valor no es 0, fin de la rutina
```

```
; Calcula la siguiente línea
ld      a, l                  ; Carga el valor en A
add     a, $20                ; Añade 1 a la línea (%0010 0000)
ld      l, a                  ; Carga el valor en L
ret     c                     ; Si hay acarreo, ha cambiado de tercio,
                              ; que ya viene ajustado de arriba. Fin de la rutina

; Si llega aquí, no ha cambiado de tercio y hay que ajustar
; ya que el primer inc h incrementó el tercio
ld      a, h                  ; Carga el valor en A
sub     $08                   ; Resta un tercio (%0000 1000)
ld      h, a                  ; Carga el valor en H
ret
```

En este punto vamos a editar el archivo Main.asm para probar la rutina NextScan.

El primer paso es indicar donde se va a cargar el programa, en nuestro caso en la dirección $8000 (32768):

```
org   $8000
```

Lo siguiente es apuntar HL a la dirección de memoria de la VideoRAM en donde vamos a empezar a dibujar, en nuestro caso en la esquina superior izquierda:

```
ld    hl, $4000
```

Si recordamos cómo se codifica una dirección de memoria de la VideoRAM:

```
010T TSSS LLLC CCCC
```

Y ponemos $4000 en binario:

```
0100 0000 0000 0000
```

Vemos que $4000 hace referencia al tercio 0, línea 0, scanline 0 y columna 0.

Vamos a pintar una columna vertical, desde arriba hacia abajo, que ocupe toda la pantalla, por lo que tenemos que hacer un bucle de 192 iteraciones, número de scanlines que tiene la pantalla, y vamos a cargar este valor en B:

```
ld    b, $c0
```

Una vez llegados a este punto, ya podemos hacer el bucle. Para ello vamos a poner una etiqueta para poder hacer referencia a ella. Cargamos el patrón 00111100 ($3C) en la dirección de la VideoRAM apuntada por HL, obtenemos la posición de memoria del scanline siguiente, y volvemos al principio del bucle hasta que B sea igual a 0:

```
loop:
ld    (hl), $3c
call  NextScan
djnz  loop
```

Como se puede ver en esta ocasión, HL va entre paréntesis, pero anteriormente cuando cargamos $4000 en HL no iba entre paréntesis. ¿Cuál es la diferencia?

Cuando escribimos `LD HL, $4000`, lo que hacemos es cargar $4000 en HL, es decir, HL = $4000. Por el contrario, al escribir `LD (HL), $3C,` lo que hacemos es cargar $3C en la posición de memoria apuntada por HL, es decir, ($4000) = $3C.

Después de cargar $3C en la posición de memoria apuntada por HL, obtenemos la dirección de memoria del siguiente scanline, lo que logramos llamando a la rutina NextScan, `CALL NextScan`.

La última instrucción, `DJNZ loop`, es el motivo de haber elegido el registro B para controlar las iteraciones del bucle. Si hubiéramos elegido otro registro de 8 bits, al llegar a este punto tendríamos que haberlo decrementado y luego comprobar que no ha llegado a 0, en cuyo caso saltaríamos a loop:

```
dec   a
jr    nz, loop
```

`DJNZ` hace todo esto, en una sola instrucción, usando el registro B, consumiendo 1 byte y 8 o 13 ciclos de reloj dependiendo de si no se cumple, o sí se cumple la condición. Usando `DEC` y `JR` se emplean 3 bytes y 11 o 17 ciclos de reloj.

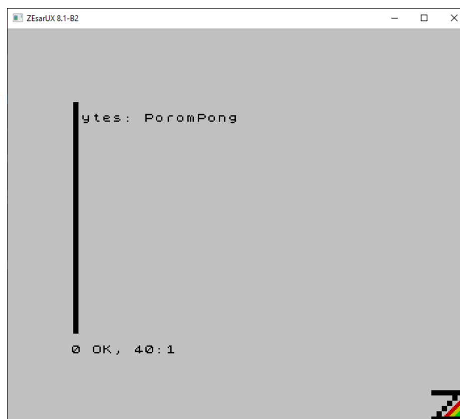
Ya solo queda indicar al programa donde debe salir, incluir el fichero donde se encuentra la rutina NextScan e indicarle a PASMO la dirección a la que tiene que llamar cuando cargue el programa.

```
ret
include "Video.asm"
end   $8000
```

Ahora vamos a compilar el programa, para lo cual vamos a utilizar PASMO. Desde la línea de comandos vamos al directorio donde tenemos los ficheros .asm, y tecleamos lo siguiente:

```
pasmo --name PoromPong --tapbas Main.asm PorompomPong.tap --public
```

Ahora podemos cargar nuestro programa en el emulador de ZX Spectrum y veremos algo así:



Como se ve, ha dibujado una columna vertical, pero es lo suficientemente rápido como para no ver como se dibuja. Para poder verlo vamos a añadir la instrucción `HALT` antes de `DJNZ`. La

instrucción `HALT` espera hasta que se produce una interrupción, que en el caso del ZX Spectrum es provocada por la ULA.

El código resultante es:

```
org     $8000

ld      hl, $4000       ; Apunta HL al primer scanline de la primera línea
                        ; del primer tercio y columna 1 de la pantalla (Columna de 0 a 31)
ld      b, $c0          ; B = 192. Número de scanlines que tiene la pantalla

loop:
ld      (hl), $3c       ; Pinta en la pantalla 001111000
call    NextScan        ; Pasa al siguiente scanline
halt                    ; Descomentar línea si se quiere ver el proceso de pintado
djnz    loop            ; Hasta que B = 0

ret

include "Video.asm"
end     $8000
```

Volvemos a compilar y ahora sí se ve como pinta scanline a scanline. Si queremos que vuelva a ir rápido, comentamos la instrucción `HALT`.

Ahora vamos a implementar, en Video.asm, la rutina que recupera la dirección de memoria del scanline anterior:

```
PreviousScan:
ld      a, h
dec     h
and     $07
ret     nz
```

Lo primero que hacemos es cargar el valor de H, `LD A, H`, tercio y scanline en A, y a continuación, decrementamos H, `DEC H`. Luego nos quedamos con los bits del scanline original, `AND $07`, que tenemos en A, y si no estaba en el scanline 0 salimos de la rutina, `RET NZ`. A contiene el valor original de H.

Si estaba en el scanline 0, al decrementar H ha pasado al scanline 7 de la línea anterior y ha decrementado el tercio.

Ahora hay que calcular la línea:

```
ld      a, l
sub     $20
ld      l, a
```

```
ret    c
```

Cargamos el valor de L, `LD A, L`, línea y columna en A, y le restamos $20, `SUB $20`, para decrementar la línea, volviendo a cargar el valor en L, `LD L, A`. Salimos si hay acarreo, `RET C`, ya que hay cambio de tercio, que se produjo al decrementar el scanline.

En el caso de no haber acarreo, es necesario dejar el tercio como estaba originalmente:

```
ld     a, h
add    a, $08
ld     h, a
ret
```

Cargamos el valor de H, tercio y scanline, en A, `LD A, H` y le sumamos $08 para incrementar el tercio, `ADD A, $08`, volviendo a cargar el valor en H, y salimos de la rutina, `RET`.

El código final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------------
; PreviousScan. https://wiki.speccy.org/cursos/ensamblador/gfx2_direccionamiento
; Obtiene la posición de memoria correspondiente al scanline anterior al indicado.
; 010T TSSS LLLC CCCC
; Entrada:  HL -> scanline actual.
; Salida:   HL -> scanline anterior.
; Altera el valor de los registros AF, BC y HL.
; ------------------------------------------------------------------------------
PreviousScan:
ld      a, h           ; Carga el valor en A
dec     h              ; Decrementa H para decrementar el scanline
and     $07            ; Se queda con los bits del scanline original
ret     nz             ; Si no estaba en el 0, fin de la rutina

; Calcula la línea anterior
ld      a, l           ; Carga el valor de L en A
sub     $20            ; Resta una línea
ld      l, a           ; Carga el valor en L
ret     c              ; Si hay acarreo, fin de la rutina

; Si llega aquí, ha pasado al scanline 7 de la línea anterior
; y ha restado un tercio, que volvemos a sumar
ld      a, h           ; Carga el valor de H en A
add     a, $08         ; Vuelve a dejar el tercio como estaba
ld      h, a           ; Carga el valor en h
ret
```

Por último, volvemos a Main.asm para implementar la prueba de PreviousScan. Vamos añadir el nuevo código después de la instrucción `DJNZ loop`.

Lo primero es cargar en HL la dirección de la VideoRAM donde vamos a pintar, en este caso la esquina inferior derecha:

```
ld     hl, $57ff
```

Si ponemos $57FF en binario:

```
0101 0111 1111 1111
```

Vemos que hace referencia al tercio 2, línea 7, scanline 7 y columna 31.

El bucle vuelve a ser de 192 iteraciones, para dibujar hasta la esquina superior derecha. Cargamos el valor en B:

```
ld     b, $c0
```

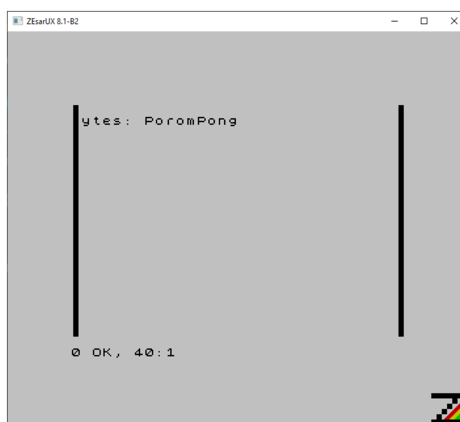
Y luego hacemos el bucle:

```
loopUp:
ld     (hl), $3c
call   PreviousScan
halt
djnz   loopUp
```

La única diferencia con el bucle loop radica en el `CALL`, que en esta ocasión se hace a PreviousScan en lugar de a NextScan. `HALT` está sin comentar para que se pueda apreciar como pinta.

Volvemos a compilar y vemos el resultado cargando el programa generado en el emulador de ZX Spectrum:

```
pasmo --name PoromPong --tapbas Main.asm PorompomPong.tap --public
```



El código completo de Main.asm es:

```
; Dibuja dos líneas verticales, una de abajo a arriba y otra de arriba a abajo
; para probar las rutinas NextScan y PreviousScan.
org     $8000
```

```
ld      hl, $4000       ; Apunta HL al primer scanline, primera línea, primer tercio
                        ; y columna 0 de la pantalla (Columna de 0 a 31)
ld      b, $c0          ; B = 192. Número de scanlines que tiene la pantalla

loop:
ld      (hl), $3c       ; Pinta en la pantalla 001111000
call    NextScan        ; Pasa al siguiente scanline
; halt                  ; Descomentar línea si se quiere ver el proceso de pintado
djnz    loop            ; Hasta que B = 0

ld      hl, $57ff       ; Apunta HL al último scanline, última línea, último tercio
                        ; y columna 31 de la pantalla (Columna de 0 a 31)
ld      b, $c0          ; B = 192. Número de scanlines que tiene la pantalla

loopUp:
ld      (hl), $3c       ; Pinta en la pantalla 001111000
call    PreviousScan    ; Pasa al scanline anterior
; halt                  ; Descomentar línea si se quiere ver el proceso de pintado
djnz    loopUp          ; Hasta que B = 0
ret

include "Video.asm"
end     $8000
```

Paso 2: teclas de control

En este paso vamos a desarrollar la rutina que comprueba si se han pulsado las teclas de control de nuestro juego, y devuelve cuales son las teclas pulsadas.

El teclado del ZX Spectrum está dividido en ocho semi filas, cada una de las cuales contiene cinco teclas.

Cuando se evalúa si se ha pulsado alguna tecla de una semi fila, los valores vienen en un byte, en los bits 0 a 4, cuyos valores son 1 si no se ha pulsado, y 0 si se ha pulsado. El bit 0 hace referencia a la tecla más alejada del centro (Caps Shift, A, Q, 1, 0, P, Enter, Space) y el 4 a la tecla más cercana al centro (V, G, T, 5, 6, Y, H y B).

Cada semi fila está identificada por un número:

Semifila	Valor Hexadecimal	Valor Binario
Caps Shift-V	$FE	1111 1110
A-G	$FD	1111 1101
Q-T	$FB	1111 1011
1-5	$F7	1111 0111
0-6	$EF	1110 1111
P-Y	$DF	1101 1111
Enter-H	$BF	1011 1111
Space-B	$7F	0111 1111

Como se puede observar, para calcular el valor de la semi fila anterior o posterior, solo hay que hacer rotaciones circulares de bits (RLC, RRC).

Dentro de la carpeta Pong, creamos una carpeta llamada Paso02, y dentro de la misma los ficheros Main.asm y Controls.asm.

La rutina que vamos a usar para verificar los controles está sacada del [Curso de Ensamblador para Z80 de Compiler Software]() de [Santiago Romero](). Podéis encontrar dicho curso en [El wiki de speccy.org]().

Los controles que vamos a usar son: A-Z para el jugador 1, y 0-O para el jugador 2.

La rutina que vamos a implementar para comprobar si se ha pulsado alguna de las teclas expuestas, devuelve en el registro D las teclas que se han pulsado, usando el bit 0 para la tecla A, el bit 1 para la tecla Z, el bit 2 para la tecla 0 y el bit 3 para la tecla O. Los valores que toman estos bits son 1 si se pulsado la tecla y 0 en el caso contrario.

Lo primero que va a hacer la rutina es poner a 0 el registro D:

```
ScanKeys:
ld     d, $00
```

A continuación, comprueba si se ha pulsado la tecla A:

```
scanKeys_A:
ld     a, $fd
in     a, ($fe)
```

```
bit   $00, a
jr    nz, scanKeys_Z
set   $00, d
```

Con `LD A, $FD` cargamos el identificador de la semi fila A-G ($FD = 11111101) en A.

A continuación, con `IN A, ($FE)`, leemos el puerto de entrada $FE (254) y dejamos el valor en A. El puerto de entrada $FE es el puerto desde el que leemos el estado del teclado.

Lo siguiente es comprobar si se ha pulsado la tecla A; para ello usamos la sentencia `BIT $00, A`, que evalúa el estado del bit 0 del registro A. Si el bit está a 0 se activa el flag Z, de lo contrario se desactiva.

Con la siguiente instrucción, `JR NZ, scanKeys_Z`, si el bit viene a 1 salta a evaluar la pulsación de la tecla Z.

Si el bit viene a 0, activamos el bit 0 del registro D, `SET $00, D`, para devolver que se ha pulsado la tecla A.

El siguiente paso es comprobar si se ha pulsado la tecla Z:

```
scanKeys_Z:
ld    a, $fe
in    a, ($fe)
bit   $01, a
jr    nz, scanKeys_0
set   $01, d
```

La diferencia con la comprobación de la tecla A radica en que cargamos en A la semi fila Caps Shift-V, `LD A, $FE`, comprobamos el estado del bit 1 correspondiente a la tecla Z, `BIT $01, A`, si no se ha pulsado saltamos a comprobar la pulsación de la tecla 0, `JR NZ, scanKeys_0`, y, por último, activamos el bit 1 de D, `SET $01, D`, si se ha pulsado la tecla Z.

Se puede dar el caso de que se pulsen a la vez las teclas A y Z. Si se diera, vamos a desactivar los indicadores para asimilar que no se ha pulsado ninguna. La otra opción sería dejar los indicadores de las dos teclas pulsadas y mover el personaje primero hacia arriba y luego hacia abajo, quedándose donde estaba.

Vamos a comprobar si se han pulsado las dos teclas, y si es así desactivamos los bits correspondientes:

```
ld    a, d
cp    $03
jr    nz, scanKeys_0
xor   a
ld    d, a
```

Lo primero es cargar el valor de D en A, `LD A, D`, y verificar si el valor es 3, `CP $03`, en cuyo caso se habrían pulsado las dos teclas. Si el valor de la comprobación no es 0, no se han

pulsado las dos teclas y saltamos a comprobar la pulsación de la tecla 0, `JR NZ, scanKeys_0`.

Si el resultado es 0, ponemos A = 0, `XOR A`, y cargamos el valor en D, `LD D, A`.

La instrucción CP evalúa el valor del registro A con el valor de otro registro, un número o el valor de una dirección de memoria apuntada por (HL), (IX+N) o (IY+N). CP resta cualquiera de estos valores al valor del registro A. CP no altera el valor de A, pero sí altera los indicadores (registro F), de la siguiente manera:

Valor del flag	Significado
Z	A = Valor
NZ	A <> Valor
C	A < Valor
NC	A >= Valor

Para cargar 0 en A, en lugar de `LD A, $00` hemos utilizado `XOR A`.

Las instrucciones `AND, OR y XOR`, tienen como destino, siempre, el registro A y el resultado que dan a nivel de bits es el siguiente:

Operación	Bit 1	Bit 2	Resultado
AND	1	1	1
	1	0	0
	0	1	0
	0	0	0
OR	1	1	1
	1	0	1
	0	1	1
	0	0	0
XOR	1	1	0
	1	0	1
	0	1	1
	0	0	0

Como se puede ver en la tabla, `XOR A` siempre da como resultado 0, una operación que tiene 1 byte y consume 4 ciclos de reloj. Por el contrario, `LD A, $00` tiene 2 bytes y consume 7 ciclos de reloj, por lo que ganamos 1 byte y 3 ciclos. Pero no todo son ventajas, ya que `XOR` afecta a los flags mientras que `LD` no.

También podríamos haber puesto D a 0, `LD D, $00`, pero no habríamos visto la instrucción XOR, aunque habríamos ahorrado un ciclo de reloj.

Hay otra forma más óptima de hacerlo; sustituimos `CP $03` por `SUB $03`, y luego cargamos A en D, `LD D, A`:

```
ld    a, d
sub   $03
jr    nz, scanKeys_0
ld    d, a
```

Estaríamos consumiendo 7 ciclos y dos bytes con `SUB $03`, y 4 ciclos y un byte con `LD D, A`, ahorrándonos 3 o 4 ciclos, y un byte.

Por último, hay que comprobar si han pulsado las teclas 0 y O, y si se han pulsado las dos a la vez. El código es casi igual a lo que hemos visto hasta ahora, por lo que vamos a ver el código completo de la rutina:

```
; -------------------------------------------------------------------------------
; ScanKeys
; Escanea las teclas de control y devuelve las pulsadas.
; Salida:      D -> Teclas pulsadas.
;                     Bit 0 -> A pulsada 0/1.
;                     Bit 1 -> Z pulsada 0/1.
;                     Bit 2 -> 0 pulsada 0/1.
;                     Bit 3 -> O pulsada 0/1.
; Altera el valor de los registros AF y D.
; -------------------------------------------------------------------------------
ScanKeys:
ld      d, $00         ; Pone el registro D a 0.

scanKeys_A:
ld      a, $fd         ; Carga en A la semi fila A-G
in      a, ($fe)       ; Lee el estado de la semi fila
bit     $00, a         ; Comprueba si se ha pulsado la A
jr      nz, scanKeys_Z ; Si no se ha pulsado, salta
set     $00, d         ; Pone a 1 el bit correspondiente a la A

scanKeys_Z:
ld      a, $fe         ; Carga en A la semi fila CS-V
in      a, ($fe)       ; Lee el estado de la semi fila
bit     $01, a         ; Comprueba si se ha pulsado la Z
jr      nz, scanKeys_0 ; Si no se ha pulsado, salta
set     $01, d         ; Pone a 1 el bit correspondiente a la Z

; Comprueba que no se hayan pulsado las dos teclas de dirección
ld      a, d           ; Carga el valor de D en A
sub     $03            ; Comprueba si se han pulsado la A y la Z a la vez
jr      nz, scanKeys_0 ; Si no se han pulsado, salta
ld      d, a           ; Pone D a 0

scanKeys_0:
ld      a, $ef         ; Carga la semi fila 0-6
in      a, ($fe)       ; Lee el estado de la semi fila
```

```
bit     $00, a         ; Comprueba si se ha pulsado el 0
jr      nz, scanKeys_O ; Si no se ha pulsado, salta
set     $02, d         ; Pone a 1 el bit correspondiente al 0

scanKeys_O:
ld      a, $cf         ; Carga la semi fila P-Y
in      a, ($fe)       ; Lee el estado de la semi fila
bit     $01, a         ; Comprueba si se ha pulsado la O
ret     nz             ; Si no se ha pulsado, salta
set     $03, d         ; Pone a 1 el bit correspondiente a la O

; Comprueba que no se hayan pulsado las dos teclas de dirección
ld      a, d           ; Carga el valor de D en A
and     $0c            ; Se queda con los bits correspondientes a 0 y O
cp      $0c            ; Comprueba si se han pulsado las dos teclas
ret     nz             ; Si no se han pulsado, sale
ld      a, d           ; Se han pulsado, carga el valor de D en A
and     $03            ; Se queda con los bits correspondientes a la A y Z
ld      d, a           ; Carga el valor en D

ret
```

Las diferencias más importantes con respecto a la comprobación de la pulsación de A-Z, están en la comprobación de si se han pulsado a la vez las dos teclas.

Antes de comprobar si están activos los bits del registro D, que se corresponden con 0 y O ($0C = 0000 1100), hay que quedarse sólo con estos bits, de lo contrario, si se hubieran pulsado la A o la Z, `CP $0C` nunca daría 0, es por eso que antes de esta instrucción se ha incluido `AND $0C`, para quedarnos con el valor de los bits 2 y 3.

La segunda diferencia es la forma en la que ponemos a 0 los bits 2 y 3, en el caso de que se hayan pulsado a la vez 0 y O.

Anteriormente hicimos `XOR A` o `SUB $03` y `LD D, A`, porque lo único que teníamos en A era si se habían pulsado a la vez A y Z, pero esta vez, además de si se han pulsado 0 y O, tenemos las pulsaciones de A y Z, y si hiciéramos `XOR A` o `SUB $03` y `LD D, A`, estaríamos destruyendo esta información.

Para evitar destruir esta información, cargamos en A el valor del registro D, `LD A, D`, luego nos quedamos solo con el valor de los bits 0 y 1, `AND $03`, y volvemos a cargar el valor en D, `LD D, A`. De esta manera hemos puesto a 0 el valor de los bits 2 y 3 sin destruir el valor de los bits 0 y 1.

Podemos optimizar sustituyendo `LD A, D` y `AND $03` por `XOR D`. XOR D tendría el mismo efecto que las otras dos líneas, y solo consumiríamos 4 ciclos de reloj y un byte.

```
Si el valor de A es 00001100 y el valor de D es 00001101
```

```
después de XOR D, el valor de A es 00000001.
```

Ya solo queda probar la rutina. Para ello vamos a pintar en la esquina superior izquierda el valor de D, una vez que vuelve de la rutina de chequeo de las pulsaciones de las teclas. El código lo vamos a escribir en el archivo Main.asm.

El primer paso es especificar la dirección donde se carga el programa:

```
org   $8000
```

Apuntamos HL a la esquina superior izquierda de la pantalla:

```
ld    hl, $4000
```

Y hacemos un bucle infinito que llame a la rutina ScanKeys y cargue en la esquina superior izquierda de la ventana el valor del registro D:

```
Bucle:
call  ScanKeys
ld    (hl), d
jr    Bucle
```

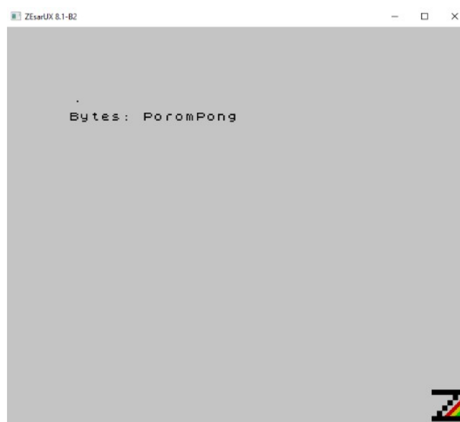
Por último, incluimos el archivo Controls.asm y le indicamos a PASMO la dirección donde llamar cuando cargue el programa.

```
include "Controls.asm"
end   $8000
```

Llegados a este punto, compilamos el programa y cargamos en el emulador para ver el resultado.

```
pasmo --name PoromPong --tapbas Main.asm PorompomPong.tap --public
```

El resultado del programa será algo así:



El código final del archivo Main.asm quedará como sigue:

```
; Comprueba el funcionamiento de los controles A-Z y 0-O
; Pinta la representación de las teclas pulsadas.
org     $8000
ld      hl, $4000      ; Posiciona HL en la primera posición de la pantalla
```

```
Bucle:
call    ScanKeys       ; Escanea las teclas pulsadas
ld      (hl), d        ; Pinta la representación de las teclas pulsadas
jr      Bucle          ; Bucle infinito

include "Controls.asm"
end     $8000
```

Hemos dejado una optimización pendiente, que veremos en la última entrega del tutorial, con la que ahorraremos un ciclo de reloj en la comprobación de cada tecla pulsada, lo que hará un total de 4 ciclos de reloj de ahorro en la rutina ScanKeys.

Paso 3: palas y línea central

Ya hemos adquirido los conocimientos suficientes para empezar con el desarrollo de nuestro Pong. Hemos implementado una buena parte de la base del programa.

En este paso vamos a:

- Cambiar el color del borde.
- Asignar los atributos de color a la pantalla.
- Dibujar la línea central del campo.
- Dibujar las palas de ambos jugadores.
- Mover las palas hacia arriba y hacia abajo.

Como siempre, creamos una carpeta a la que vamos a llamar Paso03, y dentro de la misma creamos los archivos Main.asm y Sprite.asm.

Esta vez no empezamos desde cero, ya que hemos desarrollado en los pasos anteriores código, en los ficheros Controls.asm y Video.asm, que vamos a usar en este paso, por lo que copiamos los dos ficheros en el nuevo directorio.

Cambiar el color del borde

Es el primer paso que vamos a realizar. Aunque el color del borde final será igual al del resto de la pantalla, en los primeros pasos lo vamos a poner en rojo para visualizar los límites de la misma.

Vamos a editar el fichero Main.asm, y lo primero, como siempre, es indicar la dirección de memoria dónde vamos a cargar el programa:

```
org   $8000
```

Lo siguiente es poner el borde en rojo:

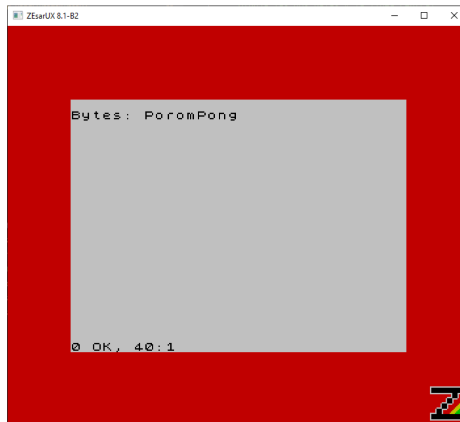
```
ld     a, $02
out    ($fe), a
```

Con `LD A, $02` cargamos el valor del color rojo en A. Luego escribimos este valor en el puerto $FE (256), `OUT ($FE), A`. Este puerto ya lo conocemos, pues es el puerto desde dónde leemos el estado del teclado.

Por último, salimos del programa e indicamos a PASMO dónde llamar cuando lo cargue.

```
ret
end   $8000
```

Compilamos con PASMO y vemos el resultado final:



El código de Main.asm queda así:

```
org     $8000
ld      a, $02              ; A = 2
out     ($fe), a            ; Pone el borde en rojo
ret
end     $8000
```

Asignar los atributos de color a la pantalla.

En nuestro caso, los atributos son blanco para la tinta y negro para el fondo.

Vamos a implementar una rutina, Cls, que limpia la pantalla y pone el fondo en negro y la tinta en blanco.

Los atributos de la pantalla se encuentran a continuación del área donde se dibuja; empieza en la dirección $5800 y tiene una longitud de $300 (768) bytes, 32 columnas por 24 líneas. En el ZX Spectrum, los atributos de color van a nivel de carácter. Cada atributo afecta a un área de 8x8 píxeles, siendo éste el motivo del famoso Attribute Clash.

Los atributos de un carácter están definidos en un byte:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Parpadeo (0/1)	Brillo (0/1)	Fondo (0 a 7)			Tinta (0 a 7)		

La rutina Cls consta de dos partes:

- Limpia la pantalla.
- Asigna el color de tinta y fondo.

Vamos a editar el archivo Video.asm y vamos a implementar la rutina:

```
Cls:
ld      hl, $4000
ld      (hl), $00
ld      de, $4001
ld      bc, $17ff
ldir
```

```
ret
```

Lo primero que hace nuestra rutina es apuntar HL al inicio de la VideoRAM, `LD HL, $4000`, y limpia ese byte de la pantalla, `LD (HL), $00`.

El siguiente paso es apuntar DE a la posición siguiente a HL, `LD DE, $4001`, y cargar en BC el número de bytes a limpiar, `LD BC, $17FF`, que es toda el área de la VideoRAM ($1800) menos uno, que es la posición donde apunta HL, y ya está limpia.

LDIR, **LoadData, Increment and Repeat**, carga el valor que hay en la posición de memoria a la que apunta HL, a la posición de memoria a la que apunta DE. Una vez realizado esto, incrementa HL y DE. Repite en bucle hasta que BC llegue a 0. Por último, salimos de la rutina.

Abrimos el archivo Main.asm y antes de RET añadimos la llamada a Cls:

```
call  Cls
```

Antes de END $8000, añadimos la línea para incluir el archivo Video.asm:

```
include "Video.asm"
```

Compilamos con PASMO y cargamos en el emulador:

Como se puede aprecia en la imagen, ya no sale la línea `Bytes: PoromPong`, lo cual demuestra que hemos limpiado la pantalla.

Para implementar la segunda parte de la rutina, la asignación de los atributos de color, vamos a escribir las siguientes líneas justo antes de la instrucción `RET` de la rutina Cls:

```
ld    hl, $5800
ld    (hl), $07
ld    de, $5801
ld    bc, $2ff
ldir
```

Lo primero que hace esta parte de la rutina es apuntar HL al inicio del área de atributos, `LD HL, $5800`, y pone esa zona sin parpadeo, sin brillo, con el fondo en negro y la tinta en blanco, `LD (HL), $07`.

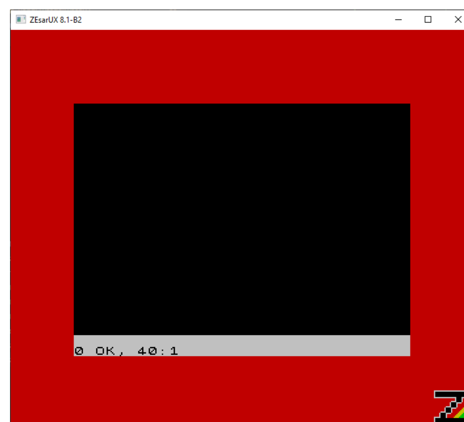
$07 = 0000 0111 = 0 (parpadeo) 0 (brillo) 000 (fondo) 111 (tinta)

El siguiente paso es apuntar DE a la posición siguiente a HL, `LD DE, $5801`, y cargar en BC el número de bytes a cargar, `LD BC, $2FF`, que es toda el área de atributos ($300) menos uno, que es la posición donde apunta HL, y ya tiene los atributos. Se ejecuta `LDIR`, y se asigna el color a toda la pantalla.

El código completo de la rutina es:

```
; -----------------------------------------------------------------------------
; Limpia la pantalla, tinta 7, fondo 0.
; Altera el valor de los registros AF, BC, DE y HL.
; -----------------------------------------------------------------------------
Cls:
; Limpia los píxeles de la pantalla
ld      hl, $4000           ; Carga en HL el inicio de la VideoRAM
ld      (hl), $00           ; Limpia los píxeles de esa dirección
ld      de, $4001           ; Carga en DE la siguiente posición de la VideoRAM
ld      bc, $17ff           ; 6143 repeticiones
ldir                        ; Limpia todos los píxeles de la VideoRAM

; Pone la tinta en blanco y el fondo en negro
ld      hl, $5800           ; Carga en HL el inicio del área de atributos
ld      (hl), $07           ; Lo pone con la tinta en blanco y el fondo en negro
ld      de, $5801           ; Carga en DE la siguiente posición del área de atributos
ld      bc, $2ff            ; 767 repeticiones
ldir                        ; Asigna el valor a toda el área de atributos

ret
```

Llegados a este punto, compilamos y vemos el resultado:



Como se puede observar, además de limpiar la pantalla, ha puesto el fondo en negro y la tinta en blanco, aunque al no haber pintado nada en la pantalla, no se ve si la tinta está realmente en blanco.

Para ver distintos efectos, cambiad los valores que cargáis en (HL).

Esta rutina se puede cambiar, haciéndonos ahorrar 8 ciclos de reloj y 4 bytes. Dejamos en vuestras manos averiguar la manera de hacerlo, y daremos la solución en el último capítulo. No os preocupéis, no es una rutina crítica, así que no va a afectar al desarrollo de nuestro videojuego.

Dibujar la línea central del campo.

La línea central del campo está compuesta por un primer scanline en blanco, otros seis con el bit 7 a 1 y un último scanline en blanco:

00000000

10000000

10000000

10000000

10000000

10000000

10000000

00000000

En este caso solo vamos a definir la parte en blanco y la parte que pinta línea. Abrimos el fichero Sprite.asm y añadimos las siguientes líneas:

```
ZERO: EQU   $00

LINE: EQU   $80
```

Con la directiva EQU se definen valores constantes que no se compilan, al contrario, lo que hace el compilador es sustituir todas las referencias que haya en el código a estas etiquetas, por el valor que se ha asignado a las mismas.

```
Ejemplo:    ld     a, ZERO      ->     Compilador  ->     ld     a, $00
```

Una vez que tenemos el “sprite” de la línea, vamos a implementar la rutina para pintarla. Volvemos al archivo Video.asm:

```
PrintLine:

ld     b, $18

ld     hl, $4010
```

Vamos a pintar el “sprite” de nuestra línea en las 24 líneas de la pantalla, `LD B, $18`, y vamos a empezar en el primer scanline, de la primera línea, del primer tercio, columna 16, `LD HL, $4010`.

```
printLine_loop:

ld     (hl), ZERO

inc    h

push   bc
```

Pintamos el primer scanline en blanco, `LD (HL), ZERO`, luego pasamos al siguiente scanline, `INC H`, y por último preservamos el valor de BC en la pila, ya que vamos a usar B para hacer un bucle que pinte la parte que se ve de la línea.

Para cambiar de scanline, directamente incrementamos H en lugar de llamar a NextScan. ¿Por qué? Sencillo. Dado que vamos a pintar los 8 scanlines de un mismo carácter, ni cambiamos de línea, ni de tercio, por lo que con aumentar el scanline es suficiente, y ahorramos tiempo de proceso y bytes.

Otra cosa que hacemos es subir un valor a la pila, concretamente el de BC. Es muy importante recordar que cada PUSH debe tener un POP, y además si hay varios PUSH, tiene que haber el mismo número de POP, pero en orden inverso:

```
push  af
push  bc
pop   bc
pop   af
```

Ahora vamos a hacer el bucle que pinte la parte que se ve de la línea:

```
ld    b, $06
printLine_loop2:
ld    (hl), LINE
inc   h
djnz  printLine_loop2
pop   bc
```

Lo primero es indicar el número de iteraciones del nuevo bucle, `LD B, $06`, pintamos el scanline con la parte visible de la línea, `LD (HL), LINE`, pasamos al siguiente scanline, `INC H`, y repetimos hasta que B valga 0, `DJNZ printLine_loop2`. Cuando B valga 0, recuperamos el valor de BC de la pila para continuar con el bucle de las 24 líneas de la pantalla, `POP BC`.

Y llegamos así a la parte final de la rutina:

```
ld    (hl), ZERO
call  NextScan
djnz  printLine_loop
ret
```

Pintamos el último scanline del carácter, `LD (HL), ZERO`, recuperamos el siguiente scanline, `CALL NextScan`, y repetimos hasta que B valga 0 y se hayan pintado las 24 líneas de la pantalla, `DJNZ printLine_loop`. Esta vez sí llamamos a NextScan, ya que cambiamos de línea.

El aspecto final de la rutina es el siguiente:

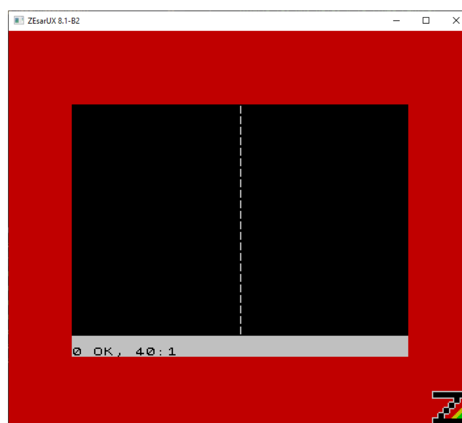
```
; -------------------------------------------------------------------------------
; Imprime la línea central.
; Altera el valor de los registros AF, B y HL.
```

```
; ------------------------------------------------------------------------------
PrintLine:
ld      b, $18                  ; Se imprime en las 24 líneas de pantalla
ld      hl, $4010               ; Se empieza en la línea 0, columna 16

printLine_loop:
ld      (hl), ZERO              ; En el primer scanline se imprime el byte en blanco
inc     h                       ; Pasa al siguiente scanline

push    bc                      ; Preserva el valor de BC para realizar el segundo bucle
ld      b, $06                  ; Se imprime seis veces
printLine_loop2:
ld      (hl), LINE              ; Imprime el byte de la línea, $10, b00010000
inc     h                       ; Pasa el siguiente scanline
djnz    printLine_loop2         ; Hasta que B = 0
pop     bc                      ; Recupera el valor de BC
ld      (hl), ZERO              ; Imprime el último byte de la línea a 0
call    NextScan                ; Pasa al siguiente scanline
djnz    printLine_loop          ; Hasta que B = 0 = 24 líneas
ret
```

Y ahora ya sólo queda probarlo, para lo cual abrimos el fichero Main.asm y añadimos tras la llamada a Cls, la llamada a PrintLine e incluimos el fichero Sprite.asm, igual que hicimos con el fichero Video.asm:

```
call   PrintLine
include "Sprite.asm"
```

Compilamos y vemos el resultado en el emulador:



Ahora sí se puede observar que habíamos puesto la tinta en blanco.

Dibujar las palas de ambos jugadores.

En este paso vamos a dibujar las palas de ambos jugadores, que van a ocupar 1x3 caracteres, 1 byte (8 píxeles) y 24 scanlines.

Vamos a usar el mismo tipo de definición que usamos para definir la línea horizontal, y lo vamos a hacer en el archivo Sprite.asm:

```
PADDLE:     EQU   $3c
```

Esta sería la parte visible de la pala, 00111100, ya que vamos a pintar el primer scanline en blanco, 22 scanlines con esta definición y el último scanline en blanco.

Las palas van a ser elementos móviles, por lo que además de su “sprite”, necesitamos saber en qué posición se encuentran y cuáles son los márgenes superior e inferior a los que las podemos mover.

Seguimos en el fichero Sprite.asm:

```
PADDLE_BOTTOM:    EQU   $a8   ; TTLLLSSS

PADDLE_TOP:       EQU   $00   ; TTLLLSSS

paddle1pos:       dw    $4861 ; 010T TSSS LLLC CCCC

paddle2pos:       dw    $487e ; 010T TSSS LLLC CCCC
```

En las dos primeras constantes, que son los límites hasta donde podemos mover las palas, vamos a especificar la coordenada Y expresada en tercio, línea y scanline. Mientras que PADDLE_TOP sí apunta al límite superior de la pantalla (tercio 0, línea 0, scanline 0), PADDLE_BOTTOM no apunta al límite inferior de la pantalla (tercio 2, línea 7, scanline 7), por el contrario, apunta al tercio 2, línea 5, scanline 0, que es resultado de restarle al límite inferior ($BF), 23 scanlines para que podamos pintar los 24 scanlines del sprite de la pala, sin invadir el área de atributos de la pantalla.

paddle1pos y paddle2pos no son constantes, pues estos valores van a cambiar respondiendo a las pulsaciones de las teclas de control.

La posición inicial de las palas es:

	Pala 1	Pala 2
Tercio	1	1
Línea	3	3
Scanline	0	0
Columna	1	30

Una vez definido esto, vamos al archivo Video.asm e implementamos la rutina que dibuja las palas. Esta rutina tiene como parámetro de entrada la posición de la pala, que se recibe en HL. Es necesario porque tenemos dos palas que imprimir, y la otra alternativa sería duplicar la rutina y que cada una imprimiera una pala.

```
PrintPaddle:

ld    (hl), ZERO

call  NextScan
```

Lo primero que hace es pintar en blanco el primer scanline de la pala, `LD (HL), ZERO`, y luego obtiene el siguiente scanline.

Al contrario de lo que pasaba al pintar la línea central, en esta rutina si son necesarias las llamadas a NextScan. Nuestro movimiento de la pala va a ser pixel a pixel, esto en vertical es

scanline a scanline, lo que hace que no sepamos de antemano cuándo cambiamos de línea (en realidad sí podríamos saberlo).

Lo siguiente es pintar la parte visible de la pala:

```
ld    b, $16
printPaddle_loop:
ld    (hl), PADDLE
call  NextScan
djnz  printPaddle_loop
```

La parte visible de la pala la vamos a pintar en 22 scanlines, `LD B, $16`, cargando en la posición apuntada por HL el sprite de la pala, `LD (HL), PADDLE`, y obteniendo el siguiente scanline, `CALL NextScan`, hasta que B valga 0, `DJNZ printPaddle_loop`.

Por último, pinta en blanco el último scanline de la pala:

```
ld    (hl), ZERO
ret
```

Pintar en blanco el primer y el último scanline sirve para que, al mover la pala, se vaya auto borrando y no deje rastro.

El aspecto final de la rutina es el siguiente:

```
; ------------------------------------------------------------------------------
; Imprime la pala.
; Entrada:     HL -> Posición de la pala
; Altera el valor de los registros B y HL.
; ------------------------------------------------------------------------------
PrintPaddle:
ld      (hl), ZERO            ; Imprime el primer byte de la pala en blanco
call    NextScan              ; Pasa al siguiente scanline
ld      b, $16                ; Pinta el byte visible de la pala 22 veces
printPaddle_loop:
ld      (hl), PADDLE          ; Imprime el byte de la pala
call    NextScan              ; Pasa al siguiente scanline

djnz    printPaddle_loop      ; Hasta que B = 0

ld      (hl), ZERO            ; Imprime el último byte de la pala en blanco


ret
```

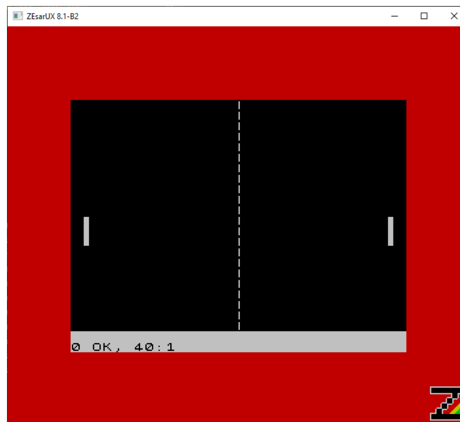
Por último, tenemos que probar si nuestra rutina funciona. Abrimos el archivo Main.asm y añadimos después de la llamada a PrintLine:

```
ld    hl, (paddle1pos)
```

```
call  PrintPaddle
ld    hl, (paddle2pos)
call  PrintPaddle
```

Cargamos en HL la posición de la pala 1, `LD HL, (paddle1pos)`, y la pintamos, `CALL PrintPaddle`. Hacemos lo mismo con la pala 2.

Compilamos y vemos los resultados:



Mover las palas hacia arriba y hacia abajo.

Abordamos la última parte del paso 3.

Anteriormente declaramos unas constantes con los límites inferior y superior. Ahora vamos a implementar las rutinas que comprueban si una posición de memoria, de la VideoRAM, ha llegado o está fuera de un límite especificado.

El conjunto de rutinas que vamos a implementar, recibe en el registro A el límite en formato TTLLLSSS, y la posición actual en HL en formato 010TTSSS LLLCCCCC. Estas rutinas devuelven Z si se ha alcanzado el límite y NZ en el caso contrario:

```
CheckBottom:
call  checkVerticalLimit
ret   c
```

Lo primero que hace es llamar a la rutina checkVerticalLimit, `CALL checkVerticalLimit`, y en el caso de que haya acarreo sale, `RET C`, con NZ. Si hay acarreo, la posición de memoria está por encima del límite inferior.

```
checkBottom_bottom:
xor   a
ret
```

Si llega hasta aquí es porque ha llegado al límite inferior, activa el flag Z, `XOR A`, y sale, `RET`.

Esta rutina no hace gran cosa, por lo que se puede suponer que el grueso de la lógica estará en checkVerticalLimit.

Vamos a implementar la rutina para el límite superior:

```
CheckTop:
call  checkVerticalLimit
jr    c, checkTop_top
ret   nz
```

Igual que en la rutina anterior, se llama a checkVerticalLimit. En este caso no se ha llegado al límite si no hay acarreo y el resultado de checkVerticalLimit no es 0, o lo que es lo mismo, es mayor de 0, de ahí la doble condición, `JR C, checkTop_top` y `RET NZ`.

```
checkTop_top:
xor   a
ret
```

Llega aquí si el resultado de checkVerticalLimit es <= 0 (hay acarreo o el resultado es 0), en cuyo caso activa el flag Z, `XOR A`, y sale, `RET`.

El grueso de la detección de los límites, inferior y superior, lo realiza la rutina checkVerticalLimit, que recibe en A el límite vertical (TTLLLSSS) y en HL la posición actual (010TTSSS LLLCCCCC), o posición con la qué comparar.

Debido al distinto formato que tenemos en HL y en A, el primer paso es pasar el contenido que tiene HL al mismo formato que tiene el contenido de A.

```
checkVerticalLimit:
ld    b, a
ld    a, h
and   $18
rlca
rlca
rlca
ld    c, a
```

Lo primero que hacemos es preservar el valor de A, `LD B, A`, y acto seguido cargamos el valor de H en A, `LD A, H`, y nos quedamos con el tercio, `AND $18`. Rotamos circularmente tres veces el registro A hacia la izquierda, `RLCA`, para poner el tercio en los bits 6 y 7, y cargamos el valor en C, `LD C, A`. Ahora C tiene el tercio de la posición que hemos recibido en HL.

```
ld    a, h
and   $07
or    c
ld    c, a
```

Volvemos a cargar el valor de H en A, `LD A, H`, pero esta vez nos quedamos con el scanline, `AND $07`. Ahora tenemos en A el scanline que viene en HL, y le añadimos el tercio que hemos guardado en C, `OR C`, y cargamos el resultado en C, `LD C, A`. Ahora C tiene el tercio y el scanline que hemos recibido en HL, pero con el mismo formato que el valor que hemos recibido en A (TT000SSS).

```
ld    a, l
and   $e0
rrca
rrca
or    c
```

Ahora vamos a poner el valor de la línea donde le corresponde, cargando el valor de L en A, `LD A, L`, quedándonos con los bits donde viene la línea, `AND $E0`, y rotando circularmente dos veces los bits resultantes para poner la línea en los bits 3, 4 y 5, `RRCA`. Por último, agregamos el tercio y el scanline que hemos guardado en C, `OR C`, de tal manera que en A tenemos ahora el tercio, la línea y el scanline que venían en HL, pero con el formato que necesitamos (TTLLLSSS).

```
cp    b
ret
```

El último paso es comparar lo que ahora tenemos en A con lo que tenemos en B, que es el valor original de A (límite vertical), `CP B`.

Esta última operación va a alterar, entre otros, los flags de acarreo y cero:

Resultado	Z	C
A = B	1 - Z	0 - NC
A < B	0 - NZ	1 - C
A > B	0 - NZ	0 - NC

Dependiendo de estos flags, y si se está evaluando el límite inferior o el superior, sabremos si se ha llegado o traspasado dicho límite.

El código completo de este conjunto de rutinas es el siguiente:

```
; -------------------------------------------------------------------------------
; Evalúa si se ha alcanzado el límite inferior.
; Entrada:     A -> Límite superior (TTLLLSSS).
;              HL -> Posición actual (010TTSSS LLLCCCCC).
; Salida:      Z = Se ha alcanzado.
;              NZ = No se ha alcanzado.
; Altera el valor de los registros AF y BC.
; -------------------------------------------------------------------------------
CheckBottom:
call    checkVerticalLimit    ; Compara la posición actual con el límite
; Si Z o NC, ha llegado al tope, se pone Z, de lo contrario NZ
ret     c
checkBottom_bottom:
xor     a                     ; Activa Z
ret
```

```
; ---------------------------------------------------------------------------
; Evalúa si se ha alcanzado el límite superior.
; Entrada:     A -> Margen superior (TTLLLSSS).
;              HL -> Posición actual (010TTSSS LLLCCCCC).
; Salida:      Z = Se ha alcanzado.
;              NZ = No se ha alcanzado.
; Altera el valor de los registros AF y BC.
; ---------------------------------------------------------------------------
CheckTop:
call   checkVerticalLimit    ; Compara la posición actual con el límite
; Si Z o C, ha llegado al tope, se pone Z, de lo contrario NZ
jr     c, checkTop_top       ; Ha llegado al límite superior y salta
ret    nz                    ; No ha llegado al límite superior y sale
checkTop_top:
xor    a                     ; Activa Z
ret

; ---------------------------------------------------------------------------
; Evalúa si se ha alcanzado el límite vertical.
; Entrada:     A -> Límite vertical (TTLLLSSS).
;              HL -> Posición actual (010TTSSS LLLCCCCC).
; Altera el valor de los registros AF y BC.
; ---------------------------------------------------------------------------
checkVerticalLimit:
ld     b, a           ; Guarda el valor de A en B
ld     a, h           ; Carga en A el valor de H (010TTSSSS)
and    $18            ; Se queda con el tercio
rlca
rlca
rlca                  ; Pone el valor del tercio en los bits 6 y 7
ld     c, a           ; Carga el valor en C
ld     a, h           ; Vuelve a cargar en A el valor de H (010TTSSSS)
and    $07            ; Se queda con el scanline
or     c              ; Añade el tercio
ld     c, a           ; Carga el valor en C
ld     a, l           ; Carga en A el valor de L (LLLCCCCC)
and    $e0            ; Se queda con la línea
rrca
rrca                  ; Pone el valor de la línea en los bits 3, 4 y 5
or     c              ; Añade el tercio y el scanline. A = TTLLLSSS
cp     b              ; Lo compara con B. B = valor original de A = Límite vertical
```

```
ret
```

Usando estas rutinas, ya podemos implementar el movimiento de las palas y evitar que se salgan de la pantalla.

Editamos el fichero Main.asm e incluimos el fichero Controls.asm:

```
include “Controls.asm”
```

Vamos a implementar un bucle infinito en el que se evalúa si se ha pulsado alguna tecla de control, en cuyo caso movemos la pala que corresponda. El bucle lo vamos a implementar justo después de la llamada a PrintLine:

```
loop:
call  ScanKeys
```

Lo primero que hace el bucle es evaluar si se ha pulsado alguna de las teclas de control, `CALL ScanKeys`.

```
MovePaddle1Up:
bit   $00, d
jr    z, MovePaddle1Down
ld    hl, (paddle1pos)
ld    a, PADDLE_TOP
call  CheckTop
jr    z, MovePaddle2Up
call  PreviousScan
ld    (paddle1pos), hl
jr    MovePaddle2Up
```

Después de evaluar los controles, evalúa si se ha pulsado la tecla de control para mover la pala 1 hacia arriba, `BIT $00, D`, y si no es así salta a la siguiente comprobación, `JR Z, MovePaddle1Down`.

Para mover la pala hacia arriba tenemos que ver si al moverla se sale del límite superior, para lo cual necesitamos saber la posición actual de la pala, `LD HL, (paddle1pos)`, obtener el límite superior, `LD A, PADDLE_TOP`, y verificar si se ha alcanzado, `CALL CheckTop`.

Si CheckTop activa el flag Z significa que hemos alcanzado el límite, por lo que saltamos a comprobar el movimiento de la pala 2, `JR Z, MovePaddle2Up`.

Si no se activa el flag Z, obtenemos la posición en la que se debe pintar la pala, `CALL PreviousScan`, y la cargamos en memoria, `LD (paddle1pos), HL`. Por último, saltamos a comprobar el movimiento de la pala 2, `JR MovePaddle2Up`.

Si no se ha pulsado la tecla de control arriba de la pala 1, se verifica si se ha pulsado la de abajo:

```
MovePaddle1Down:
bit   $01, d
```

```
jr    z, MovePaddle2Up
ld    hl, (paddle1pos)
ld    a, PADDLE_BOTTOM
call  CheckBottom
jr    z, MovePaddle2Up
call  NextScan
ld    (paddle1pos), hl
```

Evalúa si se ha pulsado la tecla de control para mover la pala 1 hacia abajo, `BIT $01, D`, y si no es así salta a la siguiente comprobación, `JR Z, MovePaddle2Up`.

Para mover la pala hacia abajo tenemos que comprobar si, al moverla, se sale del límite inferior, para lo cual necesitamos saber la posición actual de la pala, `LD HL, (paddle1pos)`, obtener el límite inferior, `LD A, PADDLE_BOTTOM`, y verificar si se ha alcanzado, `CALL CheckBottom`.

Si CheckBottom activa el flag Z significa que hemos alcanzado el límite, por lo que saltamos a comprobar el movimiento de la pala 2, `JR Z, MovePaddle2Up`.

Si no se activa el flag Z, obtenemos la posición en la que se debe pintar la pala, `CALL NextScan`, y la cargamos en memoria, `LD (paddle1pos), HL`. En esta ocasión no saltamos, ya que en la siguiente instrucción se empieza a comprobar el movimiento de la pala 2.

Debido a que la comprobación del movimiento de la pala 2 es muy parecido al de la pala 1, cambian las posiciones de memoria para obtener la posición de la pala 2 y las de salto, no vamos a entrar a explicarlo en detalle:

```
MovePaddle2Up:
bit   2, d
jr    z, MovePaddle2Down
ld    hl, (paddle2pos)
ld    a, PADDLE_TOP
call  CheckTop
jr    z, MovePaddleEnd
call  PreviousScan
ld    (paddle2pos), hl
jr    MovePaddleEnd

MovePaddle2Down:
bit   3, d
jr    z, MovePaddleEnd
ld    hl, (paddle2pos)
ld    a, PADDLE_BOTTOM
```

```
call  CheckBottom
jr    z, MovePaddleEnd
call  NextScan
ld    (paddle2pos), hl

MovePaddleEnd:
```

La última línea, `MovePaddleEnd`, es una etiqueta que hemos usado para poder saltar a la zona donde se pintan las palas.

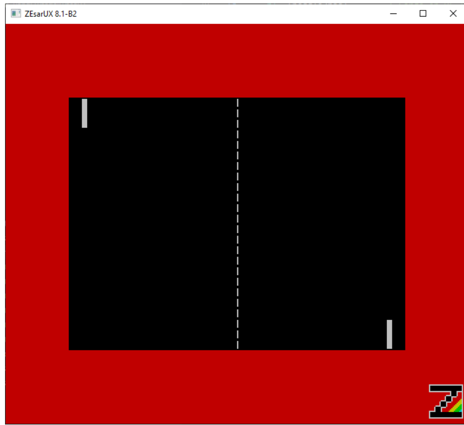
Por último, después de pintar las palas vamos a sustituir `RET` por `JR loop`, para quedarnos en un bucle infinito.

El código final del archivo Main.asm queda como sigue:

```
; Dibuja las dos palas y la línea central.
; Mueve las palas arriba y abajo como respuesta a la pulsación de las teclas de control.
org     $8000
ld      a, $02                ; A = 2
out     ($fe), a              ; Pone el borde en rojo

call    Cls                   ; Limpia la pantalla
call    PrintLine             ; Imprime la línea central

loop:
call    ScanKeys              ; Escanea las teclas pulsadas

MovePaddle1Up:
bit     $00, d                ; Evalúa si se ha pulsado la A
jr      z, MovePaddle1Down    ; Si no se ha pulsado salta
ld      hl, (paddle1pos)      ; Carga en HL la posición de la pala 1
ld      a, PADDLE_TOP         ; Carga en A el margen superior
call    CheckTop              ; Evalúa si se ha alcanzado el margen superior
jr      z, MovePaddle2Up      ; Si se ha alcanzado, salta
call    PreviousScan          ; Obtiene el scanline anterior a la posición de la pala 1
ld      (paddle1pos), hl      ; Carga en memoria la nueva posición de la pala 1
jr      MovePaddle2Up         ; Salta

MovePaddle1Down:
bit     $01, d                ; Evalúa si se ha pulsado la Z
jr      z, MovePaddle2Up      ; Si no se ha pulsado salta
ld      hl, (paddle1pos)      ; Carga en HL la posición de la pala 1
ld      a, PADDLE_BOTTOM      ; Carga en A el margen inferior
```

```
call    CheckBottom           ; Evalúa si se ha alcanzado el margen inferior
jr      z, MovePaddle2Up      ; Si se ha alcanzado, salta
call    NextScan              ; Obtiene el scanline siguiente a la posición de la pala 1
ld      (paddle1pos), hl      ; Carga en memoria la nueva posición de la pala 1

MovePaddle2Up:
bit     $02, d                ; Evalúa si se ha pulsado el 0
jr      z, MovePaddle2Down    ; Si no se ha pulsado salta
ld      hl, (paddle2pos)      ; Carga en HL la posición de la pala 2
ld      a, PADDLE_TOP         ; Carga en A el margen superior
call    CheckTop              ; Evalúa si se ha alcanzado el margen superior
jr      z, MovePaddleEnd      ; Si se ha alcanzado, salta
call    PreviousScan          ; Obtiene el scanline anterior a la posición de la pala 2
ld      (paddle2pos), hl      ; Carga en memoria la nueva posición de la pala 2
jr      MovePaddleEnd         ; Salta

MovePaddle2Down:
bit     $03, d                ; Evalúa si se ha pulsado la O
jr      z, MovePaddleEnd      ; Si no se ha pulsado salta
ld      hl, (paddle2pos)      ; Carga en HL la posición de la pala 2
ld      a, PADDLE_BOTTOM      ; Carga en A el margen inferior
call    CheckBottom           ; Evalúa si se ha alcanzado el margen inferior
jr      z, MovePaddleEnd      ; Si se ha alcanzado, salta
call    NextScan              ; Obtiene el scanline siguiente a la posición de la pala 2
ld      (paddle2pos), hl      ; Carga en memoria la nueva posición de la pala 2

MovePaddleEnd:
ld      hl, (paddle1pos)      ; Carga en HL la posición de la pala 1
call    PrintPaddle           ; Pinta la pala 1
ld      hl, (paddle2pos)      ; Carga en HL la posición de la pala 2
call    PrintPaddle           ; Pinta la pala 2
jr      loop                  ; Bucle infinito

include "Controls.asm"
include "Sprite.asm"
include "Video.asm"


end     $8000
```

Compilamos y vemos los resultados en el emulador:


ZEsarUX 8.1-B2

Paso 4: empezamos a mover la bola

Creamos la carpeta Paso04, dentro de la misma creamos el archivo Main.asm y copiamos los archivos Sprite.asm y Video.asm que tenemos en la carpeta Paso03.

Empezamos editando el archivo Sprite.asm para definir los datos necesarios relativos a la bola:

```
BALL_BOTTOM:        EQU    $ba
BALL_TOP:           EQU    $00
```

Como hicimos con las palas, definimos los límites inferior y superior para la bola, en formato TTLLLSSS.

```
ballPos:            dw           $4870
ballSetting:        db           $00
ballRotation:       db           $f8
```

Al igual que con las palas, vamos a usar una variable donde vamos a tener la posición de la bola en cada momento, `ballPos`.

En `ballSetting` vamos a guardar en los bits 0 a 3 la velocidad X, en los bits 4 y 5 la velocidad Y, en el bit 6 la dirección X (0 derecha / 1 izquierda) y en el bit 7 la dirección Y (0 arriba / 1 abajo).

Por último, en `ballRotation` vamos a guardar la rotación de la bola, indicando con los valores positivos la rotación hacia la derecha y con los negativos hacia la izquierda.

La rotación es necesaria debido a la forma en la que vamos a realizar el movimiento horizontal.

La bola va a constar de un scanline en blanco, cuatro scanlines con la parte visible y otro scanline en blanco. Los scanlines en blanco hacen que la bola no deje rastro al moverse.

Vamos a definir 2 bytes para pintar la bola, y a definir cada movimiento píxel a píxel:

```
; Sprite de la bola. 1 línea a 0, 4 líneas visibles, 1 línea a 0
ballRight:          ; Derecha       Sprite        Izquierda
db     $3c, $00     ; +0/$00    00111100 00000000 -8/$f8
db     $1e, $00     ; +1/$01    00011110 00000000 -7/$f9
db     $0f, $00     ; +2/$02    00001111 00000000 -6/$fa
db     $07, $80     ; +3/$03    00000111 10000000 -5/$fb
db     $03, $c0     ; +4/$04    00000011 11000000 -4/$fc
db     $01, $e0     ; +5/$05    00000001 11100000 -3/$fd
db     $00, $f0     ; +6/$06    00000000 11110000 -2/$fe
db     $00, $78     ; +7/$07    00000000 01111000 -1/$ff
ballLeft:
db     $00, $3c     ; +8/$08    00000000 00111100 +0/$00
```

Cada línea define la parte visible de la bola, dependiendo de cómo estén los píxeles. Definimos dos bytes por cada posición. En el comentario vemos la rotación cuando la bola va hacia la derecha, los bits que vamos a pintar, y la rotación cuando la bola va hacia la izquierda.

La bola inicialmente se pinta tal y como muestra el primer sprite:

```
00111100 00000000
```

Si se mueve un píxel a la derecha, no cambiamos la posición de la bola, cambiamos la rotación y pintamos el segundo sprite:

```
00011110 00000000
```

Al llegar a la última rotación, es cuando cambiamos la posición de la bola, más concretamente la columna. El aspecto final del código es:

```
; Limites de los objetos en pantalla
BALL_BOTTOM:   EQU     $ba     ; TTLLLSSS
BALL_TOP:      EQU     $00     ; TTLLLSSS

; Sprite de la bola. 1 línea a 0, 4 líneas 3c, 1 línea a 0
ballRight:             ; Derecha       Sprite               Izquierda
db      $3c, $00       ; +0/$00       00111100 00000000     -8/$f8
db      $1e, $00       ; +1/$01       00011110 00000000     -7/$f9
db      $0f, $00       ; +2/$02       00001111 00000000     -6/$fa
db      $07, $80       ; +3/$03       00000111 10000000     -5/$fb
db      $03, $c0       ; +4/$04       00000011 11000000     -4/$fc
db      $01, $e0       ; +5/$05       00000001 11100000     -3/$fd
db      $00, $f0       ; +6/$06       00000000 11110000     -2/$fe
db      $00, $78       ; +7/$07       00000000 01111000     -1/$ff
ballLeft:
db      $00, $3c       ; +8/$08       00000000 00111100     +0/$00

; Posición de la bola
ballPos:       dw      $4870          ; 010T TSSS LLLC CCCC

; Velocidad y dirección de la bola.
; bits 0 a 3:  velocidad X: 1 a 4
; bits 4 a 5:  velocidad Y: 0 a 3
; bit 6:       dirección X: 0 derecha / 1 izquierda
; bit 7:       dirección Y: 0 arriba / 1 abajo
ballSetting:   db      $00
; Rotación de la bola
; Valores positivos derecha, negativos izquierda
ballRotation:  db      $f8
```

Ahora vamos a implementar, en el archivo Video.asm, la rutina que pinta la bola, que vamos a poner después de la rutina PreviousScan:

```
PrintBall:
ld    b, $00
ld    a, (ballRotation)
ld    c, a
cp    $00
ld    a, $00
jp    p, printBall_right
```

Lo primero es averiguar hacia dónde va la bola, izquierda o derecha. Una vez averiguado, al sprite base de la bola hay que sumarle o restarle la rotación, para obtener el sprite correcto. La dirección del sprite base la vamos a guardar en HL y restaremos o sumaremos la rotación que tendremos en BC, por eso lo primero es poner B a 0, `LD B, $00`.

El siguiente paso es cargar la rotación de la bola en A, `LD A, (ballRotation)`, y de ahí cargarlo en C, `LD C, A`. Podríamos cargar el valor directamente en C, previo paso por HL, pero dependiendo del valor obtenemos si va a derecha o izquierda. Para obtener este valor, comparamos el valor con 0, y como las comparaciones siempre se hacen contra el registro A, de ahí que sea necesario cargar la rotación en este registro.

Comparamos el valor de A con 0, `CP A, $00`, y si el resultado es positivo la bola se mueve hacia la derecha y salta, `JP P, printBall_right`. Antes de eso hemos cargado 0 en A para los siguientes cálculos, `LD A, $00`.

Continuamos implementando el movimiento hacia la izquierda:

```
printBall_left:
ld    hl, ballLeft
sub   c
add   a, a
ld    c, a
sbc   hl, bc
jr    printBall_continue
```

Si la bola se mueve hacia la izquierda, lo primero es cargar en HL la dirección del sprite base izquierda, `LD HL, ballLeft`.

En este punto A vale 0, por lo que se le resta la rotación que tenemos en C, de esta forma conseguimos el valor a restar para situarnos en el sprite correcto:

```
Ejemplo:    C = $FF, A = $00 ✉ A – C = $01
```

Debido a que cada sprite ocupa 2 bytes, hay que duplicar el valor que se va a restar a HL, `ADD A, A`, y posteriormente cargarlo en C, `LD C, A`.

Ahora ya podemos calcular la dirección de memoria donde se encuentra el sprite a imprimir, `SBC HL, BC`, y saltar a imprimir la bola, `JR printBall_continue`.

Implementamos ahora el movimiento hacia la derecha:

```
printBall_right:
ld    hl, ballRight
add   a, c
add   a, a
ld    c, a
add   hl, bc
```

Si la bola se mueve hacia la derecha, la rutina es ligeramente distinta a la anterior. Volvemos a cargar en HL la dirección del sprite base, `LD HL, ballRight`, en este caso hacia la derecha, sumamos la rotación en A, `ADD A, C`, multiplicamos por dos, `ADD A, A`, y cargamos el resultado en C, `LD C, A`, para luego sumárselo a HL, `ADD HL, BC`, y así obtenemos la dirección del sprite a imprimir.

Y ahora imprimimos la bola:

```
printBall_continue:
ex    de, hl
ld    hl, (ballPos)
```

Como la rutina NextScan recibe en HL la dirección actual y devuelve, también en HL, la nueva dirección, lo primero es cargar el valor de HL en DE, `EX DE, HL`. Con EX intercambiamos el valor de los registros y ahorramos 4 ciclos de reloj y un byte con respecto de hacerlo con LD (`LD D, H` y `LD E, L`).

Después cargamos la posición de la bola en HL, `LD HL, (ballPos)`.

```
ld    (hl), ZERO
inc   l
ld    (hl), ZERO
dec   l
call  NextScan
```

Pintamos a 0 el primer byte del primer scanline, `LD (HL), ZERO`, pasamos al siguiente byte incrementando la columna, `INC L`, pintamos el segundo byte, `LD (HL), ZERO`, volvemos a dejar la columna como estaba, `DEC L`, y calculamos la dirección del siguiente scanline, `CALL NextScan`.

El siguiente paso es pintar los 4 scanlines que realmente se ven de la bola:

```
ld    b, $04
printBall_loop:
ld    a, (de)
ld    (hl), a
```

```
inc   de
inc   l
ld    a, (de)
ld    (hl), a
dec   de
dec   l
call  NextScan
djnz  printBall_loop
```

Carga en B el número de scanlines que vamos a pintar, `LD B, $04`, carga el primer byte del sprite en A, `LD A, (DE)`, y lo pinta en pantalla, `LD (HL), A`.

Apunta DE al siguiente byte del sprite, `INC DE`, apunta HL a la siguiente columna, `INC L`, carga el sprite en A, `LD A, (DE)`, y lo pinta en pantalla, `LD (HL), A`.

Vuelve a apuntar DE al primer byte del sprite, `DEC DE`, vuelve a apuntar HL a la columna anterior, `DEC L`, y calcula la dirección del scanline siguiente, `CALL NextScan`.

Repite estas operaciones hasta que B valga 0, `DJNZ printBall_loop`.

```
ld    (hl), ZERO
inc   l
ld    (hl), ZERO

ret
```

Pinta el último scanline de la bola en blanco, primero el primer byte, `LD (HL), ZERO`, y tras apuntar HL a la siguiente columna, `INC L`, el segundo, `LD (HL), ZERO`.

El código final de la rutina queda de la siguiente manera:

```
; ------------------------------------------------------------------------------
; Pinta la bola.
; Altera el valor de los registros AF, BC, DE y HL.
; ------------------------------------------------------------------------------
PrintBall:
ld      b, $00                 ; Pone B a 0
ld      a, (ballRotation)      ; Obtiene la rotación de la bola, para averiguar qué pintar
ld      c, a                   ; Carga el valor en C
cp      $00                    ; Compara el valor de la rotación con 0 para ver
                               ; si rota a derecha o izquierda
ld      a, $00                 ; Pone A = 0
jp      p, printBall_right     ; Si es positivo salta, rota a derecha

printBall_left:
```

```
; La rotación de la bola es a izquierda
ld      hl, ballLeft          ; Carga la dirección donde están los bytes de la bola
sub     c                     ; Resta de A el valor de C, rotación de la bola
add     a, a                  ; Suma A + A. Cada definición de la bola son dos bytes
ld      c, a                  ; Carga el valor en C
sbc     hl, bc                ; Resta a HL (dirección de los bytes de la bola)
                              ; el desplazamiento para posicionarse en los correctos
jr      printBall_continue

printBall_right:
; La rotación de la bola es a derecha
ld      hl, ballRight         ; Carga la dirección donde están los bytes de la bola
add     a, c                  ; Suma en A el valor de C, rotación de la bola
add     a, a                  ; Suma A + A. Cada definición de la bola son dos bytes
ld      c, a                  ; Carga el valor en C
add     hl, bc                ; Suma a HL (dirección de los bytes de la bola)
                              ; el desplazamiento para posicionarse en los correctos

printBall_continue:
; Se carga en DE la dirección dónde está la definición de la bola
ex      de, hl
ld      hl, (ballPos)         ; Carga en HL la posición de la bola

; Pinta la primera línea en blanco
ld      (hl), ZERO            ; Mueve blanco a la posición de pantalla
inc     l                     ; Pasa a la siguiente columna
ld      (hl), ZERO            ; Mueve blanco a la posición de pantalla
dec     l                     ; Vuelve a la columna anterior
call    NextScan              ; Pasa al siguiente scanline

ld      b, $04                ; Pinta la definición de la bola en las siguientes 4 líneas
printBall_loop:
ld      a, (de)               ; Carga en A la definición de la bola
ld      (hl), a               ; Carga la definición de la bola a la posición de pantalla
inc     de                    ; Pasa al siguiente byte de la definición de la bola
inc     l                     ; Pasa a la siguiente columna
ld      a, (de)               ; Carga en A la definición de la bola
ld      (hl), a               ; Carga la definición de la bola a la posición de pantalla
dec     de                    ; Vuelve al primer byte de la definición de la bola
dec     l                     ; Vuelve a la columna anterior
call    NextScan              ; Pasa al siguiente scanline
```

```
djnz   printBall_loop        ; Hasta que B = 0


; Pinta la última línea en blanco
ld     (hl), ZERO            ; Mueve blanco a la posición de pantalla
inc    l                     ; Pasa a la siguiente columna
ld     (hl), ZERO            ; Mueve blanco a la posición de pantalla


ret
```

Y ahora ya sólo queda ver si todo lo que hemos implementado funciona, para lo cual vamos a editar el archivo Main.asm:

```
org   $8000


ld    a, $02
out   ($fe), a


ld    a, $00
ld    (ballRotation), a
```

Indicamos la dirección donde cargar el programa, `ORG $8000`, ponemos A = 2, `LD A, $02`, para poner el borde en rojo, `OUT ($FE), A`, y luego ponemos A = 0, `LD A, $00`, para inicializar la rotación de la bola, `LD (ballRotation), A`.

Vamos a implementar un bucle infinito para que la bola se mueva indefinidamente:

```
Loop:
call  PrintBall
```

Lo primero es imprimir la bola, `CALL PrintBall`, en la posición inicial:

```
loop_cont:
ld    b, $08
loopRight:
exx
ld    a, (ballRotation)
inc   a
ld    (ballRotation), a
call  PrintBall
exx
halt
djnz  loopRight
```

En esta primera parte vamos a desplazar, rotar, la bola 8 píxeles hacia la derecha, `LD B, $08`, haciendo un intercambio de valores con los registros alternativos para preservar el valor de B, `EXX`.

EXX intercambia el valor de los registros de propósito común, con el de los registros alternativos:

```
BC ⇄ 'BC
DE ⇄ 'DE
HL ⇄ 'HL
```

Hemos optado en este caso por EXX porque tarda 4 ciclos de reloj y ocupa 1 byte, mientras que PUSH BC tarda 11 ciclos de reloj, y el valor de los registros, exceptuando el de B, no es crítico para ninguna operación que debamos realizar en el bucle, y de paso vemos esta instrucción.

Cargamos en A la rotación actual de la bola, `LD A, (ballRotation)`, incrementamos la rotación, `INC A`, y cargamos el valor resultante en memoria, `LD (ballRotation), A`.

Pintamos la bola, `CALL PrintBall`, volvemos a intercambiar el valor de los registros, `EXX`, para recuperar el valor de B y hacemos una pausa para poder ver cómo se mueve la bola, `HALT`.

Repetimos hasta que B valga 0, `DJNZ loopRight`.

Volvemos a poner a 0 la rotación de la bola, pero esta vez sin pintarla, para empezar a rotar los píxeles hacia la izquierda (ver definición del sprite de la bola):

```
ld     a, $00
ld     (ballRotation), a
```

Ahora vamos a desplazar, rotar, la bola 8 píxeles hacia la izquierda. Solo cambian una instrucción y una etiqueta respecto al desplazamiento hacia la derecha, por lo que no se explica la rutina, simplemente se marca la instrucción que cambia en rojo, para que se vea la diferencia:

```
ld     b, $08
loopLeft:
exx
ld     a, (ballRotation)
dec    a
ld     (ballRotation), a
call   PrintBall
exx
halt
djnz   loopLeft
```

Para terminar, volvemos a poner la rotación a 0, cargamos el valor en memoria y volvemos a repetir el bucle:

```
ld    a, $00
ld    (ballRotation), a

jr    loop_cont
```

Sin olvidarnos de incluir los ficheros Sprite.asm y Video.asm, e indicarle a PASMO dónde tiene que llamar al cargar el programa:

```
include "Sprite.asm"
include "Video.asm"
end   $8000
```

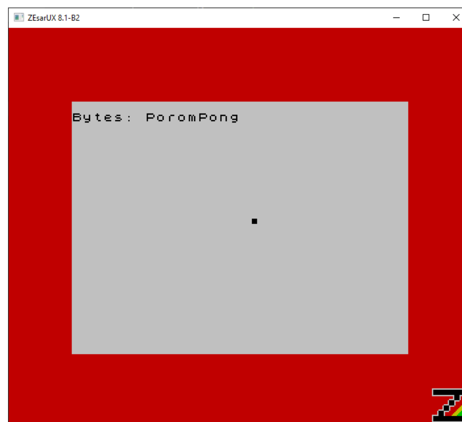
En realidad, la bola no se mueve, muy al contrario, lo que hacemos es pintarla siempre en las mismas dos columnas, desplazando los píxeles 8 veces hacia la derecha y luego 8 veces hacia la izquierda, para volver a empezar una y otra vez.

El aspecto final del archivo Main.asm es el siguiente:

```
; Mueve la bola de izquierda a derecha entre dos columnas.
org     $8000

ld      a, $02              ; A = 2
out     ($fe), a            ; Pone el borde en rojo
ld      a, $00              ; A = 0
ld      (ballRotation), a   ; Pone la rotación de la bola a 0

Loop:
call    PrintBall           ; Imprime la bola

loop_cont:
ld      b, $08              ; Mueve la bola 8 píxeles a la derecha
loopRight:
exx                         ; Intercambia el valor de los registros para preservar B
ld      a, (ballRotation)   ; Recupera la rotación de la bola
inc     a                   ; Incrementa la rotación
ld      (ballRotation), a   ; Guarda el valor de la rotación
call    PrintBall           ; Imprime la bola
exx                         ; Intercambia el valor de los registros para recuperar B
halt                        ; Se sincroniza con el refresco de la pantalla
djnz    loopRight           ; Hasta que B = 0

ld      a, $00              ; A = 0
```

```
ld      (ballRotation), a     ; Pone la rotación de la bola a 0

ld      b, $08                ; Mueve la bola 8 píxeles a la derecha
loopLeft:
exx                           ; Intercambia el valor de los registros para preservar B
ld      a, (ballRotation)     ; Recupera la rotación de la bola
dec     a                     ; Decrementa la rotación
ld      (ballRotation), a     ; Guarda el valor de la rotación
call    PrintBall             ; Imprime la bola
exx                           ; Intercambia el valor de los registros para recuperar B
halt                          ; Se sincroniza con el refresco de la pantalla
djnz    loopLeft              ; Hasta que B = 0

ld      a, $00                ; A = 0
ld      (ballRotation), a     ; Pone la rotación de la bola a 0

jr      loop_cont             ; Bucle infinito

include "Sprite.asm"
include "Video.asm"

end     $8000
```

Ya solo queda compilar y ver los resultados en el emulador:



Paso 5: movemos la bola por la pantalla

Creamos la carpeta Paso05, dentro de la misma creamos los archivos Main.asm y Game.asm, y copiamos los archivos Sprite.asm y Video.asm que tenemos en la carpeta Paso04.

Empezamos editando Sprite.asm para añadir dos nuevas contantes que vamos a necesitar para mover la bola por la pantalla:

```
MARGIN_LEFT:       EQU    $00
MARGIN_RIGHT:      EQU    $1e
```

Igual que tenemos los límites verticales y horizontales, necesitamos los límites derecho e izquierdo para que la bola se mantenga dentro de los mismos.

El siguiente paso es implementar la lógica del movimiento de la bola, lo que haremos en Game.asm:

```
MoveBall:
ld     a, (ballSetting)
and    $80
jr     nz, moveBall_down
```

Primero cargamos en A la configuración actual de la bola, `LD A, (ballSetting)`, y nos quedamos con el bit 7, `AND $80`, que indica si la bola se desplaza hacia arriba o hacia abajo. Si el bit no está a 0, la bola se desplaza hacia abajo y salta, `JR NZ, moveBall_down`.

Si el bit está a 0, la bola se desplaza hacia arriba:

```
moveBall_up:
ld     hl, (ballPos)
ld     a, BALL_TOP
call   CheckTop
jr     z, moveBall_upChg
call   PreviousScan
ld     (ballPos), hl
jr     moveBall_x
```

Cargamos la posición actual de la bola en HL, `LD HL, (ballPos)`, el límite vertical en A, `LD A, BALL_TOP`, y comprobamos si se ha alcanzado dicho límite, `CALL CheckTop`. Si se activa el flag Z, se ha alcanzado el límite y salta para cambiar la dirección vertical de la bola, `JR Z, moveBall_upChg`.

Si la bola no ha llegado al límite vertical, calcula la nueva posición, `CALL PreviousScan`, la carga en memoria, `LD (ballPos), HL`, y salta a comprobar el desplazamiento horizontal, `JR moveBall_x`.

En el caso de haber alcanzado el límite superior, hay que cambiar la dirección vertical de la bola:

```
moveBall_upChg:
ld    a, (ballSetting)
or    $80
ld    (ballSetting), a
call  NextScan
ld    (ballPos), hl
jr    moveBall_x
```

Primero cargamos la configuración de la bola en A, `LD A, (ballSetting)`, luego activamos el bit 7, `OR $80`, para indicar que ahora la bola debe ir hacia abajo, y cargamos el valor en memoria, `LD (ballSetting), A`. Calculamos la nueva posición vertical de la bola, `CALL NextScan`, cargamos el valor en memoria, `LD (ballPos), HL`, y saltamos a comprobar el desplazamiento horizontal, `JR moveBall_x`.

Para activar el bit 7 hemos hecho un OR con $80 (10000000). Es conveniente recordar el resultado de la operación OR, dependiendo del valor de los bits:

Bit 1	Bit 2	Resultado
0	0	0
1	0	1
0	1	1
1	1	1

Según se ve en la tabla, al aplicar OR $80, pone el bit 7 a 1 y el resto los deja como estaban.

Si al iniciar la rutina la bola iba hacia abajo, hay que hacer algo parecido a lo visto anteriormente:

```
moveBall_down:
ld    hl, (ballPos)
ld    a, BALL_BOTTOM
call  CheckBottom
jr    z, moveBall_downChg
call  NextScan
ld    (ballPos), hl
jr    moveBall_x
```

Primero cargamos la posición de la bola en HL, `LD HL, (ballPos)`, el límite inferior en A, `LD A, BALL_BOTTOM`, y comprobamos si se ha alcanzado, `CALL CheckBottom`, en cuyo caso saltamos para cambiar la dirección de la bola, `JR Z, moveBall_downChg`.

Si no se ha alcanzado el límite inferior, calculamos la nueva posición de la bola, `CALL NextScan`, la cargamos en memoria, `LD (ballPos), HL`, y saltamos a comprobar el desplazamiento horizontal, `JR moveBall_x`.

En el caso de haber alcanzado el límite inferior, hay que cambiar la dirección vertical de la bola:

```
moveBall_downChg:
```

```
ld    a, (ballSetting)
and   $7f
ld    (ballSetting), a
call  PreviousScan
ld    (ballPos), hl
```

Primero cargamos la configuración de la bola en A, `LD A, (ballSetting)`, luego desactivamos el bit 7, `AND $7F`, para indicar que ahora la bola debe ir hacia arriba, y cargamos el valor en memoria, `LD (ballSetting), A`. Calculamos la nueva posición vertical de la bola, `CALL PreviousScan`, y cargamos el valor en memoria, `LD (ballPos), HL`.

Para desactivar el bit 7 hemos hecho un AND con $7F (01111111). Es conveniente recordar el resultado de la operación AND, dependiendo del valor de los bits:

Bit 1	Bit 2	Resultado
0	0	0
1	0	0
0	1	0
1	1	1

Según se ve en la tabla, al aplicar AND $7F, pone el bit 7 a 0 y el resto los deja como estaban.

Empezamos a calcular el desplazamiento horizontal:

```
moveBall_x:
ld    a, (ballSetting)
and   $40
jr    nz, moveBall_left
```

Cargamos la configuración de la bola en A, `LD A, (ballSetting)`, comprobamos el estado del bit 6, `AND $40`, y si no está a 0, la bola va hacia la izquierda y salta, `JR NZ, moveBall_left`.

Si el bit 6 está a 0, la bola va hacia la derecha:

```
moveBall_right:
ld    a, (ballRotation)
cp    $08
jr    z, moveBall_rightLast
inc   a
ld    (ballRotation), a
jr    moveBall_end
```

Cargamos la rotación en A, `LD A, (ballRotation)`, y comprobamos si está en la última, `CP $08`, en cuyo caso saltamos, `JR Z, movelBall_rightLast`.

Si no está en la última rotación, incrementamos la misma, `INC A`, la cargamos en memoria, `LD (ballRotation), A`, y saltamos al final de la rutina, `JR moveBall_end`.

Si, por el contrario, ha llegado a la última rotación y no está en el límite derecho, desplazamos la bola a la siguiente columna:

```
moveBall_rightLast:
ld    a, (ballPos)
and   $1f
cp    MARGIN_RIGHT
jr    z, moveBall_rightChg
ld    hl, ballPos
inc   (hl)
ld    a, $01
ld    (ballRotation), a
jr    moveBall_end
```

Cargamos la línea y columna en A, `LD A, (ballPos)`, nos quedamos con la columna, `AND $1F`, y evaluamos si ha llegado al límite derecho, `CP MARGIN_RIGHT`, en cuyo caso saltamos para cambiar la dirección de la bola, `JR Z, moveBall_rightChg`.

Si no se ha llegado al límite derecho, desplazamos la bola a la siguiente columna. Cargamos la dirección donde se encuentra la posición de la bola en HL, `LD HL, ballPos`, e incrementamos la columna, `INC (HL)`.

Ponemos la rotación de la bola a 1, `LD A, $01`, lo cargamos en memoria, `LD (ballRotation), A`, y saltamos al fin de la rutina, `JR moveBall_end`.

Como se puede ver, para cargar la columna en A, la instrucción usada ha sido `LD A, (ballPos)`, y para incrementar la columna `LD HL, ballPos` y `INC (HL)`.

Teniendo en cuenta que las posiciones de memoria de la VideoRAM se codifican 010TTSSS LLLCCCCC, ¿no estaríamos cargando y alterando el scanline? No, y ello se debe a que el Z80 es un micro de tipo Little Endian.

Un micro Little Endian, cuando carga valores de 16 bits en memoria, carga en la primera posición de memoria el byte menos significativo, y en la siguiente el más significativo, de tal manera que si en la posición de memoria $C000 se carga el valor $4000, en la posición $C000 se carga $00 y en la $C001 se carga $40. Es por eso que cuando se carga en A el valor desde (ballPos), lo que carga es el byte menos significativo que es donde están la línea y columna. De igual modo al incrementar (HL), incrementa la columna.

Si la carga se hace sobre un registro de 16 bits, carga el byte menos significativo en la parte baja del registro, y el más significativo en la parte alta. Es por eso que al cargar ballPos en HL, carga en H el byte más significativo de la dirección de memoria y en L el menos significativo.

Seguimos con la rutina…

Si ha llegado al límite derecho, hay que cambiar la dirección de la bola:

```
moveBall_rightChg:
ld    a, (ballSetting)
```

```
or    $40
ld    (ballSetting), a
ld    a, $ff
ld    (ballRotation), a
jr    moveBall_end
```

Cargamos la configuración de la bola en A, `LD A, (ballSetting)`, activamos el bit 6 para cambiar la dirección hacia la izquierda, `OR $40`, y cargamos el valor en memoria, `LD (ballSetting), A`.

Ponemos la rotación de la bola a -1, `LD A, $FF`, la cargamos en memoria, `LD (ballRotation), A`, y saltamos al fin de la rutina, `JR moveBall_end`.

Si la bola va hacia la izquierda, hay que hacer algo parecido a lo visto anteriormente:

```
moveBall_left:
ld    a, (ballRotation)
cp    $f8
jr    z, moveBall_leftLast
dec   a
ld    (ballRotation), a
jr    moveBall_end
```

Cargamos la rotación en A, `LD A, (ballRotation)`, comprobamos si está en la última, `CP $F8`, y de ser así saltamos, `JR Z, moveBall_leftLast`.

Si no está en la última rotación la decrementamos, `DEC A`, cargamos el valor en memoria, `LD (ballRotation), A`, y saltamos al fin de la rutina, `JR moveBall_end`.

Si ha llegado a la última rotación y no ha alcanzado el límite izquierdo, desplazamos la bola a la columna anterior:

```
moveBall_leftLast:
ld    a, (ballPos)
and   $1f
cp    MARGIN_LEFT
jr    z, moveBall_leftChg
ld    hl, ballPos
dec   (hl)
ld    a, $ff
ld    (ballRotation), a
jr    moveBall_end
```

Cargamos la línea y columna en A, `LD A, (ballPos)`, nos quedamos con la columna, `AND $1F`, y comprobamos si ha llegado al límite izquierdo, `CP MARGIN_LEFT`, en cuyo caso saltamos, `JR Z, moveBall_leftChg`.

Si no ha llegado al límite izquierdo, cargamos la dirección donde está la posición de la bola en HL, `LD HL, ballPos`, y decrementamos la columna, `DEC (HL)`.

Ponemos la rotación de la bola a -1, `LD A, $FF`, cargamos el valor en memoria, `LD (ballRotation), A`, y saltamos al fin de la rutina.

Terminamos la rutina con el cambio de dirección, si se ha alcanzado el límite izquierdo:

```
moveBall_leftChg:
ld    a, $01
ld    (ballRotation), a
ld    a, (ballSetting)
and   $bf
ld    (ballSetting), a

moveBall_end:
ret
```

Ponemos la rotación de la bola a 1, `LD A, $01`, y la cargamos en memoria, `LD (ballRotation), A`. Cargamos la configuración de la bola en A, `LD A, (ballSetting)`, desactivamos el bit 6 para que la dirección sea hacia la derecha, `AND $BF`, y cargamos el valor en memoria, `LD (ballSetting), A`.

Podemos ahorrar 2 ciclos de reloj y 5 bytes haciendo una pequeña modificación. Lo dejamos en vuestras manos y veremos la forma de hacerlo en el paso 10.

El aspecto final de la rutina es el siguiente:

```
; ---------------------------------------------------------------------------
; Calcula la posición, rotación y dirección de la bola para pintarla.
; Altera el valor de los registros AF y HL.
; ---------------------------------------------------------------------------
MoveBall:
ld      a, (ballSetting)      ; Carga en A la dirección y velocidad de la bola
and     $80                   ; Comprueba la dirección vertical
jr      nz, moveBall_down     ; Si el bit 7 está a uno, va hacia abajo

moveBall_up:
; La bola va hacia arriba
ld      hl, (ballPos)         ; Carga la posición de la bola en HL
ld      a, BALL_TOP           ; Carga en A el margen superior
call    CheckTop              ; Evalúa si se ha alcanzado el margen superior
```

```
jr      z, moveBall_upChg     ; Si se ha alcanzado salta
call    PreviousScan          ; Obtiene el scanline anterior a la posición de la bola
ld      (ballPos), hl         ; Carga en memoria la nueva posición de la bola
jr      moveBall_x            ; Salta

moveBall_upChg:
; La bola va hacia arriba, pero ha llegado al tope y cambia de dirección
ld      a, (ballSetting)      ; Carga en A la dirección y velocidad de la bola
or      $80                   ; Pone la dirección vertical hacia abajo
ld      (ballSetting), a      ; Carga en memoria la nueva dirección de la bola
call    NextScan              ; Obtiene el scanline siguiente a la posición de la bola
ld      (ballPos), hl         ; Carga en memoria la nueva posición de la bola
jr      moveBall_x            ; Salta

moveBall_down:
; La bola va hacia abajo
ld      hl, (ballPos)         ; Carga la posición de la bola en HL
ld      a, BALL_BOTTOM        ; Carga en A el margen superior
call    CheckBottom           ; Evalúa si se ha alcanzado el margen superior
jr      z, moveBall_downChg   ; Si se ha alcanzado salta
call    NextScan              ; Obtiene el scanline siguiente a la posición de la bola
ld      (ballPos), hl         ; Carga en memoria la nueva posición de la bola
jr      moveBall_x            ; Salta

moveBall_downChg:
; La bola va hacia abajo, pero ha llegado al tope y cambia de dirección
ld      a, (ballSetting)      ; Carga en A la dirección y velocidad de la bola
and     $7f                   ; Pone la dirección vertical hacia arriba
ld      (ballSetting), a      ; Carga en memoria la nueva dirección de la bola
call    PreviousScan          ; Obtiene el scanline anterior a la posición de la bola
ld      (ballPos), hl         ; Carga en memoria la nueva posición de la bola

moveBall_x:
ld      a, (ballSetting)      ; Carga en A la dirección y velocidad de la bola
and     $40                   ; Comprueba la dirección horizontal
jr      nz, moveBall_left     ; Si el bit 6 está a uno, va hacia la izquierda

moveBall_right:
; La bola va hacia la derecha
ld      a, (ballRotation)     ; Carga la rotación actual de la bola
cp      $08                   ; Comprueba si ya está en la última rotación
```

```
jr      z, moveBall_rightLast ; Si está en la última rotación salta
inc     a                     ; Incrementa la rotación
ld      (ballRotation), a     ; La carga en memoria
jr      moveBall_end          ; Fin de la rutina

moveBall_rightLast:
; Está en la última rotación
; Si no ha llegado al límite derecho pone la rotación a 1
; y pone la bola en la siguiente columna
ld      a, (ballPos)          ; Carga la línea y columna de la bola en A
and     $1f                   ; Se queda solo con la columna
cp      MARGIN_RIGHT          ; Lo comprara con el límite derecho
jr      z, moveBall_rightChg  ; Si lo ha alcanzado salta
ld      hl, ballPos           ; Carga la dirección de la posición de la bola en HL
inc     (hl)                  ; Incrementa la columna
ld      a, $01                ; Pone la rotación a 1
ld      (ballRotation), a     ; Carga el valor en memoria
jr      moveBall_end          ; Fin de la rutina

moveBall_rightChg:
; Ha llegado al límite derecho
; Pone la rotación a -1 y cambia la dirección horizontal de la bola
ld      a, (ballSetting)      ; Carga en A la dirección y velocidad de la bola
or      $40                   ; Pone la dirección horizontal hacia la izquierda
ld      (ballSetting), a      ; Carga la nueva dirección de la bola en memoria
ld      a, $ff                ; Carga -1 en A
ld      (ballRotation), a     ; Lo carga en memoria Rotación = -1
jr      moveBall_end          ; Fin de la rutina

moveBall_left:
; La bola va hacia la izquierda
ld      a, (ballRotation)     ; Carga la rotación actual de la bola
cp      $f8                   ; Comprueba si ya está en la última rotación
jr      z, moveBall_leftLast  ; Si está en la última rotación salta
dec     a                     ; Decrementa la rotación
ld      (ballRotation), a     ; La carga en memoria
jr      moveBall_end          ; Fin de la rutina

moveBall_leftLast:
; Esta en la última rotación
; Si no ha llegado al límite izquierdo pone la rotación a -1
```

```
; y pone la bola en la columna anterior
ld      a, (ballPos)          ; Carga la línea y columna en A
and     $1f                   ; Se queda solo con la columna
cp      MARGIN_LEFT           ; Lo comprara con el límite izquierdo
jr      z, moveBall_leftChg   ; Si lo ha alcanzado salta
ld      hl, ballPos           ; Carga la dirección de la posición de la bola en HL
dec     (hl)                  ; Pasa a la columna anterior
ld      a, $ff                ; Pone la rotación a -1
ld      (ballRotation), a     ; Carga el valor en memoria
jr      moveBall_end          ; Fin de la rutina

moveBall_leftChg:
; Ha llegado al límite izquierdo
; Pone la rotación a 1 y cambia la dirección
ld      a, $01                ; Carga la posición de la bola en HL
ld      (ballRotation), a     ; Carga el valor en memoria Rotación = 1
ld      a, (ballSetting)      ; Carga en A la dirección y velocidad de la bola
and     $bf                   ; Pone la dirección horizontal hacia la derecha
ld      (ballSetting), a      ; Carga la nueva dirección de la bola en memoria

moveBall_end:
ret
```

Ha llegado el momento de probar todo lo implementado; vamos a editar el archivo Main.asm. En este caso la implementación es muy sencilla:

```
org   $8000
ld    a, $02
out   ($fe), a
call  PrintBall
```

Indicamos la dirección dónde cargar el programa, ponemos el borde en rojo y pintamos la bola en la posición inicial.

```
Loop:
call  MoveBall
call  PrintBall
halt
jr    Loop
```

Implementamos un bucle infinito en el que movemos la bola, la pintamos, esperamos al refresco de la pantalla y volvemos a realizar estas tres operaciones indefinidamente.

```
Include "Game.asm"
```

```
Include "Sprite.asm"
Include "Video.asm"
end     $8000
```

Por último, incluimos los archivos necesarios e indicamos a PASMO dónde tiene que llamar al cargar el programa.

El aspecto final de Main.asm es el siguiente:

```
; Mueve la bola por la pantalla trazando diagonales
org     $8000

ld      a, $02  ; A = 2
out     ($fe), a        ; Pone el borde en rojo
call    PrintBall       ; Imprime la bola

Loop:
call    MoveBall        ; Mueve la bola
call    PrintBall       ; Pinta la bola
halt                    ; Espera al refresco de pantalla
jr      Loop            ; Bucle infinito

include "Game.asm"
include "Sprite.asm"
include "Video.asm"

end     $8000
```

Llega el gran momento…compilamos y vemos el resultado en el emulador:

Paso 6: campo, palas, bola y temporización

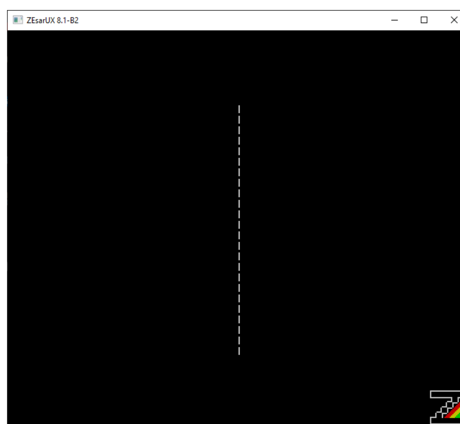
Creamos la carpeta Paso06 y copiamos desde la carpeta Paso05 los archivos Game.asm, Sprite.asm y Video.asm, y desde la carpeta Paso03 el archivo Controls.asm. También creamos el archivo Main.asm.

Empezamos editando el archivo Main.asm, indicando la posición de carga, poniendo el borde en negro, limpiando la pantalla, pintando la línea central y haciendo un bucle infinito para no volver al Basic.

También vamos a incluir el resto de ficheros e indicarle a PASMO dónde llamar al cargar el programa:

```
org   $8000

Main:
ld    a, $00
out   ($fe), a
call  Cls
call  PrintLine

Loop:
jr    Loop

include "Controls.asm"
include "Game.asm"
include "Sprite.asm"
include "Video.asm"

end   $8000
```

Compilamos y vemos el resultado en el emulador.



El siguiente paso es pintar el borde del campo.

Incluimos al inicio del fichero Sprite.asm una nueva constante:

```
FILL: EQU   $ff
```

La rutina para pintar el borde la implementamos en el archivo Video.asm, antes de la rutina PrintLine:

```
PrintBorder:
ld    hl, $4100
ld    de, $56e0
ld    b, $20
ld    a, FILL
```

Cargamos en HL la dirección del tercio 0, línea 0, scanline 1, `LD HL, $4100`, carga en DE la dirección del tercio 2, línea 7, scanline 6, `LD DE, $56E0`, y en B las 32 columnas en las que pintar el borde, `LD B, $20`. Por último, cargamos el sprite del borde en A, `LD A, FILL`.

Implementamos el bucle para pintar el borde:

```
printBorder_loop:
ld    (hl), a
ld    (de), a
inc   l
inc   e
djnz  printBorder_loop
ret
```

Pintamos el sprite del borde en la dirección a la que apunta HL, `LD (HL), A`, y hacemos lo mismo con la dirección a la que apunta DE, `LD (DE), A`.

Apuntamos HL a la siguiente columna, `INC L`, y hacemos lo mismo con DE, `INC E`. Repetimos hasta que B valga 0, `DJNZ printBorder_loop`, tras lo cual salimos de la rutina, `RET`.

El aspecto final de la rutina PrintBorder es el siguiente:

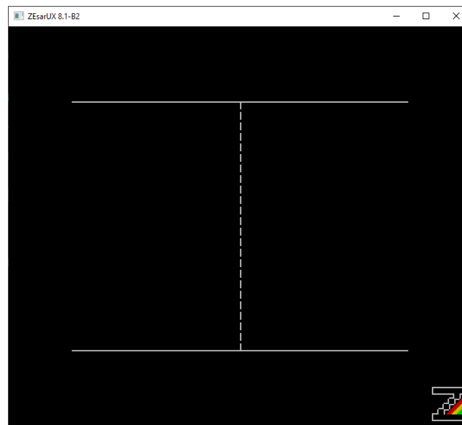
```
; -----------------------------------------------------------------------------
; Pinta el borde del campo.
; Altera el valor de los registros AD, B, DE y HL.
; -----------------------------------------------------------------------------
PrintBorder:
ld      hl, $4100     ; Carga en HL la dirección del tercio 0, línea 0 y scanline 1
ld      de, $56e0     ; Carga en DE la dirección del tercio 2, línea 7 y scanline 6
ld      b, $20        ; Carga en B las 32 columnas en las que pintar
ld      a, FILL       ; Carga en A el byte a pintar

printBorder_loop:
```

```
ld      (hl), a        ; Pinta en la dirección apuntada por HL
ld      (de), a        ; Pinta en la dirección apuntada por DE
inc     l              ; Apunta HL a la siguiente columna
inc     e              ; Apunta DE a la siguiente columna
djnz    printBorder_loop      ; Bucle hasta que B llegue a 0
ret
```

Para probar esta rutina, volvemos al archivo Main.asm y tras la llamada a PrintLine ponemos la llamada a la nueva rutina:

```
call          PrintBorder
```

Compilamos y vemos los resultados en el emulador.

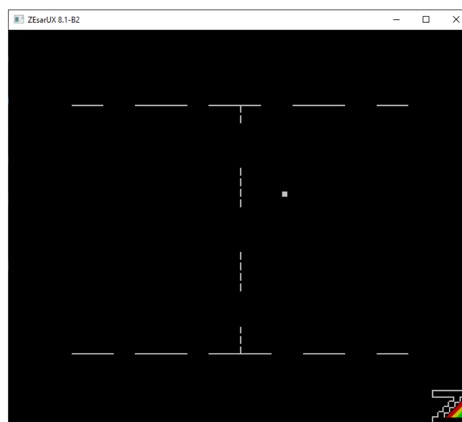


Ya tenemos dibujado el campo donde se va a desarrollar la acción.

Vamos a introducir la bola en nuestro campo. Como la vamos a estar moviendo y pintando constantemente, vamos a introducir las llamadas a mover y pintar la bola dentro del bucle, entre Loop y JR Loop:

```
call  MoveBall
call  PrintBall
```

Compilamos y vemos los resultados en el emulador:



Al ver el resultado, observamos dos problemas: la bola borra la línea central y el borde, y se mueve a una velocidad endiablada.

Lo primero que vamos a abordar es la velocidad a la que se mueve la bola.

En el paso anterior poníamos un HALT para esperar el refresco de la pantalla, pero esto hace que vaya demasiado lenta. Para reducir la velocidad de la bola, vamos a hacer que no se mueva cada vez que pase por el bucle; se va a mover una de cada N veces.

Seguimos en el archivo Main.asm y, antes de END $8000, declaramos la variable donde vamos a llevar la cuenta de las veces que se ha pasado por el bucle:

```
countLoopBall:    db    $00
```

Y ahora vamos a implementar la parte en la que comprueba si ha pasado las veces suficientes para que movamos la bola, justo después de la etiqueta Loop:

```
ld    a, (countLoopBall)
inc   a
ld    (countLoopBall), a
cp    $0f
jr    nz, loop_continue
call  MoveBall
```

Cargamos el contador donde guardamos las veces que se ha pasado por el bucle sin mover la bola en A, `LD A, (countLoopBall)`, lo incrementamos, `INC A`, y lo guardamos en memoria, `LD (countLoopBall), A`.

Comparamos si el contador ha llegado al número de veces necesarias para mover la bola, `CP $0F`, y si no ha llegado salta, `JR NZ, loop_continue`.

Si ya hemos llegado al número de veces necesarias de pasadas por el bucle para mover la bola, la movemos, `CALL MoveBall`.

La etiqueta `loop_continue` es nueva y la vamos a poner justo encima de la llamada a PrintBall:

```
loop_continue:
call  PrintBall
```

Tenemos que hacer una última cosa. Si el contador ha llegado al número de veces necesario para mover la bola, después de moverla hay que volver a poner el contador a cero, de lo contrario habría que esperar otras 255 veces, en lugar de las que hemos puesto.

Añadimos las siguientes líneas después de la llamada a MoveBall y antes de la etiqueta loop_continue:

```
ld    a, ZERO
ld    (countLoopBall), a
```

La implementación de Main.asm quedaría así:

```
org     $8000
Main:
ld      a, $00
```

```
out     ($fe), a              ; Pone el borde en negro

call    Cls                   ; Limpia la pantalla
call    PrintLine             ; Pinta la línea central
call    PrintBorder           ; Pinta el borde

Loop:
ld      a, (countLoopBall)    ; Carga en A el contador de la bola
inc     a                     ; Incrementa el contador
ld      (countLoopBall), a    ; Carga el valor en memoria
cp      $0f                   ; Comprueba si el contador ha llegado a 15
jr      nz, loop_continue     ; Si no ha llegado, salta
call    MoveBall              ; Mueve la posición de la bola
ld      a, ZERO               ; Pone A = 0
ld      (countLoopBall), a    ; Carga el valor en memoria

loop_continue:
call    PrintBall             ; Pinta la bola
jr      Loop                  ; Bucle infinito

include "Game.asm"
include "Controls.asm"
include "Sprite.asm"
include "Video.asm"

countLoopBall: db $00         ; Contador para controlar cuando se mueve la bola
end     $8000
```

Compilamos y vemos el resultado en el emulador. Esta vez sí vemos como se mueve la bola a una velocidad más aceptable.

No hemos definido una constante para la comparación con el contador de la bola ya que, en un futuro, la velocidad será variable.

Ahora vamos a abordar el problema de las partes que va borrando la bola a su paso, y vamos a empezar por la línea central.

En una primera aproximación, vamos a repintar la parte de la línea que coincide en la coordenada Y con la bola, sin importar si la bola está pasando por encima o no. Parece innecesario, pero nos va a ayudar a temporizar.

Abrimos el archivo Video.asm e implementamos después de la rutina PrintLine:

```
ReprintLine:
ld      hl, (ballPos)
```

```
ld    a, l
and   $e0
or    $10
ld    l, a
```

Cargamos la posición de la bola en HL, `LD HL, (ballPos)`, cargamos la línea y columna en A, `LD A, L`, nos quedamos con el valor de la línea, `AND $E0`, ponemos la columna a 16, que es donde está la línea vertical, `OR $10`, y cargamos el valor en L, `LD L, A`.

Vamos a repintar 6 scanlines, que son los mismos que tiene la bola:

```
ld          b, $06
reprintLine_loop:
ld          a, h
```

Cargamos en B el número de scanlines que se repintan, `LD B, $06`, y cargamos tercio y scanline en A, `LD A, H`.

Para pintar la línea, en los scanlines 0 y 7 pintábamos en blanco, y en el resto la parte visible de la línea:

```
and         $07
cp          $01
jr          c, reprintLine_00
cp          $07
jr          z, reprintLine_00
```

Nos quedamos con la parte del scanline, `AND $07`, y comprobamos si es 1, `CP $01`. Si el scanline es menor que 1 saltamos, `JR C, reprintLine_00`, en el caso contrario comprobamos si es 7, `CP $07`. Si el scanline es 7 saltamos, `JR Z, reprintLine_00`.

Si no hemos saltado, el scanline está entre 1 y 6:

```
ld    c, LINE
jr    reprintLine_loopCont
```

Cargamos el sprite de la línea en C, `LD C, LINE`, y saltamos, `JR reprintLine_loopCont`.

Si anteriormente saltamos, el scanline es 0 o 7:

```
reprintLine_00:
ld    c, ZERO
```

Cargamos el sprite en blanco en C, `LD C, ZERO`, y pintamos lo que corresponda:

```
reprintLine_loopCont:
ld    a, (hl)
or    c
ld    (hl), a
```

```
call  NextScan
djnz  reprintLine_loop

ret
```

Cargamos el valor de la dirección de memoria del byte que vamos a repintar en A, `LD A, (HL)`, le añadimos los píxeles del repintado de la línea, `OR C`, y lo pintamos en pantalla, `LD (HL), A`. Calculamos la dirección de memoria del scanline siguiente, `CALL NextScan`, y repetimos la operación hasta que B valga 0, `DJNZ reprintLine_loop`. Por último, salimos, `RET`.

El aspecto final de la rutina es el siguiente:

```
; -------------------------------------------------------------------------------
; Repinta la línea central.
; Altera el valor de los registros AF, BC y HL.
; -------------------------------------------------------------------------------
ReprintLine:
ld      hl, (ballPos)          ; Carga en HL la posición de la bola
ld      a, l                   ; Carga la línea y columna en A
and     $e0                    ; Se queda con la línea
or      $10                    ; Pone la columna a 16 ($10)
ld      l, a                   ; Carga el valor en L. HL = Posición inicial repintar

ld      b, $06                 ; Se repintan 6 scanlines
reprintLine_loop:
ld      a, h                   ; Carga tercio y scanline en A
and     $07                    ; Se queda con el scanline
; Si está en los scanline 0 o 7 pinta ZERO
; Si está en los scanline 1, 2, 3, 4, 5 o 6 pinta LINE
cp      $01                    ; Comprueba si está en scanline 1 o superior
jr      c, reprintLine_00      ; Si está por debajo, pinta $00
cp      $07                    ; Comprueba si está en scanline 7
jr      z, reprintLine_00      ; Si es así, pinta ZERO

ld      c, LINE                ; Esta en scanline 1 a 6, pinta LINE
jr      reprintLine_loopCont   ; Salta
reprintLine_00:
ld      c, ZERO                ; Está en scanline 0 o 7, pinta ZERO
reprintLine_loopCont:
ld      a, (hl)                ; Obtiene los pixeles de la posición actual
or      c                      ; Los mezcla con C
ld      (hl), a                ; Pinta el resultado en la posición actual
```

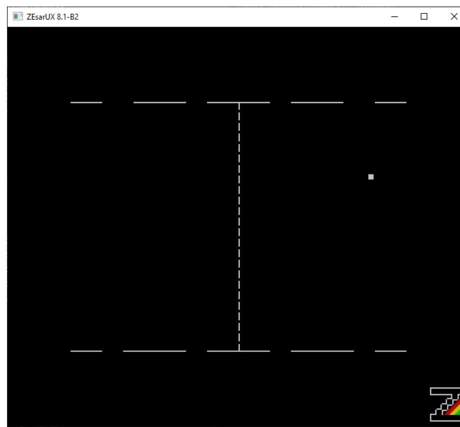
```
call    NextScan               ; Obtiene el scanline siguiente
djnz    reprintLine_loop       ; Hasta que B = 0


ret
```

Podemos ahorrar 4 bytes y 19 ciclos de reloj modificando seis líneas de la rutina. Lo dejamos en vuestras manos y veremos la forma de hacerlo en el paso 10.

Ya solo queda probar lo que hemos implementado, para lo cual abrimos el archivo Main.asm y después de la llamada a PrintBall incluimos la llamada a ReprintLine:

```
call  ReprintLine
```

Compilamos y vemos los resultados en el emulador:



La línea central ya no se borra, pero podemos apreciar que la velocidad de la bola ha disminuido. Hay que tener en cuenta que ahora realizamos más operaciones que antes. Según avancemos iremos ajustando la velocidad de la bola.

Vamos ahora a evitar que se borre el borde, para lo cual vamos a modificar los límites superior e inferior de la bola, en el fichero Sprite.asm:

```
BALL_BOTTOM:        EQU    $b8
BALL_TOP:           EQU    $02
```

Compilamos, cargamos en el emulador y comprobamos que ya no se borra el borde.

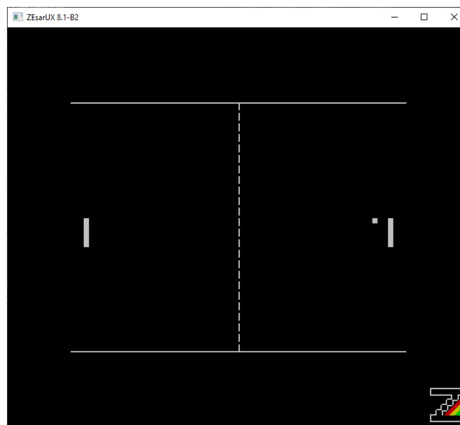
Ahora vamos a empezar con las palas. Volvemos a Main.asm y añadimos las siguientes líneas entre `CALL ReprintLine` y `JR Loop`:

```
ld     hl, (paddle1pos)
call   PrintPaddle
ld     hl, (paddle2pos)
call   PrintPaddle
```

Cargamos la posición de la pala 1 en HL, `LD HL, (paddle1pos)`, y la pintamos, `CALL PrintPaddle`. Hacemos lo mismo con la pala 2.

Como se puede apreciar, las palas se pintan en todas las iteraciones del bucle, al igual que la bola y el repintado de línea.

Compilamos y vemos los resultados en el emulador:



Se dibujan las palas y la bola no las borra al pasar. También se aprecia que ahora la bola va mucho más lenta, debido a que hacemos más operaciones en cada iteración del bucle.

Para que la bola vuelva a ir más rápida, vamos a cambiar en Main.asm el valor que tenía que alcanzar el contador para que la bola se moviese:

```
ld            (countLoopBall), a
cp            $06
jr            nz, loop_continue
```

Compilamos, cargamos en el emulador y comprobamos que la bola vuelve a ir más rápido.

Ahora vamos a implementar la rutina para mover las palas; ya vimos cómo hacerlo en el paso 03. Editamos el archivo Game.asm y vamos al final del mismo.

La rutina que vamos a implementar, recibe en el registro D las pulsaciones de las teclas de control:

```
MovePaddle:
bit   $00, d
jr    z, movePaddle_1Down
```

Evaluamos si se ha pulsado la tecla arriba del jugador 1, `BIT $00, D`. Si no se ha pulsado saltamos a comprobar si se ha pulsado la tecla abajo, `JR Z, movePaddle_1Down`.

Si no salta, se ha pulsado la tecla arriba del jugador 1:

```
ld    hl, (paddle1pos)
ld    a, PADDLE_TOP
call  CheckTop
jr    z, movePaddle_2Up
```

Cargamos la posición de la pala 1 en HL, `LD HL, (paddle1pos)`, el límite superior para las palas en A, `LD A, PADDLE_TOP`, y comprobamos si se ha alcanzado, `CALL CheckTop`. Si se ha alcanzado el límite, saltamos a comprobar los controles del jugador 2, `JR Z, movePaddle_2Up`.

Si no se ha alcanzado el límite superior, movemos la pala 1:

```
call  PreviousScan
ld    (paddle1pos), hl
jr    movePaddle_2Up
```

Calculamos la nueva posición para la pala 1, `CALL PreviousScan`, la cargamos en memoria, `LD (paddle1pos), HL`, y saltamos a comprobar los controles del jugador 2, `JR movePaddle_2Up`.

Si no se ha pulsado la tecla arriba del jugador 1, comprobamos si se ha pulsado la tecla abajo:

```
movePaddle_1Down:
bit   $01, d
jr    z, movePaddle_2Up
```

Evaluamos si se ha pulsado la tecla abajo del jugador 1, `BIT $01, D`. Si no se ha pulsado saltamos a comprobar los controles del jugador 2, `JR Z, movePaddle_2Up`.

Si no salta, se ha pulsado la tecla abajo del jugador 1:

```
ld    hl, (paddle1pos)
ld    a, PADDLE_BOTTOM
call  CheckBottom
jr    z, movePaddle_2Up
```

Cargamos la posición de la pala 1 en HL, `LD HL, (paddle1pos)`, el límite inferior para las palas en A, `LD A, PADDLE_BOTTOM`, y comprobamos si se ha alcanzado, `CALL CheckBottom`. Si se ha alcanzado el límite saltamos a comprobar los controles del jugador 2, `JR Z, movePaddle_2Up`.

Si no se ha alcanzado el límite inferior, mueve la pala 1:

```
call  NextScan
ld    (paddle1pos), hl
```

Calculamos la nueva posición para la pala 1, `CALL NextScan`, y la cargamos en memoria, `LD (paddle1pos), HL`.

Hacemos las comprobaciones con los controles del jugador 2. Dada la semejanza simplemente marcamos en rojo los cambios con respecto a la comprobación del jugador 1:

```
movePaddle_2Up:
bit   $02, d
jr    z, movePaddle_2Down
ld    hl, (paddle2pos)
ld    a, PADDLE_TOP
call  CheckTop
jr    z, movePaddle_End
```

```
call   PreviousScan
ld     (paddle2pos), hl
jr     movePaddle_End

movePaddle_2Down:
bit    $03, d
jr     z, movePaddle_End
ld     hl, (paddle2pos)
ld     a, PADDLE_BOTTOM
call   CheckBottom
jr     z, movePaddle_End
call   NextScan
ld     (paddle2pos), hl

movePaddle_End:
ret
```

El aspecto final de la rutina es el siguiente:

```
; -------------------------------------------------------------------------------
; Calcula la posición de las palas para moverlas.
; Entrada:     D -> Pulsaciones de los controles
; Altera el valor de los registros AF y HL.
; -------------------------------------------------------------------------------
MovePaddle:
bit     $00, d                ; Evalúa si se ha pulsado la A
jr      z, movePaddle_1Down   ; Si no se ha pulsado salta
ld      hl, (paddle1pos)      ; Carga en HL la posición de la pala 1
ld      a, PADDLE_TOP         ; Carga en A el margen superior
call    CheckTop              ; Evalúa si se ha alcanzado el margen superior
jr      z, movePaddle_2Up     ; Si se ha alcanzado, salta
call    PreviousScan          ; Obtiene el scanline anterior a la posición de la pala 1
ld      (paddle1pos), hl      ; Carga en memoria la nueva posición de la pala 1
jr      movePaddle_2Up        ; Salta

movePaddle_1Down:
bit     $01, d                ; Evalúa si se ha pulsado la Z
jr      z, movePaddle_2Up     ; Si no se ha pulsado salta
ld      hl, (paddle1pos)      ; Carga en HL la posición de la pala 1
ld      a, PADDLE_BOTTOM      ; Carga en A el margen inferior
```

```
call    CheckBottom            ; Evalúa si se ha alcanzado el margen inferior
jr      z, movePaddle_2Up      ; Si se ha alcanzado, salta
call    NextScan               ; Obtiene el scanline siguiente a la posición de la pala 1
ld      (paddle1pos), hl       ; Carga en memoria la nueva posición de la pala 1

movePaddle_2Up:
bit     $02, d                 ; Evalúa si se ha pulsado el 0
jr      z, movePaddle_2Down    ; Si no se ha pulsado salta
ld      hl, (paddle2pos)       ; Carga en HL la posición de la pala 2
ld      a, PADDLE_TOP          ; Carga en A el margen superior
call    CheckTop               ; Evalúa si se ha alcanzado el margen superior
jr      z, movePaddle_End      ; Si se ha alcanzado, salta
call    PreviousScan           ; Obtiene el scanline anterior a la posición de la pala 2
ld      (paddle2pos), hl       ; Carga en memoria la nueva posición de la pala 2
jr      movePaddle_End         ; Salta

movePaddle_2Down:
bit     $03, d                 ; Evalúa si se ha pulsado la O
jr      z, movePaddle_End      ; Si no se ha pulsado salta
ld      hl, (paddle2pos)       ; Carga en HL la posición de la pala 2
ld      a, PADDLE_BOTTOM       ; Carga en A el margen inferior
call    CheckBottom            ; Evalúa si se ha alcanzado el margen inferior
jr      z, movePaddle_End      ; Si se ha alcanzado, salta
call    NextScan               ; Obtiene el scanline siguiente a la posición de la pala 2
ld      (paddle2pos), hl       ; Carga en memoria la nueva posición de la pala 2

movePaddle_End:
ret
```

Podemos ahorrar 2 bytes y 2 ciclos de reloj, de la misma forma que en la entrega anterior. Esta vez no daremos la solución en la última entrega, ya que es similar a la que se verá para la entrega anterior.

Para terminar, vamos a implementar en Main.asm las llamadas a esta rutina, dentro de nuestro bucle infinito, justo encima de la etiqueta loop_continue:

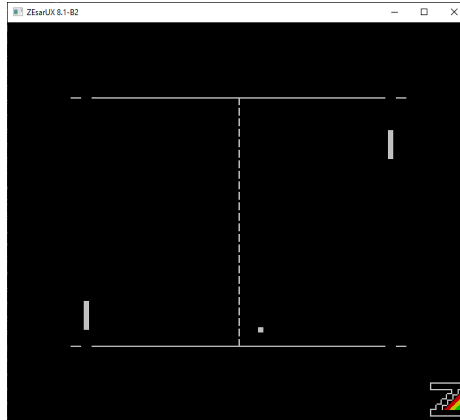
```
loop_paddle:
call  ScanKeys
call  MovePaddle
```

Primero comprobamos las teclas de control que se han pulsado, `CALL ScanKeys`, y luego movemos las palas, `CALL MovePaddle`.

También tenemos que cambiar la etiqueta a la que salta cuando la bola no se mueve, cuatro líneas más arriba:

```
cp      $06
jr      nz, loop_paddle
call    MoveBall
```

Compilamos y probamos en el emulador:



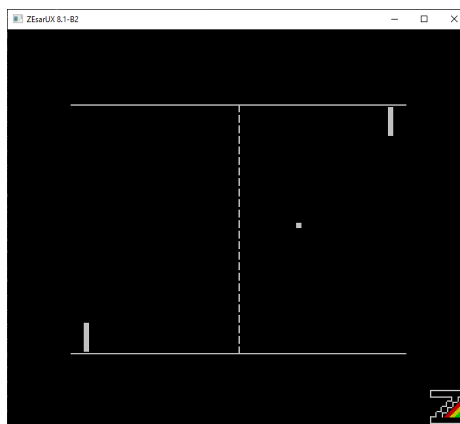
Observamos dos problemas:

1 Las palas borran el borde.
2 Las palas se mueven muy rápido y son difíciles de controlar.

Para resolver el primer problema vamos a modificar las constantes que marcan los límites superior e inferior de las palas, que están en Sprite.asm:

```
PADDLE_BOTTOM:     EQU     $a6
PADDLE_TOP:        EQU     $02
```

Compilamos, cargamos en el emulador y comprobamos que ya no se borra el borde:



Para reducir la velocidad del movimiento de las palas, vamos a usar la misma técnica que usamos con la bola, no vamos a mover las palas en cada iteración del bucle.

Lo primero es declarar la variable que usaremos como contador, lo que hacemos antes de END $8000:

```
countLoopPaddle:  db     $00
```

Ahora, justo debajo de la etiqueta loop_paddle, implementamos la comprobación del contador:

```
ld    a, (countLoopPaddle)
inc   a
ld    (countLoopPaddle), a
cp    $02
jr    nz, loop_continue
call  ScanKeys
call  MovePaddle
```

Cargamos el contador en A, `LD A, (countLoopPaddle)`, lo incrementamos, `INC A`, y lo cargamos en memoria, `LD (countLoopPaddle), A`. Evaluamos si han pasado las veces que hemos definido para mover las palas, `CP $02`, y si no es así saltamos, `JR NZ, loop_continue`.

Si no salta, comprobamos si se ha pulsado alguna tecla de control, `CALL ScanKeys`, y movemos las palas, `CALL MovePaddle`, y tal y como hicimos con la bola, hay que poner el contador a cero. Añadimos las líneas siguientes antes de la etiqueta loop_continue:

```
ld    a, ZERO
ld    (countLoopPaddle), a
```

Cargamos 0 en A, `LD A, ZERO`, y lo cargamos en memoria, `LD (countLoopPaddle), A`, poniendo el contador a 0.

Compilamos y cargamos en el emulador. Ahora el control de las palas es menos rápido y más preciso.

El código final de Main.asm es el siguiente:

```
; Pintado de campo, movimiento de palas y bola y temporización
org     $8000

; ------------------------------------------------------------------------------
; Entrada al programa
; ------------------------------------------------------------------------------
Main:
ld      a, $00              ; A = 0
out     ($fe), a            ; Pone el borde en negro

call    Cls                 ; Limpia la pantalla
call    PrintLine           ; Imprime la línea central
call    PrintBorder         ; Imprime el borde del campo

Loop:
```

```
ld      a, (countLoopBall)    ; Carga el contador de vueltas de la bola
inc     a                     ; Lo incrementa
ld      (countLoopBall), a    ; Lo carga en memoria
cp      $06                   ; Comprueba si ha llegado a 6
jr      nz, loop_paddle       ; Si no ha llegado a 4 salta
call    MoveBall              ; Mueve la bola
ld      a, ZERO               ; Pone el contador a 0
ld      (countLoopBall), a    ; Lo carga en memoria

loop_paddle:
ld      a, (countLoopPaddle)  ; Carga el contador de vueltas de las palas
inc     a                     ; Lo incrementa
ld      (countLoopPaddle), a  ; Lo carga en memoria
cp      $02                   ; Comprueba si ha llegado a 2
jr      nz, loop_continue     ; Si no ha llegado a 2 salta
call    ScanKeys              ; Escanea las teclas pulsadas
call    MovePaddle            ; Mueva las palas
ld      a, ZERO               ; Pone el contador a 0
ld      (countLoopPaddle), a  ; Lo carga en memoria

loop_continue:
call    PrintBall             ; Pinta la bola
call    ReprintLine           ; Repinta la línea
ld      hl, (paddle1pos)      ; Carga en HL la posición de la pala 1
call    PrintPaddle           ; Pinta la pala 1
ld      hl, (paddle2pos)      ; Carga en HL la posición de la pala 2
call    PrintPaddle           ; Pinta la pala 2
jr      Loop                  ; Bucle infinito

include "Game.asm"
include "Controls.asm"
include "Sprite.asm"
include "Video.asm"

countLoopBall:        db $00  ; Contador de vueltas de la bola
countLoopPaddle:      db $00  ; Contador de vueltas de las palas


end     $8000
```

Paso 7: detección de colisiones

Creamos la carpeta Paso07 y copiamos desde la carpeta Paso06 los archivos Controls.asm, Game.asm, Main.asm, Sprite.asm y Video.asm.

A partir de aquí, vamos a utilizar todo lo que hemos implementado hasta ahora, evolucionándolo.

Vamos a implementar la detección de colisiones de la bola con las palas. Para ello necesitamos definir la columna en la que se produce dicha colisión, lo que vamos a hacer en Sprite.asm:

```
CROSS_LEFT:        EQU    $01
CROSS_RIGHT:       EQU    $1d
```

Para comprobar la colisión en la coordenada X, vamos a usar la columna. Para comprobar la colisión en la coordenada Y vamos a usar tercio, línea y scanline.

Como hemos visto anteriormente, la composición de la coordenada Y se encuentra en dos bytes distintos (010T TSSS LLLC CCCC), por lo que vamos a implementar una rutina que reciba una posición de memoria de la pantalla y devuelva la coordenada Y (TTLLLSSS).

La rutina la vamos a implementar en Video.asm, tras la rutina Cls, y recibe la posición de memoria de la pantalla en HL y devuelve la coordenada Y obtenida en A:

```
GetPtrY:
ld     a, h
and    $18
rlca
rlca
rlca
ld     e, a
```

Cargamos el tercio y scanline en A, `LD A, H`, nos quedamos con el tercio, `AND $18`, lo pasamos a los bits 6 y 7, `RLCA, RLCA, RLCA`, y cargamos el resultado en E, `LD E, A`.

```
ld     a, h
and    $07
or     e
ld     e, a
```

Volvemos a cargar tercio y scanline en A, `LD A, H`, nos quedamos con el scanline, `AND $07`, le agregamos el tercio, `OR E`, y cargamos el resultado en E, `LD E, A`.

```
ld     a, l
and    $e0
rrca
rrca
or     e
```

```
ret
```

Cargamos la línea y columna en A, `LD A, L`, nos quedamos con la línea, `AND $E0`, ponemos el valor en los bits 3 a 5, `RRCA, RRCA`, y le añadimos tercio y scanline, `OR E`.

El aspecto final de la rutina es:

```
; ------------------------------------------------------------------------------
; Obtiene tercio, línea y scanline de una posición de memoria.
; Entrada:     HL      ->      Posición de memoria.
; Salida:      A       ->      Tercio, línea y scanline obtenido.
; Altera el valor de los registros AF y E.
; ------------------------------------------------------------------------------
GetPtrY:
ld      a, h    ; Carga en A el valor de H (tercio y scanline)
and     $18     ; Se queda con el tercio
rlca
rlca
rlca            ; Pasa el valor del tercio a los bits 6 y 7
ld      e, a    ; Carga el valor en E
ld      a, h    ; Carga en A el valor de H (tercio y scanline)
and     $07     ; Se queda con el scanline
or      e       ; Lo mezcla con E
ld      e, a    ; Carga el valor en E TT***SSS
ld      a, l    ; Carga en A el valor de L (línea y columna)
and     $e0     ; Se queda con la línea
rrca
rrca            ; Pasa el valor a los bits 3 a 5
or      e       ; Lo mezcla con E (TTLLLSSS)

ret
```

Este tipo de conversión ya la hicimos en la rutina checkVerticalLimit, pero al ser necesaria en más de una rutina, la hemos implementado como una rutina aparte.

Para probarla, vamos a modificar la rutina checkVerticalLimit, sustituyendo casi toda ella por una llamada a GetPtrY, quedando de la siguiente manera:

```
; ------------------------------------------------------------------------------
; Evalúa si se ha alcanzado el límite vertical.
; Entrada:     A -> Límite vertical (TTLLLSSS).
;              HL -> Posición actual (010TTSSS LLLCCCCC).
; Altera el valor de los registros AF y BC.
; -----------------------------------------------------------------------------
checkVerticalLimit:
```

```
ld      b, a            ; Guarda el valor de A en B
call    GetPtrY         ; Obtiene la coordenada Y (TTLLLSSSS)
                        ; de la posición actual
cp      b               ; Lo compara con B. B = valor original de A = Límite vertical
ret
```

Compilamos, cargamos en el emulador y comprobamos que no se ha roto nada.

Ahora vamos a implementar la detección de colisiones, en el archivo Game.asm.

Empezamos por la rutina que evalúa si hay colisión en el eje X. Esta rutina recibe en C la columna donde se produce la colisión, y activa el flag Z si se ha producido:

```
CheckCrossX:
ld     a, (ballPos)
and    $1f
cp     c
ret
```

Cargamos la posición de la bola en A, `LD A, (ballPos)`, nos quedamos con la columna, `AND $1F`, y comparamos el valor resultante con la columna de colisión, `CP C`.

El siguiente paso es implementar la rutina que evalúa si hay colisión en el eje Y. Esta rutina recibe en HL la posición de la pala, y activa el flag Z si hay colisión:

```
CheckCrossY:
call   GetPtrY
inc    a
ld     c, a
```

Obtenemos la coordenada Y de la pala, `CALL GetPtrY`. Como el primer scanline de la pala es blanco, no lo tenemos en cuenta para las colisiones, así que pasamos al siguiente, `INC A`, y cargamos el valor en C, `LD C, A`.

```
ld     hl, (ballPos)
call   GetPtrY
ld     b, a
```

Cargamos la posición de la bola en HL, `LD HL, (ballPos)`, obtenemos la coordenada Y, `CALL GetPtrY`, y cargamos el valor en B, `LD B, A`.

```
add    a, $04
sub    c
ret    c
```

En A tenemos la coordenada Y de la bola. Apuntamos A al penúltimo scanline de la bola, el último que no es blanco, `ADD A, $04`, le restamos la coordenada Y de la pala, `SUB C`, y si hay acarreo salimos, `RET C`, ya que la bola pasa por encima de la pala.

Si no hemos salido, tenemos que comprobar si la bola pasa por debajo de la pala:

```
ld    a, c
add   a, $16
ld    c, a
ld    a, b
inc   a
sub   c
ret   nc
xor   a
ret
```

Cargamos la coordenada Y de la pala en A, `LD A, C`, le sumamos $16 (22) para posicionarnos en el penúltimo scanline, el último que no está a 0, `ADD A, $16`, y cargamos el valor en C, `LD C, A`.

Cargamos la coordenada Y de la bola en A, `LD A, B`, la apuntamos al scanline 1, el primero que no está a 0, `INC A`, y le restamos la coordenada Y de la pala, `SUB C`.

Si tras la resta no hay acarreo salimos, `RET NC`, pues o bien la bola pasa por debajo, o colisiona en el último scanline de la pala, que está en la misma coordenada Y que el primer scanline de la bola, y al restar se activa el flag Z.

Si hay acarreo, la bola colisiona con el resto de la pala, por lo que activamos el flag Z, `XOR A`, y salimos, `RET`.

El siguiente paso es implementar la rutina principal a la que vamos a llamar para comprobar si hay colisión, en cuyo caso vamos a realizar las acciones necesarias:

```
CheckBallCross:
ld    a, (ballSetting)
and   $40
jr    nz, checkBallCross_left
```

Cargamos la configuración de la bola en A, `LD A, (ballSetting)`, y nos quedamos con el bit 6, `AND $40`, que especifica si la bola va hacia la derecha o hacia la izquierda. Si el bit 6 está a 1, la bola va hacia la izquierda y saltamos a comprobar si se produce colisión con la pala del jugador 1, `JR NZ, checkBallCross_left`.

Si no se ha producido el salto, la bola va hacia la derecha y comprobamos si hay colisión con la pala del jugador 2:

```
checkBallCross_right:
ld    c, CROSS_RIGHT
call  CheckCrossX
ret   nz
ld    hl, (paddle2pos)
```

```
call   CheckCrossY
ret    nz
```

Cargamos la columna de colisión en C, `LD C, CROSS_RIGHT`, evaluamos si se produce colisión en el eje X, `CALL CheckCrossX`. Si no se produce la colisión salimos de la rutina, `RET NZ`.

Si se ha producido colisión en el eje X, cargamos la posición de la pala 2 en HL, `LD HL, (paddle2pos)`, y evaluamos si se produce colisión en el eje Y, `CALL CheckCrossY`. Si no se produce colisión, salimos de la rutina, `RET NZ`.

Si no hemos salido de la rutina, se ha producido colisión:

```
ld     a, (ballSetting)
or     $40
ld     (ballSetting), a
ld     a, $ff
ld     (ballRotation), a
ret
```

Cargamos la configuración de la bola en A, `LD A, (ballSetting)`, ponemos el bit 6 a 1 para cambiar la dirección de la bola hacia la izquierda, `OR $40`, y cargamos el valor en memoria, `LD (ballSetting), A`.

Cargamos -1 en A, `LD A, $FF`, cambiamos la rotación de la bola, `LD (ballRotation), A`, y salimos de la rutina, `RET`.

La comprobación de si hay colisión con la pala del jugador 1 es similar a lo visto anteriormente, por lo que solo vamos a poner el código y a marcar en rojo la diferencias, sin entrar en detalle:

```
checkBallCross_left:
ld     c, CROSS_LEFT
call   CheckCrossX
ret    nz
ld     hl, (paddle1pos)
call   CheckCrossY
ret    nz

ld     a, (ballSetting)
and    $bf
ld     (ballSetting), a
ld     a, $01
ld     (ballRotation), a
ret
```

El aspecto final de las rutinas de comprobación de colisiones entre las palas y la bola es el siguiente:

```
; --------------------------------------------------------------------------------
; Evalúa si hay colisión entre la bola y las palas.
; Altera el valor de los registros AF, C y HL.
; --------------------------------------------------------------------------------
CheckBallCross:
ld      a, (ballSetting)            ; Carga la dirección/velocidad de la bola en A
and     $40                         ; Se queda con el bit 6 (izquierda/derecha)
jr      nz, checkBallCross_left     ; Si no está a 0 va hacia la izquierda y salta

checkBallCross_right:
ld      c, CROSS_RIGHT              ; Carga la columna de colisión en C
call    CheckCrossX                 ; Evalúa si se produce colisión en el eje X
ret     nz                          ; Si no se produce, fin de la rutina
ld      hl, (paddle2pos)            ; Carga la posición de la pala 2 en HL
call    CheckCrossY                 ; Evalúa si se produce colisión en el eje Y
ret     nz                          ; Si no se produce colisión, fin de la rutina

; Si llega aquí hay colisión
ld      a, (ballSetting)            ; Carga la dirección/velocidad de la bola en A
or      $40                         ; Cambia la dirección, la pone hacia la izquierda
ld      (ballSetting), a            ; Carga el valor en memoria
ld      a, $ff                      ; Cambia la rotación de la bola
ld      (ballRotation), a           ; La carga en memoria
ret                                 ; Fin de la rutina

checkBallCross_left:
; La bola va hacia la izquierda
ld      c, CROSS_LEFT               ; Carga la columna de colisión en C
call    CheckCrossX                 ; Evalúa si se produce colisión en el eje X
ret     nz                          ; Si no se produce, fin de la rutina
ld      hl, (paddle1pos)            ; Carga la posición de la pala 1 en HL
call    CheckCrossY                 ; Evalúa si se produce colisión en el eje Y
ret     nz                          ; Si no se produce colisión, fin de la rutina

; Si llega aquí hay colisión
ld      a, (ballSetting)            ; Carga la dirección/velocidad de la bola en A
and     $bf                         ; Cambia la dirección, la pone hacia la derecha
ld      (ballSetting), a            ; Carga el valor en memoria
ld      a, $01                      ; Cambia la rotación de la bola
```

```
ld      (ballRotation), a             ; La carga en memoria
ret                                   ; Fin de la rutina

; ------------------------------------------------------------------------------
; Evalúa si la bola colisiona en el eje X con la pala.
; Entrada:     C -> Columna dónde se produce la colisión.
; Salida:      Z -> Colisiona.
;              NZ -> No colisiona.
; Altera el valor de los registros AF.
; ------------------------------------------------------------------------------
CheckCrossX:
ld      a, (ballPos)          ; Carga la línea y columna donde está la bola
and     $1f                   ; Se queda con la columna
cp      c                     ; Lo compara con la columna de colisión

ret

; ------------------------------------------------------------------------------
; Evalúa si la bola colisiona en el eje Y con la pala.
; Entrada:     HL -> Posición de la pala
; Salida:      Z -> Colisiona.
;              NZ -> No colisiona.
; Altera el valor de los registros AF, BC y HL.
; ------------------------------------------------------------------------------
CheckCrossY:
call    GetPtrY               ; Obtiene la posición vertical de la pala (TTLLLSSS)
; La posición devuelta apunta al primer scanline de la pala que está a 0,
; apunta al siguiente
inc     a
ld      c, a                  ; Carga el valor en C
ld      hl, (ballPos)         ; Carga en HL la posición de la bola
call    GetPtrY               ; Obtiene la posición vertical de la bola (TTLLLSSS)
ld      b, a                  ; Carga el valor en B
; Comprueba si la bola pasa por encima de la pala
; La bola está compuesta de 1 scanline a 0, 4 a $3c y otro a 0
; La posición apunta al 1er scanline, y se comprueba la colisión con el 5º
add     a, $04                ; Apunta la posición de la bola al 5º scanline
sub     c                     ; Resta a la posición de la bola, la posición de la pala
ret     c                     ; Si hay acarreo sale porque la bola pasa por encima
; Comprueba si la bola pasa por debajo de la pala
ld      a, c                  ; Carga la posición vertical de la pala en A
```

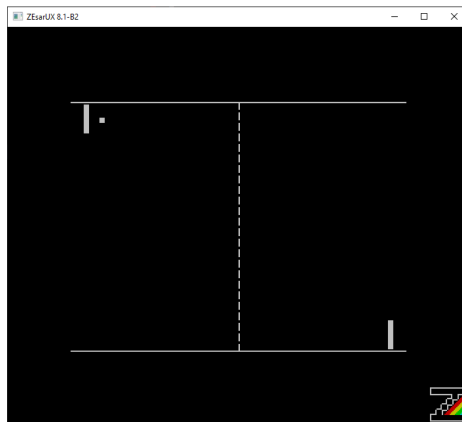
```
add     a, $16                  ; Le suma 22 para apuntar al penúltimo scanline,
                                ; último que no es 0
ld      c, a                    ; Lo vuelve a cargar en C
ld      a, b                    ; Carga la posición vertical de la bola
inc     a                       ; Le suma 1 para apuntar el scanline 1, primero que no es 0
sub     c                       ; Resta a la posición de la bola, la posición de la pala
ret     nc                      ; Si no hay acarreo la bola pasa por debajo de la pala
                                ; o colisiona en el último scanline.
                                ; En este último caso se activa el flag Z
; Hay colisión
xor     a                       ; Activa el flag Z
ret
```

Ahora ya solo nos queda ver si lo que hemos implementado hace lo que pretendemos.

Abrimos el archivo Main.asm y justo debajo de la etiqueta loop_continue, añadimos la siguiente línea:

```
call  CheckBallCross
```

Compilamos y cargamos en el emulador para ver los resultados. Si todo ha ido bien, la bola choca contra las palas; la detección de colisiones está funcionando.



El aspecto final del archivo Main.asm es el siguiente:

```
; Detección de colisiones
org     $8000


; ------------------------------------------------------------------------------
; Entrada al programa
; ------------------------------------------------------------------------------
Main:
ld      a, $00                  ; A = 0
out     ($fe), a                ; Pone el borde en negro
```

```
call    Cls                   ; Limpia la pantalla
call    PrintLine             ; Imprime la línea central
call    PrintBorder           ; Imprime el borde del campo

Loop:
ld      a, (countLoopBall)    ; Carga el contador de vueltas de la bola
inc     a                     ; Lo incrementa
ld      (countLoopBall), a    ; Lo carga en memoria
cp      $06                   ; Comprueba si ha llegado a 6
jr      nz, loop_paddle       ; Si no ha llegado a 6 salta
call    MoveBall              ; Mueve la bola
ld      a, ZERO               ; Pone el contador a 0
ld      (countLoopBall), a    ; Lo carga en memoria

loop_paddle:
ld      a, (countLoopPaddle)  ; Carga el contador de vueltas de las palas
inc     a                     ; Lo incrementa
ld      (countLoopPaddle), a  ; Lo carga en memoria
cp      $02                   ; Comprueba si ha llegado a 2
jr      nz, loop_continue     ; Si no ha llegado a 2 salta
call    ScanKeys              ; Escanea las teclas pulsadas
call    MovePaddle            ; Mueva las palas
ld      a, ZERO               ; Pone el contador a 0
ld      (countLoopPaddle), a  ; Lo carga en memoria

loop_continue:
call    CheckBallCross        ; Evalúa si hay colisión entre la bola y las palas
call    PrintBall             ; Pinta la bola
call    ReprintLine           ; Repinta la línea
ld      hl, (paddle1pos)      ; Carga en HL la posición de la pala 1
call    PrintPaddle           ; Pinta la pala 1
ld      hl, (paddle2pos)      ; Carga en HL la posición de la pala 2
call    PrintPaddle           ; Pinta la pala 2
jr      Loop                  ; Bucle infinito

include "Game.asm"
include "Controls.asm"
include "Sprite.asm"
include "Video.asm"

countLoopBall:        db $00  ; Contador de vueltas de la bola
```

```
countLoopPaddle:      db $00 ; Contador de vueltas de las palas


end    $8000
```

Paso 8: partida a dos jugadores y cambio de velocidad de la bola

En este paso vamos a implementar la partida a dos jugadores, con marcador, y la posibilidad de cambiar la velocidad de la bola.

Creamos la carpeta Paso08 y copiamos los archivos Controls.asm, Game.asm, Main.asm, Sprite.asm y Video.asm desde la carpeta Paso07.

Vamos a empezar por el marcador, definiendo la posición donde vamos a pintar la puntuación, y definiendo también los sprites necesarios en el archivo Sprite.asm:

```
POINTS_P1:  EQU   $450d

POINTS_P2:  EQU   $4511
```

Cada dígito de los marcadores ocupa 8x16 píxeles, o lo que es lo mismo, 1 carácter de ancho por 2 de alto (1 byte x 16 bytes/scanlines):

```
Blanco_sprite:

ds $10      ; 16 espacios = 16 bytes a $00


Cero_sprite:

db $00, $7e, $7e, $66, $66, $66, $66, $66

db $66, $66, $66, $66, $66, $7e, $7e, $00


Uno_sprite:

db $00, $18, $18, $18, $18, $18, $18, $18

db $18, $18, $18, $18, $18, $18, $18, $00


Dos_sprite:

db $00, $7e, $7e, $06, $06, $06, $06, $7e

db $7e, $60, $60, $60, $60, $7e, $7e, $00


Tres_sprite:

db $00, $7e, $7e, $06, $06, $06, $06, $3e

db $3e, $06, $06, $06, $06, $7e, $7e, $00


Cuatro_sprite:

db $00, $66, $66, $66, $66, $66, $66, $7e

db $7e, $06, $06, $06, $06, $06, $06, $00
```

```
Cinco_sprite:
db $00, $7e, $7e, $60, $60, $60, $60, $7e
db $7e, $06, $06, $06, $06, $7e, $7e, $00

Seis_sprite:
db $00, $7e, $7e, $60, $60, $60, $60, $7e
db $7e, $66, $66, $66, $66, $7e, $7e, $00

Siete_sprite:
db $00, $7e, $7e, $06, $06, $06, $06, $06
db $06, $06, $06, $06, $06, $06, $06, $00

Ocho_sprite:
db $00, $7e, $7e, $66, $66, $66, $66, $7e
db $7e, $66, $66, $66, $66, $7e, $7e, $00

Nueve_sprite:
db $00, $7e, $7e, $66, $66, $66, $66, $7e
db $7e, $06, $06, $06, $06, $7e, $7e, $00
```

Una vez que hemos definido los sprites, definimos la composición de los números haciendo referencia a las etiquetas de los sprites:

```
Cero:
dw          Blanco_sprite, Cero_sprite

Uno:
dw          Blanco_sprite, Uno_sprite

Dos:
dw          Blanco_sprite, Dos_sprite

Tres:
dw          Blanco_sprite, Tres_sprite

Cuatro:
dw          Blanco_sprite, Cuatro_sprite
```

```
Cinco:
dw          Blanco_sprite, Cinco_sprite

Seis:
dw          Blanco_sprite, Seis_sprite

Siete:
dw          Blanco_sprite, Siete_sprite

Ocho:
dw          Blanco_sprite, Ocho_sprite

Nueve:
dw          Blanco_sprite, Nueve_sprite

Diez:
dw          Uno_sprite, Cero_sprite

Once:
dw          Uno_sprite, Uno_sprite

Doce:
dw          Uno_sprite, Dos_sprite

Trece:
dw          Uno_sprite, Tres_sprite

Catorce:
dw          Uno_sprite, Cuatro_sprite

Quince:
dw          Uno_sprite, Cinco_sprite
```

Ahora necesitamos definir el lugar donde vamos a guardar la puntuación de cada jugador. Abrimos el archivo Main.asm y añadimos las siguientes variables antes de END $8000:

```
p1points:   db    $00
p2points:   db    $00
```

Ya tenemos todo listo para empezar a implementar el marcador.

Lo primero que tenemos que saber es que sprite tenemos que pintar, dependiendo del marcador de cada jugador. Para saber qué sprite pintar, vamos a implementar una rutina que recibe en A la puntuación, y devuelve en HL la dirección del sprite a pintar.

Abrimos el archivo Video.asm e implementamos justo antes de la rutina NextScan:

```
GetPointSprite:
ld    hl, Cero
ld    bc, $04
inc   a
```

Cargamos en HL la dirección del sprite para el cero, `LD HL, Cero`. Como cada sprite está a 4 bytes del anterior, cargamos este desplazamiento en BC, `LD BC, $04`, e incrementamos A para que el bucle no empiece en 0, `INC A`, en el caso de que la puntuación sea 0.

Ahora hacemos un bucle para que HL apunte al sprite correcto:

```
getPointSprite_loop:
dec   a
ret   z
add   hl, bc
jr    getPointSprite_loop
```

Decrementamos A, `DEC A`, y si hemos llegado a 0, HL ya apunta al sprite correcto y salimos, `RET Z`. Si todavía no hemos llegado a 0, sumamos el desplazamiento a HL, `ADD HL, BC`, y volvemos a ejecutar el bucle, `JR getPointSprite_loop`.

El aspecto final de la rutina es:

```
; -----------------------------------------------------------------------------
; Obtiene el sprite correspondiente a pintar en el marcador.
; Entrada:     A -> puntuación.
; Salida:      HL -> Dirección del sprite a pintar.
; Altera el valor de los registros AF, BC y HL.
; -----------------------------------------------------------------------------
GetPointSprite:
ld      hl, Cero              ; Carga en HL la dirección del sprite del 0
ld      bc, $04               ; Cada sprite está del anterior a 4 bytes
inc     a                     ; Incrementa A para que el inicio del bucle no sea 0
getPointSprite_loop:
dec     a                     ; Decrementa A
ret     z                     ; Si ha llegado a 0, fin de rutina
add     hl, bc                ; Suma 4 a la dirección del sprite; siguiente sprite
jr      getPointSprite_loop   ; Bucle hasta que A = 0
```

```
ret
```

Y ahora vamos a implementar la rutina que pinta los marcadores, al final del archivo Video.asm:

```
PrintPoints:
ld    a, (p1points)
call  GetPointSprite
```

Cargamos la puntuación del jugador 1 en A, `LD A, (p1points)`, y obtenemos la dirección de memoria donde está la definición del sprite correspondiente a dicha puntuación, `CALL GetPointSprite`.

GetPointSprite nos devuelve en HL la dirección de memoria donde está definido el sprite. Si la puntuación es cero, HL nos traerá la dirección de memoria donde está definida la etiqueta Cero, cuya definición es la siguiente:

```
Cero:
dw    Blanco_sprite, Cero_sprite
```

Como podemos ver, Cero está definido por otras dos direcciones de memoria: la primera es la dirección de memoria donde está definido el sprite blanco, usado para justificar a dos dígitos, y la segunda es la dirección de memoria donde está definido el sprite del cero.

Si las direcciones de memoria fueran las siguientes:

```
$9000 Blanco_sprite
$9020 Cero_sprite
$9040 Cero
```

La definición de la etiqueta Cero, una vez que se sustituyen las etiquetas Blanco_sprite y Cero_sprite por las direcciones de memoria donde están definidas, sería:

```
Cero:
dw    $9000, $9020
```

El valor que tendría HL tras llamar a GetPointSprite con el marcador a 0 sería $9040, o lo que es lo mismo, la dirección de memoria donde se define la etiqueta Cero.

Como el Z80 es Little Endian, los valores de las direcciones de memoria desde $9040 en adelante serían:

$9040	$00
$9041	$90
$9042	$20
$9043	$90

O lo que es lo mismo, las direcciones de memoria donde están definidos los sprites para Blanco_sprite y para Cero_sprite.

Esta explicación es necesaria para entender el funcionamiento del resto de la rutina:

```
push  hl
```

```
ld     e, (hl)
inc    hl
ld     d, (hl)
ld     hl, POINTS_P1
call   printPoint_print
```

Vamos a pintar el primer dígito del marcador del jugador 1. Preservamos el valor HL, que apunta al sprite del marcador que tenemos que pintar, `PUSH HL`, cargamos en E la parte baja de la dirección donde está el sprite del primer dígito, `LD E, (HL)`, apuntamos HL a la parte alta de la dirección, `INC HL`, y la cargamos en D, `LD D, (HL)`.

Cargamos en HL la dirección de memoria de pantalla donde se pinta el primer dígito del marcador del jugador 1, `LD HL, POINTS_P1`, y llamamos al pintado del dígito, `CALL printPoint_print`.

Ahora pintamos el segundo dígito del marcador del jugador 1:

```
pop    hl
inc    hl
inc    hl
```

Recuperamos el valor de HL, `POP HL`, y lo apuntamos a la parte baja de la dirección donde está definido el sprite del segundo dígito, `INC HL, INC HL`.

```
ld     e, (hl)
inc    hl
ld     d, (hl)
```

Cargamos la parte baja de dicha dirección en E, `LD E, (HL)`, apuntamos HL a la parte alta de la dirección, `INC HL`, y la cargamos en D, `LD D, (HL)`.

```
ld     hl, POINTS_P1
inc    l
call   printPoint_print
```

Por último, cargamos en HL la posición de memoria de la pantalla donde se pinta el marcador del jugador 1, `LD HL, POINTS_P1`. Como cada dígito ocupa 1 byte (columna) de ancho, situamos HL en la columna dónde se pinta el segundo dígito, `INC L`, y lo pintamos.

La forma de pintar el marcador del jugador 2 es casi igual a la del jugador 1, por lo que mostramos el código marcando en rojo los cambios y sin entrar en detalle:

```
ld     a, (p2points)
call   GetPointSprite
push   hl
; 1er dígito
ld     e, (hl)
```

```
inc   hl
ld    d, (hl)
ld    hl, POINTS_P2
call  printPoint_print

pop   hl
; 2° dígito
inc   hl
inc   hl
ld    e, (hl)
inc   hl
ld    d, (hl)
ld    hl, POINTS_P2
inc   l
```

Como se puede observar, los cambios son pocos. Se ha quitado la última línea al no ser necesario llamar a pintar el segundo dígito del jugador 2, ya que lo vamos a implementar a continuación del último INC L.

Recordemos que cada dígito ocupa 8x16 píxeles (1 columna x 16 scanlines):

```
printPoint_print:
ld    b, $10
push  de
push  hl
```

Cargamos en B el número de scanlines que vamos a pintar, `LD B, $10`, y preservamos el valor del registro DE, `PUSH DE`, y de HL, `PUSH HL`.

```
printPoint_printLoop:
ld    a, (de)
ld    (hl), a
inc   de
call  NextScan
djnz  printPoint_printLoop
```

Cargamos en A el byte a pintar, `LD A, (DE)`, y lo pintamos en pantalla, `LD (HL), A`. Apuntamos DE al siguiente byte a pintar, obtenemos la dirección del siguiente scanline, `CALL NextScan`, y repetimos la operación hasta que B sea 0 y hayamos pintado los 16 scanlines, `DJNZ printPoint_printLoop`.

Para finalizar, recuperamos los valores de HL y DE y salimos:

```
pop   hl
```

```
pop    de
ret
```

El aspecto final de la rutina de pintado del marcador es el siguiente:

```
; ------------------------------------------------------------------------------
; Pinta el marcador.
; Cada número consta de 1 byte de ancho por 16 de alto.
; Altera el valor de los registros AF, BC, DE y HL.
; ------------------------------------------------------------------------------
PrintPoints:
ld      a, (p1points)         ; Carga en A los puntos del jugador 1
call    GetPointSprite        ; Obtiene el sprite a pintar en el marcador
push    hl                    ; Preserva el valor de HL
; 1er dígito del jugador 1
ld      e, (hl)               ; Carga en E la parte baja de la dirección
                              ; donde está el primer dígito
inc     hl                    ; Apunta HL a la parte alta de la dirección
                              ; donde está el primer dígito
ld      d, (hl)               ; y la carga en D
ld      hl, POINTS_P1         ; Carga en HL la dirección de memoria donde se pintan
                              ; los puntos del jugador 1
call    printPoint_print      ; Pinta el primer dígito del marcador del jugador 1

pop     hl                    ; Recupera el valor de HL
; 2° dígito del jugador 1
inc     hl
inc     hl                    ; Apunta HL a la parte baja de la dirección
                              ; donde está el segundo dígito
ld      e, (hl)               ; y la carga en E
inc     hl                    ; Apunta HL a la parte alta de la dirección
                              ; donde está el segundo dígito
ld      d, (hl)               ; y la carga en D
ld      hl, POINTS_P1         ; Carga en HL la dirección de memoria donde se pintan
                              ; los puntos del jugador 1
inc     l                     ; Apunta HL a la dirección donde se pinta el segundo dígito
call    printPoint_print      ; Pinta el segundo dígito del marcador del jugador 1

ld      a, (p2points)         ; Carga en A los puntos del jugador 2
call    GetPointSprite        ; Obtiene el sprite a pintar en el marcador
push    hl                    ; Preserva el valor de HL
; 1er dígito del jugador 2
```

```
ld      e, (hl)             ; Carga en E la parte baja de la dirección
                            ; donde está el primer dígito
inc     hl                  ; Apunta HL a la parte alta de la dirección
                            ; donde está el primer dígito
ld      d, (hl)             ; y la carga en D
ld      hl, POINTS_P2       ; Carga en HL la dirección de memoria donde se pintan
                            ; los puntos del jugador 2
call    printPoint_print    ; Pinta el primer dígito del marcador del jugador 2

pop     hl                  ; Recupera el valor de HL
; 2º dígito del jugador 2
inc     hl
inc     hl                  ; Apunta HL a la parte baja de la dirección
                            ; donde está el segundo dígito
ld      e, (hl)             ; y la carga en E
inc     hl                  ; Apunta HL a la parte alta de la dirección
                            ; donde está el segundo dígito
ld      d, (hl)             ; y la carga en D
ld      hl, POINTS_P2       ; Carga en HL la dirección de memoria donde se pintan
                            ; los puntos del jugador 2
inc     l                   ; Apunta HL a la dirección donde se pinta el segundo dígito
; Pinta el segundo dígito del marcador del jugador 2

printPoint_print:
ld      b, $10              ; Cada dígito son 1 byte por 16 (scanlines)
push    de                  ; Preserva el valor de DE
push    hl                  ; Preserva el valor de HL
printPoint_printLoop:
ld      a, (de)             ; Carga en A el byte a pintar
ld      (hl), a             ; Pinta el byte
inc     de                  ; Apunta DE al siguiente byte
call    NextScan            ; Apunta HL al siguiente scanline
djnz    printPoint_printLoop  ; Hasta que B = 0

pop     hl                  ; Recupera el valor de HL
pop     de                  ; Recupera el valor de DE

ret
```

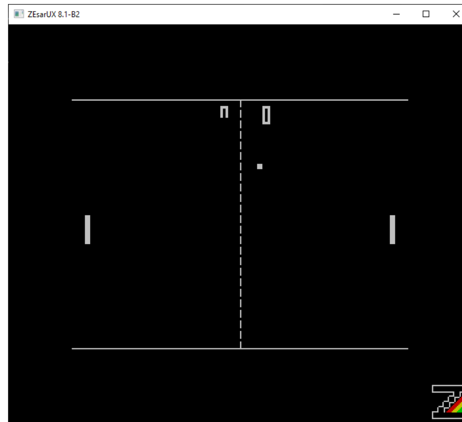
En esta rutina es sencillo ahorrar 12 ciclos de reloj y 2 bytes. Para ello hay que cambiar dos instrucciones de lugar, lo que nos permite quitar otras dos: lo veremos en el paso 10.

Y ahora solo nos queda ver si lo que hemos implementado funciona.

Abrimos Main.asm, y debajo de la llamada a PrintBorder, justo antes de Loop, añadimos la siguiente línea:

```
call   PrintPoints
```

Compilamos y cargamos en el emulador para ver los resultados:



En principio todo va bien, pero según se va moviendo la bola vemos que volvemos a tener un problema, viejo conocido nuestro, y es que la bola borra el marcador a su paso, cosa que vamos a solucionar a continuación.

Para evitar que la bola borre el marcador, hacemos lo mismo que hicimos con la línea central, vamos a repintar el marcador.

Implementamos la rutina al final del archivo Video.asm.

En realidad, la rutina de repintado del marcador es prácticamente igual que la de pintado, cambiando el nombre de las etiquetas y añadiendo una línea.

Vamos a copiar toda la rutina de pintado de marcador y la vamos a pegar al final del archivo Video.asm. Cambiamos los nombres de las etiquetas y añadimos una línea.

A continuación, mostramos el aspecto final, marcando en rojo los cambios producidos con respecto a la rutina de pintado del marcador:

```
; -----------------------------------------------------------------------------
; Repinta el marcador.
; Cada número consta de 1 bytes de ancho por 16 de alto.
; Altera el valor de los registros AF, BC, DE y HL.
; -----------------------------------------------------------------------------
ReprintPoints:
ld      a, (p1points)          ; Carga en A los puntos del jugador 1
call    GetPointSprite         ; Obtiene el sprite a pintar en el marcador
push    hl                     ; Preserva el valor de HL
; 1er dígito del jugador 1
ld      e, (hl)                ; Carga en E la parte baja de la dirección
                               ; donde está el primer dígito
```

```
inc     hl                      ; Apunta HL a la parte alta de la dirección
                                ; donde está el primer dígito
ld      d, (hl)                 ; y la carga en D
ld      hl, POINTS_P1           ; Carga en HL la dirección de memoria donde se pintan
                                ; los puntos del jugador 1
call    reprintPoint_print      ; Pinta el primer dígito del marcador del jugador 1

pop     hl                      ; Recupera el valor de HL
; 2º dígito del jugador 1
inc     hl
inc     hl                      ; Apunta HL a la parte baja de la dirección
                                ; donde está el segundo dígito
ld      e, (hl)                 ; y la carga en E
inc     hl                      ; Apunta HL a la parte alta de la dirección
                                ; donde está el segundo dígito
ld      d, (hl)                 ; y la carga en D
ld      hl, POINTS_P1           ; Carga en HL la dirección de memoria donde se pintan
                                ; los puntos del jugador 1
inc     l                       ; Apunta HL a la dirección donde se pinta el segundo dígito
call    reprintPoint_print      ; Pinta el segundo dígito del marcador del jugador 1

ld      a, (p2points)           ; Carga en A los puntos del jugador 2
call    GetPointSprite          ; Obtiene el sprite a pintar en el marcador
push    hl                      ; Preserva el valor de HL
; 1er dígito del jugador 2
ld      e, (hl)                 ; Carga en E la parte baja de la dirección
                                ; donde está el primer dígito
inc     hl                      ; Apunta HL a la parte alta de la dirección
                                ; donde está el primer dígito
ld      d, (hl)                 ; y la carga en D
ld      hl, POINTS_P2           ; Carga en HL la dirección de memoria donde se pintan
                                ; los puntos del jugador 2
call    reprintPoint_print      ; Pinta el primer dígito del marcador del jugador 2

pop     hl                      ; Recupera el valor de HL
; 2º dígito del jugador 2
inc     hl
inc     hl                      ; Apunta HL a la parte baja de la dirección
                                ; donde está el segundo dígito
ld      e, (hl)                 ; y la carga en E
inc     hl                      ; Apunta HL a la parte alta de la dirección
```

```
                                ; donde está el segundo dígito
ld      d, (hl)                 ; y la carga en D
ld      hl, POINTS_P2           ; Carga en HL la dirección de memoria donde se pintan
                                ; los puntos del jugador 2
inc     l                       ; Apunta HL a la dirección donde se pinta el segundo dígito
; Pinta el segundo dígito del marcador del jugador 2

reprintPoint_print:
ld      b, $10                  ; Cada dígito es de 1 byte por 16 (scanlines)
push    de
push    hl                      ; Preserva el valor de los registros DE y HL
reprintPoint_printLoop:
ld      a, (de)                 ; Carga en A el byte a pintar
or      (hl)                    ; Lo mezcla con lo que hay pintado en pantalla
ld      (hl), a                 ; Pinta el byte
inc     de                      ; Apunta DE al siguiente byte
call    NextScan                ; Apunta HL al siguiente scanline
djnz    reprintPoint_printLoop; Hasta que B = 0

pop     hl
pop     de                      ; Recupera el valor de los registros HL y DE

ret
```

Vamos a explicar la línea que hemos añadido:

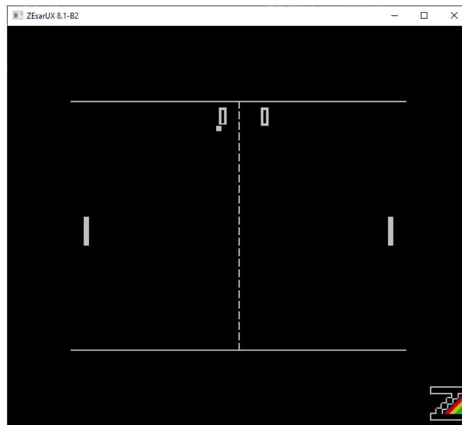
```
ld      a, (de)
or      (hl)
ld      (hl), a
```

Lo que hacemos con OR (HL) es agregar los píxeles que hay en pantalla a los píxeles del sprite del número. De esta manera repintamos el número sin borrar la bola.

Ahora queda ver si funciona. Abrimos el archivo Main.asm y añadimos la siguiente línea después de la llamada a ReprintLine:

```
call    ReprintPoints
```

Compilamos y cargamos en el emulador para ver los resultados:



Efectivamente, hemos solucionado un problema, pero ha surgido otro. El marcador ya no se borra, pero la bola va muy lenta. Por suerte la solución es sencilla, ya que la velocidad de la bola es una de las cosas que controlamos nosotros.

Como recordaréis, la bola se mueve 1 de cada 6 iteraciones del bucle principal, por lo que lo único que tenemos que hacer es reducir este intervalo en Main.asm, por ejemplo a 2:

```
ld     (countLoopBall), a
cp     $02
jr     nz, loop_paddle
```

Compilamos, cargamos en el emulador y comprobamos que la velocidad de la bola ha aumentado.

Como recodaremos, en la variable ballSetting definimos la velocidad de la bola en los bits 4 y 5, pudiendo ser 1 la más rápida y 3 la más lenta.

Vamos a utilizar este aspecto para definir y modificar la velocidad de la bola.

Lo primero es modificar el valor inicial de esta variable:

```
ballSetting:      db     $20
```

De esta manera el valor inicial es:

- Dirección vertical: arriba
- Dirección horizontal: derecha
- Velocidad: 2

Y ahora vamos a usar este valor para controlar el intervalo para mover la bola. Abrimos Main.asm, localizamos la etiqueta Loop, y añadimos justo debajo:

```
ld     a, (ballSetting)
rrca
rrca
rrca
rrca
and    $03
```

```
ld    b, a
```

Cargamos la configuración de la bola en A, `LD A, (ballSetting)`, pasamos el valor de los bits 4 y 5 a los bits 0 y 1, `RRCA, RRCA, RRCA, RRCA`, nos quedamos con el valor de los bits 0 y 1 (velocidad de la bola), `AND $03`, y cargamos el valor en B, `LD B, A`.

Cuatro líneas más abajo, cambiamos la línea CP $02:

```
cp    b
```

Compilamos y comprobamos que todo sigue funcionando igual. La única diferencia es que ahora la velocidad de la bola la tomamos desde la configuración de la misma, y podremos cambiarla.

Para cambiar la velocidad de la bola, vamos a usar las teclas del 1 al 3. Abrimos el archivo Controls.asm y empezamos a escribir tras la etiqueta ScanKeys:

```
scanKeys_speed:
ld    a, $00
ld    (countLoopBall), a
scanKeys_ctrl:
```

Si se ha pulsado algunas de las teclas de cambio de velocidad, hay que poner a 0 el contador de vueltas de bucle para pintar la bola, de lo contrario, si el contador está en 2 y ponemos la velocidad a 1, habrá que esperar 254 iteraciones hasta que la bola se vuelva a mover.

Ponemos A = 0, `LD A, $00`, y ponemos el contador de iteraciones para la bola a 0, `LD (countLoopBall), A`.

La etiqueta scanKeys_ctrl marca el punto donde empieza la rutina tal y como la tenemos ahora. La nueva implementación la vamos a hacer entre las etiquetas ScanKeys y scanKeys_speed:

```
ld    a, $f7
in    a, ($fe)
```

Cargamos la semifila 1-5 en A, `LD A, $F7`, y leemos del puerto del teclado, `IN A, ($FE)`.

```
bit   $00, a
jr    nz, scanKeys_2
```

Comprobamos si se ha pulsado el 1, `BIT $00, A`, y en caso de no haberlo pulsado saltamos a comprobar si se ha pulsado el 2, `JR NZ, scanKeys_2`.

Si se ha pulsado el 1, cambiamos la velocidad de la bola:

```
ld    a, (ballSetting)
and   $cf
or    $10
ld    (ballSetting), a
jr    scanKeys_speed
```

Cargamos la configuración de la bola en A, `LD A, (ballSetting)`, ponemos los bits de la velocidad a 0, `AND $CF`, ponemos la velocidad a 1, `OR $10`, cargamos la configuración en memoria, `LD (ballSetting), A`, y saltamos a poner a 0 el contador de iteraciones para la bola, `JR scanKeys_speed`.

La comprobación para el 2 y el 3 es muy parecida a la comprobación del 1, por lo que vemos el código completo y marcamos en rojo las diferencias:

```
scanKeys_2:
bit   $01, a
jr    nz, scanKeys_3
ld    a, (ballSetting)
and   $cf
or    $20
ld    (ballSetting), a
jr    scanKeys_speed
scanKeys_3:
bit   $02, a
jr    nz, scanKeys_ctrl
ld    a, (ballSetting)
and   $cf
or    $30
ld    (ballSetting), a
```

El aspecto final de la rutina, una vez modificada, queda de la siguiente manera:

```
; -----------------------------------------------------------------------------
; ScanKeys
; Escanea las teclas de control y devuelve las pulsadas.
; Salida:     D -> Teclas pulsadas.
;                    Bit 0 -> A pulsada 0/1.
;                    Bit 1 -> Z pulsada 0/1.
;                    Bit 2 -> 0 pulsada 0/1.
;                    Bit 3 -> O pulsada 0/1.
; Altera el valor de los registros AF y D.
; -----------------------------------------------------------------------------
ScanKeys:
ld      a, $f7              ; Carga en A la semifila 1-5
in      a, ($fe)            ; Lee el estado de la semifila
bit     $00, a              ; Comprueba si se ha pulsado el 1
jr      nz, scanKeys_2      ; Si no se ha pulsado salta
; Se ha pulsado; cambia la velocidad de la bola 1 (rápido)
```

```
ld      a, (ballSetting)      ; Carga la configuración de la bola en A
and     $cf                   ; Pone los bits de velocidad a 0
or      $10                   ; Pone los bits de velocidad a 1
ld      (ballSetting), a      ; Carga el valor en memoria
jr      scanKeys_speed        ; Salta para comprobar los controles
scanKeys_2:
bit     $01, a                ; Comprueba si se ha pulsado el 2
jr      nz, scanKeys_3        ; Si no se ha pulsado salta
; Se ha pulsado; cambia la velocidad de la bola 2 (medio)
ld      a, (ballSetting)      ; Carga la configuración de la bola en A
and     $cf                   ; Pone los bits de velocidad a 0
or      $20                   ; Pone los bits de velocidad a 2
ld      (ballSetting), a      ; Carga el valor en memoria
jr      scanKeys_speed        ; Salta para comprobar los controles
scanKeys_3:
bit     $02, a                ; Comprueba si se ha pulsado el 3
jr      nz, scanKeys_ctrl     ; Si no se ha pulsado salta
; Se ha pulsado; cambia la velocidad de la bola 3 (lento)
ld      a, (ballSetting)      ; Carga la configuración de la bola en A
or      $30                   ; Pone los bits de velocidad a 3
ld      (ballSetting), a      ; Carga el valor en memoria

scanKeys_speed:
ld      a, $00                ; Pone A = 0
ld      (countLoopBall), a    ; Pone el contador de iteraciones para la bola a 0
scanKeys_ctrl:
ld      d, $00                ; Pone el registro D a 0.

; Resto de la rutina desde ScanKeys_A
```

Es el momento de compilar y cargar en el emulador para comprobar cómo se comporta esta modificación. Si todo ha ido bien, podemos cambiar la velocidad de la bola.

Lo último que tenemos que hacer es contabilizar los puntos de cada jugador, para lo cual vamos a modificar la rutina MoveBall, en concreto moveBall_rightChg y moveBall_leftChg.

Estas rutinas se encargan de cambiar la dirección de la bola cuando llega al límite izquierdo o derecho. Vamos a implementar lo necesario para que marque los puntos.

El código nuevo lo vamos a poner justo debajo de dichas etiquetas, empezando por moveBall_rightChg:

```
moveBall_rightChg:
ld     hl, p1points
```

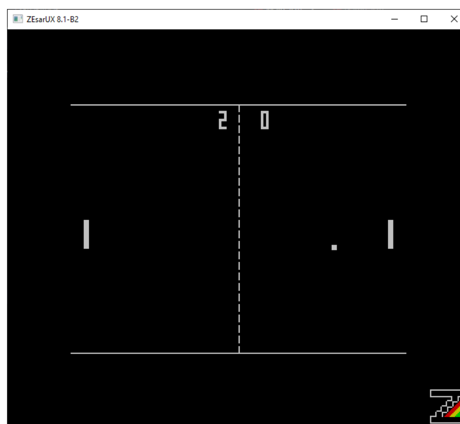
```
inc    (hl)
call   PrintPoints
```

Cargamos en HL la dirección de memoria donde se encuentra el marcador del jugador 1, `LD HL, p1points`, lo incrementamos, `INC (HL)`, y pintamos el marcador, `CALL PrintPoints`. El resto de la rutina se queda como estaba.

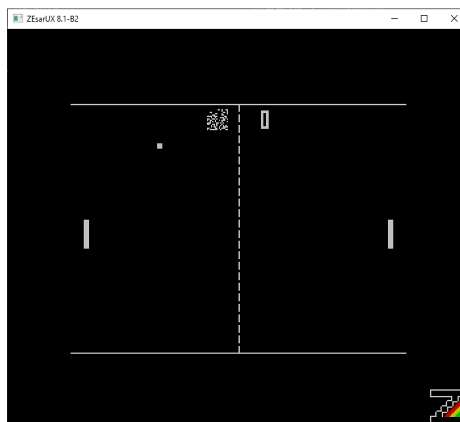
Las modificaciones en la etiqueta moveBall_leftChg son prácticamente las mismas:

```
moveBall_leftChg:
ld     hl, p2points
inc    (hl)
call   PrintPoints
```

Compilamos y cargamos en el emulador para ver los resultados:



Ya tenemos marcador, pero la partida continúa interminablemente y cuando pasamos de 15 puntos, empieza a pintar cosas sin sentido.



También podemos apreciar que cada vez va más lento. ¿Pero por qué? Pintamos el marcador en cada iteración, y para localizar el sprite del número a pintar hacemos un bucle, y no es lo mismo un bucle con 15 iteraciones como máximo, que un bucle con hasta 255 iteraciones. ¿A qué no? (En el capítulo 10 veremos la forma de implementar GetPointSprite, de tal manera que siempre tarde lo mismo y de paso nos ahorraremos 2 bytes y unos cuantos ciclos de reloj).

Lo que tenemos que hacer ahora es parar la partida cuando alguno de los dos jugadores llegue a 15 puntos; de igual manera vamos a implementar un modo de iniciar la partida, por ejemplo, pulsando el 5.

Al final del archivo Controls.asm vamos a implementar la rutina que espere a que se pulse el 5 para iniciar la partida:

```
WaitStart:
ld    a, $f7
in    a, ($fe)
bit   $04, a
jr    nz, WaitStart
ret
```

Cargamos en A la semifila 1-5, `LD A, $F7`, leemos el teclado, `IN A, ($FE)`, evaluamos si se ha pulsado el 5, `BIT $04, A`, y repetimos la operación hasta que se pulse, `JR NZ, WaitStart`.

El aspecto final de la rutina es:

```
; ------------------------------------------------------------------------------
; WaitStart.
; Espera que se pulse la tecla 5 para empezar la partida.
; Altera el valor de los registros AF.
; ------------------------------------------------------------------------------
WaitStart:
ld      a, $f7          ; Carga en A la semifila 1-5
in      a, ($fe)        ; Lee el teclado
bit     $04, a          ; Evalúa si se ha pulsado el 5
jr      nz, WaitStart   ; Bucle hasta que se pulse el 5

ret
```

Volvemos a Main.asm y después de la llamada a PrintPoints, ponemos la siguiente línea:

```
call  WaitStart
```

Si compilamos y cargamos en el emulador, hasta que no pulsemos el 5, no empezaremos la partida.

Pero con esto no es suficiente ya que la partida no finaliza cuando uno de los jugadores llega a 15 puntos.

Seguimos en Main.asm, pero esta vez al final de la rutina loop_continue, justo antes de `JR Loop`. Es aquí donde vamos a implementar el control de la puntuación:

```
ld    a, (p1points)
cp    $0f
```

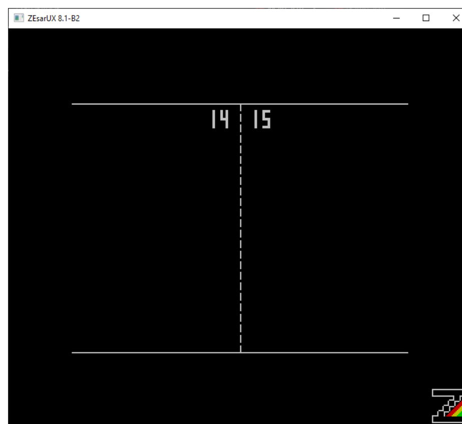
```
jr     z, Main
```

Cargamos la puntuación del jugador 1 en A, `LD A, (p1points)`, la comparamos con 15, `CP $0F`, y si es quince saltamos al inicio del programa, `JR Z, Main`.

Hacemos lo mismo con la puntuación del jugador 2:

```
ld     a, (p2points)
cp     $0f
jr     z, Main
```

Compilamos, cargamos en el emulador y comprobamos que cuando uno de los dos jugadores llega a quince puntos, la partida finaliza:



¿Pero qué pasa si volvemos a pulsar el 5? Ya no hay forma de iniciar la partida. En ningún momento ponemos el marcador a 0. Si dejamos pulsado el 5, veremos como a cada iteración del bucle, vuelve al inicio y se para.

Para solucionar esto, volvemos al inicio del archivo Main.asm, y justo después de la llamada a WaitStart, vamos a poner los marcadores a 0:

```
ld     a, ZERO
ld     (p1points), a
ld     (p2points), a
call   PrintPoints
```

Ponemos A = 0, `LD A, ZERO`, ponemos la puntuación del jugador 1 a 0, `LD (p1points), A`, ponemos la puntuación del jugador 2 a 0, `LD (p2points), A`, y pintamos el marcador, `CALL PrintPoints`. De esta manera, cada vez que iniciamos partida, ponemos los marcadores a 0 y los pintamos.

Compilamos y cargamos en el emulador para ver los resultados. Esto empieza a tomar forma.

Todavía nos quedan ajustes por realizar. Vamos a hacer que cuando se marque un tanto, la bola salga por el lado contrario, es decir, como si sacara el jugador que ha marcado.

Vamos a implementar una rutina para borrar la bola, otra para situarla en la parte derecha de la pantalla, y otra para situarla en la parte izquierda.

La rutina para borrar la bola la vamos a implementar en el archivo Video.asm, justo antes de la rutina Cls:

```
ClearBall:
ld    hl, (ballPos)
ld    a, l
and   $1f
cp    $10
jr    c, clearBall_continue
inc   l
```

Cargamos la posición de la bola en HL, `LD HL, (ballPos)`, cargamos la fila y la columna en A, `LD A, L`, nos quedamos con la columna, `AND $1F`, y lo comparamos con el centro de la pantalla, `CP $10`.

Si hay acarreo, solo puede estar en el margen izquierdo. Saltamos a borrar la bola, `JR C, clearBall_continue`. Si no salta, está en el margen derecho, pero la bola en realidad está pintada una columna más a la derecha (la bola se pinta en dos bytes/columnas); apuntamos HL a la columna dónde está pintada la bola, `INC L`.

```
clearBall_continue:
ld    b, $06
clearBall_loop:
ld    (hl), ZERO
call  NextScan

djnz  clearBall_loop

ret
```

Cargamos en B el número de scanlines que vamos a borrar, `LD B, $06`, borramos la posición apuntada por HL, `LD (HL), ZERO`, apuntamos HL al siguiente scanline, `CALL NextScan`, repetimos la operación hasta que B valga 0, `DJNZ clearBall_loop`, y salimos, `RET`.

El aspecto final de la rutina es el siguiente:

```
; -----------------------------------------------------------------------------
; Borra la bola.
; Altera el valor de los registros AF, B y HL.
; -----------------------------------------------------------------------------
ClearBall:
ld      hl, (ballPos)          ; Carga la posición de la bola en HL
ld      a, l                   ; Carga la fila y columna en A
and     $1f                    ; Se queda con la columna
cp      $10                    ; Lo compara con el centro de la pantalla
```

```
jr      c, clearBall_continue ; Si está a la izquierda salta
inc     l                     ; Incrementa la columna
clearBall_continue:
ld      b, $06                ; Bucle por 6 scanlines
clearBall_loop:
ld      (hl), ZERO            ; Borra el byte apuntado por HL
call    NextScan              ; Obtiene el scanline siguiente
djnz    clearBall_loop        ; Hasta que B = 0

ret
```

Las otras dos rutinas las vamos a implementar al final del archivo Game.asm:

```
SetBallLeft:
ld     hl, $4d60
ld     (ballPos), hl
ld     a, $01
ld     (ballRotation), a
ld     a, (ballSetting)
and    $bf
ld     (ballSetting), a
ret
```

Cargamos en HL la nueva posición de la bola, `LD HL, $4D60`, y lo cargamos en memoria, `LD (ballPos), HL`.

Cargamos la rotación de la bola en A, `LD A, $01`, y lo cargamos en memoria, `LD (ballRotation), A`.

Cargamos la configuración de la bola en A, `LD A, (ballSetting)`, ponemos la dirección horizontal hacia la derecha, `AND $BF`, lo cargamos en memoria, `LD (ballSetting), A`, y salimos, `RET`.

La rutina para posicionar la bola a la derecha es prácticamente igual; marcamos en rojo las diferencias sin entrar en detalle:

```
SetBallRight:
ld     hl, $4d7e
ld     (ballPos), hl
ld     a, $ff
ld     (ballRotation), a
ld     a, (ballSetting)
or     $40
ld     (ballSetting), a
```

```
ret
```

El aspecto final de las dos rutinas es el siguiente:

```
; -----------------------------------------------------------------------------
; Posiciona la bola a la izquierda.
; Altera el valor de los registros AF y HL.
; -----------------------------------------------------------------------------
SetBallLeft:
ld      hl, $4d60             ; Carga en HL la posición de la bola
ld      (ballPos), hl         ; Carga el valor en memoria
ld      a, $01                ; Carga 1 en A
ld      (ballRotation), a     ; Lo carga en memoria, Rotación = 1
ld      a, (ballSetting)      ; Carga en A la dirección y velocidad de la bola
and     $bf                   ; Pone la dirección horizontal hacia la derecha
ld      (ballSetting), a      ; Carga la nueva dirección de la bola en memoria

ret


; -----------------------------------------------------------------------------
; Posiciona la bola a la derecha.
; Altera el valor de los registros AF y HL.
; -----------------------------------------------------------------------------
SetBallRight:
ld      hl, $4d7e             ; Carga en HL la posición de la bola
ld      (ballPos), hl         ; Carga el valor en memoria
ld      a, $ff                ; Carga -1 en A
ld      (ballRotation), a     ; Lo carga en memoria, Rotación = -1
ld      a, (ballSetting)      ; Carga en A la dirección y velocidad de la bola
or      $40                   ; Pone la dirección horizontal hacia la izquierda
ld      (ballSetting), a      ; Carga la nueva dirección de la bola en memoria

ret
```

Para acabar con este paso, solo nos queda utilizar estas rutinas.

Vamos a modificar las rutinas moveBall_rightChg y moveBall_leftChg del archivo Game.asm.

En la rutina moveBall_rightChg, borramos las líneas que hay entre CALL PrintPoints y JR moveBall_end, y las sustituimos por:

```
call  ClearBall
call  SetBallLeft
```

El aspecto final de la rutina es el siguiente:

```
moveBall_rightChg:
; Ha llegado al límite derecho, ¡PUNTO!
ld      hl, p1points   ; Carga en HL la dirección de la puntuación del jugador 1
inc     (hl)           ; Lo incrementa
call    PrintPoints    ; Pinta el marcador
call    ClearBall      ; Borra la bola
call    SetBallLeft    ; Pone la bola a la izquierda
jr      moveBall_end   ; Fin de la rutina
```

En la rutina moveBall_leftChg, borramos las líneas que hay entre CALL PrintPoints y la etiqueta moveBall_end, y las sustituimos por:

```
call  ClearBall
call  SetBallRight
```

El aspecto final de la rutina es el siguiente:

```
moveBall_leftChg:
; Ha llegado al límite izquierdo, ¡PUNTO!
ld      hl, p2points   ; Carga en HL la dirección de la puntuación del jugador 2
inc     (hl)           ; Lo incrementa
call    PrintPoints    ; Pinta el marcador
call    ClearBall      ; Borra la bola
call    SetBallRight   ; Pone la bola a la derecha
```

Compilamos, cargamos en el emulador, y ya podemos empezar a jugar nuestras primeras partidas a dos jugadores, aunque todavía quedan cosas por hacer.

Paso 9: cambio de dirección/velocidad de la bola al golpear la pala

En este paso, vamos a prescindir de parte de lo que hemos implementado en el paso anterior. La velocidad de la bola va a cambiar dependiendo de con qué parte de la pala colisione.

Creamos la carpeta Paso09 y copiamos los archivos Controls.asm, Game.asm, Main.asm, Sprite.asm y Video.asm desde la carpeta Paso08.

Lo primero que vamos a hacer es quitar la posibilidad de cambiar la velocidad de la bola con las teclas del 1 al 3.

Abrimos el archivo Controls.asm y en la rutina ScanKeys, borramos todas las líneas hasta la etiqueta scanKeys_ctrl, quedando el inicio de la rutina de la siguiente manera:

```
ScanKeys:
ld    d, $00


scanKeys_A:
```

Si compilamos y cargamos en el emulador, vemos que la velocidad de la bola no cambia.

Vamos a añadir nuevas constantes y variables en el archivo Sprite.asm, para poder controlar la inclinación de la bola. También vamos a cambiar los sprites de las palas; ambas van a dibujar cuatro píxeles, pero en ambos casos dibujaremos los más cercanos al centro de la pantalla.

Añadimos las constantes que indican la rotación a asignar a la bola cuando se produce la colisión con la pala:

```
CROSS_LEFT_ROT:   EQU   $ff
CROSS_RIGHT_ROT:  EQU   $01
```

Añadimos la posición inicial de la bola, y el número acumulado de movimientos que debe llevar la bola para cambiar la posición Y. Este último dato lo vamos a usar para cambiar la inclinación de la bola:

```
BALLPOS_INI:      EQU   $4850
ballMovCount:     db    $00
```

Cambiamos la configuración inicial de la bola y la documentación (comentarios) de la misma:

```
; Velocidad y dirección de la bola.
; bits 0 a 3:  Movimientos de la bola para que cambie la posición Y.
;              Valores f = semiplano, 2 = semi diagonal, 1 = diagonal
; bits 4 y 5:  Velocidad de la bola: 1 muy rápido, 2 (rápido) y 3 (lento)
; bit 6:       Dirección X: 0 derecha / 1 izquierda
; bit 7:       Dirección Y: 0 arriba / 1 abajo
ballSetting:
db     $31    ; 0011 0001
```

Según la nueva configuración, la bola inicialmente se mueve hacia la derecha y hacia arriba, con una velocidad lenta, y en cada movimiento cambia la posición Y.

Añadimos distintos sprites para las palas y eliminamos el anterior:

```
PADDLE:       EQU    $3e
PADDLE1:      EQU    $0f
PADDLE2:      EQU    $f0
```

Por último, añadimos las posiciones iniciales de las palas:

```
PADDLE1POS_INI:    EQU      $4861
PADDLE2POS_INI:    EQU      $487e
```

Hemos añadido sprites distintos para cada pala y eliminado la constante que usábamos para pintar las palas; si compilamos, nos dará errores.

Vamos a solucionar esos errores modificando la rutina PrintPaddle de Video.asm.

La rutina PrintPaddle recibe en el registro HL la posición de la pala. En el registro C recibirá el sprite de la pala.

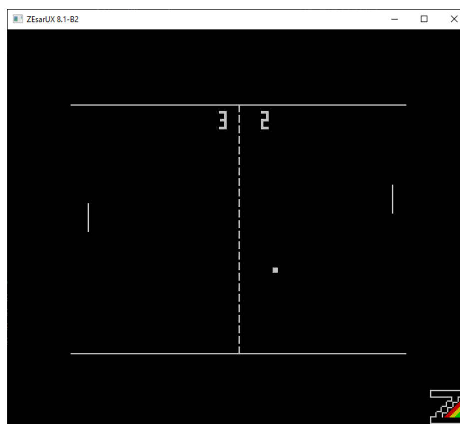
Modificamos la línea justo debajo de la etiqueta printPaddle_loop:

```
ld     (hl), PADDLE
```

y la dejamos como sigue:

```
ld     (hl), c
```

Compilamos, y aunque no da ningún error, al cargar en el emulador vemos que los resultados no son los deseados:



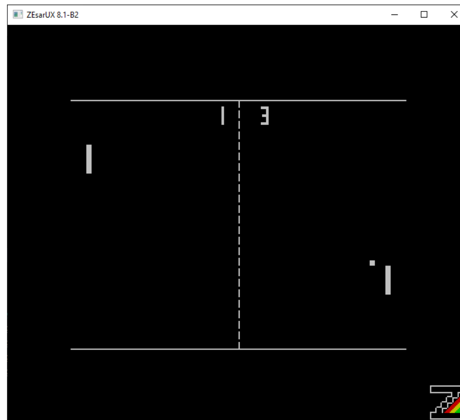
La pala que pinta no se corresponde con el sprite que hemos definido. Esto es debido a que no hemos cargado en C cual es el sprite que debe pintar.

Abrimos el archivo Main.asm, y buscamos la etiqueta loop_continue. A partir de la línea 5 es donde imprimimos las palas, cargando el HL la posición de la pala y llamando al pintado de la misma. Antes de llamar al pintado de la pala, debemos especificar qué sprite debe pintar.

Este es el aspecto una vez hecha la modificación:

```
ld     hl, (paddle1pos)
ld     c, PADDLE1
call   PrintPaddle
ld     hl, (paddle2pos)
ld     c, PADDLE2
call   PrintPaddle
```

Compilamos, abrimos en el emulador, y comprobamos que las palas se vuelven a pintar bien:



Aprovechando que estamos en Main.asm, vamos a cambiar un comportamiento del que quizás no os habéis percatado. Cuando se acaba un partido, y al iniciar otro, las palas siguen en la misma posición donde estaban al acabar el partido anterior, y la bola sale desde el campo del jugador que anotó el último punto.

Para modificar este comportamiento, vamos a añadir las siguientes líneas antes de la etiqueta Loop:

```
ld     hl, BALLPOS_INI
ld     (ballPos), hl
ld     hl, PADDLE1POS_INI
ld     (paddle1pos), hl
ld     hl, PADDLE2POS_INI
ld     (paddle2pos), hl
```

Con estas líneas situamos la bola y las palas en sus posiciones iniciales.

Si compilamos, vemos que nos da un error:

```
ERROR on line 68 of file Main.asm
ERROR: Relative jump out of range
```

Este error es debido a que, al ir añadiendo líneas, tenemos algún JR que está fuera de rango. JR solo puede saltar 127 bytes hacia adelante o 128 hacia atrás, y tenemos algún JR que salta a alguna dirección fuera de este rango.

En concreto, tenemos al final del archivo Main.asm, dos JR Main y un JR Loop. Sustituimos estos tres JR por JP, y solucionamos el error. JP ocupa un byte más que JR, por lo que nuestro programa acaba de crecer 3 bytes, pero hemos reducido 6 ciclos de reloj.

Compilamos, cargamos en el emulador y comprobamos que al acabar la partida e iniciar otra, tanto la bola como las palas vuelven a su posición inicial.

Vamos a implementar el cambio de velocidad, inclinación y dirección de la bola al colisionar con las palas.

Abrimos el archivo Game.asm y buscamos la etiqueta checkBallCross_left. Tres líneas por encima encontramos:

```
ld    a, $ff
```

Modificamos esta línea y la dejamos como sigue:

```
ld    a, CROSS_LEFT_ROT
```

Buscamos la etiqueta CheckCrossX. Tres líneas por encima encontramos:

```
ld    a, $01
```

Modificamos esta línea y la dejamos como sigue:

```
ld    a, CROSS_RIGHT_ROT
```

Hemos cambiado los valores por constantes, para si en un futuro hay que cambiar los valores, tenerlos mejor localizados.

El siguiente paso es cambiar la configuración de la bola, dependiendo de en qué parte de la pala colisiona.

Vamos a dividir la pala en 5 partes. Dependiendo de dónde colisione la bola, el comportamiento será:

Zona de golpeo	Dirección vertical	Inclinación	Velocidad
1/5	Arriba	Diagonal	3 (lento)
2/5	Arriba	Semi diagonal	2 (normal)
3/5	No cambia	Semi plano	1 (rápido)
4/5	Abajo	Semi diagonal	2 (normal)
5/5	Abajo	Diagonal	3 (lento)

Localizamos la etiqueta CheckCrossY, nos vamos a la penúltima línea, `XOR A`, e implementamos justo antes de ella:

```
ld    a, c
sub   $15
ld    c, a
ld    a, b
add   a, $04
ld    b, a
```

Cuando llegamos a este punto, en C tenemos la posición del penúltimo scanline de la pala, y en B la posición de la bola. Ambas posiciones están en formato TTLLLSSS.

Cargamos en A la posición del penúltimo scanline de la pala, `LD A, C`, nos posicionamos en el primero, `SUB $15`, y volvemos a cargar el valor en C, `LD C, A`.

Cargamos en A la posición de la bola, `LD A, B`, nos posicionamos en la parte baja de la bola, `ADD A, $04`, y volvemos a cargar el valor en B, `LD B, A`.

A partir de aquí implementamos el cambio de comportamiento, dependiendo del lugar de colisión de la bola:

```
checkCrossY_1_5:
ld    a, c
add   a, $04
cp    b
jr    c, checkCrossY_2_5
```

Cargamos la posición vertical de la pala en A, `LD A, C`, nos posicionamos en el último scanline de la primera parte, `ADD A, $04`, y lo comparamos con la posición de la bola, `CP B`. Si hay acarreo, la bola está más abajo y salta a comprobar la siguiente parte, `JR C, checkCrossY_2_5`.

Si no hay acarreo, la bola ha colisionado en esta parte y tenemos que cambiar su configuración:

```
ld    a, (ballSetting)
and   $40
or    $31
jr    checkCrossY_end
```

Cargamos la configuración de la bola en A, `LD A, (ballSetting)`, nos quedamos con la dirección horizontal (ya viene calculada), `AND $40`, y ponemos dirección vertical hacia arriba, velocidad 3 e inclinación diagonal, `OR $31`. Saltamos al final de la rutina, `JR checkCrossY_end`.

Si la bola no ha colisionado con la primera parte de la pala, comprobamos si lo ha hecho con la segunda:

```
checkCrossY_2_5:
ld    a, c
add   a, $09
cp    b
jr    c, checkCrossY_3_5
```

Cargamos la posición vertical de la pala en A, `LD A, C`, nos posicionamos en el último scanline de la segunda parte, `ADD A, $09`, y lo comparamos con la posición de la bola, `CP B`. Si hay acarreo, la bola está más abajo y salta a comprobar la siguiente parte, `JR C, checkCrossY_3_5`.

Si no hay acarreo, la bola ha colisionado en esta parte y tenemos que cambiar su configuración:

```
ld    a, (ballSetting)
and   $40
or    $22
jr    checkCrossY_end
```

Cargamos la configuración de la bola en A, `LD A, (ballSetting)`, nos quedamos con la dirección horizontal (ya viene calculada), `AND $40`, y ponemos dirección vertical hacia arriba, velocidad 2 e inclinación semi diagonal, `OR $22`. Saltamos al final de la rutina, `JR checkCrossY_end`.

Si la bola no ha colisionado con la segunda parte de la pala, comprobamos si lo ha hecho con la tercera:

```
checkCrossY_3_5:
ld    a, c
add   a, $0d
cp    b
jr    c, checkCrossY_4_5
```

Cargamos la posición vertical de la pala en A, `LD A, C`, nos posicionamos en el último scanline de la tercera parte, `ADD A, $0D`, y lo comparamos con la posición de la bola, `CP B`. Si hay acarreo, la bola está más abajo y salta a comprobar la siguiente parte, `JR C, checkCrossY_4_5`.

Si no hay acarreo, la bola ha colisionado en esta parte y tenemos que cambiar su configuración:

```
ld    a, (ballSetting)
and   $c0
or    $1f
jr    checkCrossY_end
```

Cargamos la configuración de la bola en A, `LD A, (ballSetting)`, nos quedamos con la dirección horizontal y con la vertical (ya vienen calculadas), `AND $C0`, y ponemos velocidad 1 e inclinación semi plana, `OR $1F`. Saltamos al final de la rutina, `JR checkCrossY_end`.

Si la bola no ha colisionado con la tercera parte de la pala, comprobamos si lo ha hecho con la cuarta:

```
checkCrossY_4_5:
ld    a, c
add   a, $11
cp    b
jr    c, checkCrossY_5_5
```

Cargamos la posición vertical de la pala en A, `LD A, C`, nos posicionamos en el último scanline de la cuarta parte, `ADD A, $11`, y lo comparamos con la posición de la bola, `CP B`. Si hay acarreo, la bola está más abajo y salta a comprobar la siguiente parte, `JR C, checkCrossY_5_5`.

Si no hay acarreo, la bola ha colisionado en esta parte y tenemos que cambiar su configuración:

```
ld    a, (ballSetting)
and   $40
or    $a2
jr    checkCrossY_end
```

Cargamos la configuración de la bola en A, `LD A, (ballSetting)`, nos quedamos con la dirección horizontal (ya viene calculada), `AND $40`, y ponemos dirección vertical hacia abajo, velocidad 2 e inclinación semi diagonal, `OR $A2`. Saltamos al final de la rutina, `JR checkCrossY_end`.

Si la bola no ha colisionado con la cuarta parte de la pala, lo ha hecho con la quinta:

```
checkCrossY_5_5:
ld    a, (ballSetting)
and   $40
or    $b1
```

Cargamos la configuración de la bola en A, `LD A, (ballSetting)`, nos quedamos con la dirección horizontal (ya viene calculada), `AND $40`, y ponemos dirección vertical hacia abajo, velocidad 3 e inclinación diagonal, `OR $B1.`

Por último, justo por encima de XOR A, vamos a añadir la etiqueta de fin de función a la que hemos estado haciendo referencia, y vamos a cargar la nueva configuración de la bola en memoria:

```
checkCrossY_end:
ld    (ballSetting), a
```

Después de XOR A, vamos a poner el contador de movimientos de la bola a 0:

```
ld    (ballMovCount), a
```

El aspecto final de la rutina es el siguiente:

```
; -----------------------------------------------------------------------------
; Evalúa si la bola colisiona en el eje Y con la pala.
; En el caso de colisionar, actualiza la configuración de la bola.
; Entrada:     HL -> Posición de la pala
; Salida:      Z -> Colisiona.
;              NZ -> No colisiona.
; Altera el valor de los registros AF, BC y HL.
```

```
; --------------------------------------------------------------------------------
CheckCrossY:
call    GetPtrY               ; Obtiene la posición vertical de la pala (TTLLLSSS)
; La posición devuelta apunta al primer scanline de la pala que está a 0
; apunta al siguiente
inc     a
ld      c, a                  ; Carga el valor en C
ld      hl, (ballPos)         ; Carga en HL la posición de la bola
call    GetPtrY               ; Obtiene la posición vertical de la bola (TTLLLSSS)
ld      b, a                  ; Carga el valor en B
; Comprueba si la bola pasa por encima de la pala
; La bola está compuesta de 1 scanline a 0, 4 a $3c y otro a 0
; La posición apunta al 1er scanline, y se comprueba la colisión con el 5º
add     a, $04                ; Apunta la posición de la bola al 5º scanline
sub     c                     ; Resta a la posición de la bola, la posición de la pala
ret     c                     ; Si hay acarreo sale porque la bola pasa por encima
; Comprueba si la bola pasa por debajo de la pala
ld      a, c                  ; Carga la posición vertical de la pala en A
add     a, $16                ; Le suma 22 para apuntar al penúltimo scanline,
                              ; último que no es 0
ld      c, a                  ; Lo vuelve a cargar en C
ld      a, b                  ; Carga la posición vertical de la bola
inc     a                     ; Le suma 1 para apuntar el scanline 1, primero que no es 0
sub     c                     ; Resta a la posición de la bola, la posición de la pala
ret     nc                    ; Si no hay acarreo la bola pasa por debajo
                              ; de la pala o colisiona en el último scanline.
                              ; En este último caso se activa el flag Z

; Dependiendo de donde sea la colisión, se asigna grado de inclinación
; y velocidad a la bola
ld      a, c                  ; Carga la posición del penúltimo scanline de la pala en A
sub     $15                   ; Lo vuelve a posicionar en el primero
ld      c, a                  ; Carga el valor en C

ld      a, b                  ; Carga en A la posición de la bola
add     a, $04                ; Se posiciona en la parte baja de la bola
ld      b, a                  ; Carga el valor en B

checkCrossY_1_5:
ld      a, c                  ; Carga la posición vertical de la pala en A
add     a, $04                ; Se posiciona en el último scanline de 1/5
```

```
cp      b                     ; Lo compara con la posición de la bola
jr      c, checkCrossY_2_5    ; La bola está más abajo, salta
ld      a, (ballSetting)      ; Carga la configuración de la bola en A
and     $40                   ; Se queda con la dirección horizontal
or      $31                   ; Hacia arriba, velocidad 3 e inclinación diagonal
jr      checkCrossY_end       ; Fin de la rutina

checkCrossY_2_5:
ld      a, c                  ; Carga la posición vertical de la pala en A
add     a, $09                ; Se posiciona en el último byte de 2/5
cp      b                     ; Lo compara con la posición de la bola
jr      c, checkCrossY_3_5    ; La bola está más abajo, salta
ld      a, (ballSetting)      ; Carga la configuración de la bola en A
and     $40                   ; Se queda con la dirección horizontal
or      $22                   ; Hacia arriba, velocidad 2 e inclinación semi diagonal
jr      checkCrossY_end       ; Fin de la rutina

checkCrossY_3_5:
ld      a, c                  ; Carga la posición vertical de la pala en A
add     a, $0d                ; Se posiciona en el último byte de 3/5
cp      b                     ; Lo compara con la posición de la bola
jr      c, checkCrossY_4_5    ; La bola está más abajo, salta
ld      a, (ballSetting)      ; Carga la configuración de la bola en A
and     $c0                   ; Se queda con la dirección horizontal y vertical
or      $1f                   ; Hacia arriba/abajo, velocidad 1 e inclinación semi plano
jr      checkCrossY_end       ; Fin de la rutina

checkCrossY_4_5:
ld      a, c                  ; Carga la posición vertical de la pala en A
add     a, $11                ; Se posiciona en el último byte de 4/5
cp      b                     ; Lo compara con la posición de la bola
jr      c, checkCrossY_5_5    ; La bola está más abajo, salta
ld      a, (ballSetting)      ; Carga la configuración de la bola en A
and     $40                   ; Se queda con la dirección horizontal y vertical
or      $a2                   ; Hacia abajo, velocidad 2 e inclinación semi diagonal
jr      checkCrossY_end       ; Fin de la rutina

checkCrossY_5_5:
ld      a, (ballSetting)      ; Carga la configuración de la bola en A
and     $40                   ; Se queda con la dirección horizontal
or      $b1                   ; Hacia abajo, velocidad 3 e inclinación diagonal
```

```
; Hay colisión
checkCrossY_end:
ld      (ballSetting), a      ; Carga en memoria la configuración actual de la bola
xor     a                     ; Activa el flag Z y pone A = 0
ld      (ballMovCount), a     ; Pone el contador de movimientos de la bola a 0
ret
```

Compilamos, cargamos en el emulador y vemos los resultados.

Vemos que la velocidad sí cambia dependiendo de dónde colisiona la bola, pero no la inclinación. Además, al marcar un tanto, la velocidad no se reinicia, lo cual hace que sea muy difícil seguir jugando si la bola va a la velocidad máxima.

¿Por qué cambia la velocidad, pero no la inclinación?

Si hacemos memoria, en el paso anterior implementamos la posibilidad de cambiar la velocidad de la bola con las teclas del 1 al 3. De hecho, este paso lo iniciamos avisando de que íbamos a prescindir de esta implementación, pero de lo que no se ha prescindido es del cambio que hicimos en Main.asm para tener en cuenta la velocidad de la bola que marque la configuración; por eso la velocidad cambia.

Nos falta la implementación para tener en cuenta la inclinación, y para que cuando se marca un punto, velocidad e inclinación de la bola se reinicien.

Vamos a empezar con el cambio de inclinación. Seguimos en el archivo Game.asm, implementando la rutina que va a cambiar la posición Y de la bola. La vamos a implementar después del RET de la etiqueta moveBall_end:

```
MoveBallY:
ld    a, (ballSetting)
and   $0f
ld    d, a
```

Cargamos en A la configuración de la bola, `LD A, (ballSetting)`, nos quedamos con la inclinación, `AND $0F`, y cargamos el valor en D, `LD A, D`.

```
ld    a, (ballMovCount)
inc   a
ld    (ballMovCount), a
cp    d
ret   nz
```

Cargamos los movimientos de la bola en A, `LD A, (ballMovCount)`, lo incrementamos en 1, `INC A`, cargamos el valor en memoria, `LD (ballMovCount), A`, y lo comparamos con D, que contiene el número de movimientos necesarios para cambiar la posición Y de la bola, `CP D`. Si no son iguales, no se ha llegado al valor necesario y salimos, `RET NZ`.

```
xor   a
```

```
ld     (ballMovCount), a
ret
```

Si hemos llegado al valor, ponemos A = 0 y activamos el flag Z, `XOR A`, ponemos a 0 los movimientos acumulados de la bola, `LD (ballMovCount), A`, y salimos, `RET`. Al activar el flag Z se indica, a quien llame, que se debe cambiar la posición Y de la bola.

El aspecto final de la rutina es el siguiente:

```
; -----------------------------------------------------------------------------
; Cambia la posición Y de la bola
; Altera el valor de los registros AF y D.
; -----------------------------------------------------------------------------
MoveBallY:
ld      a, (ballSetting)      ; Carga en A la configuración de la bola
and     $0f                   ; Se queda con la inclinación
ld      d, a                  ; Carga el valor en D

ld      a, (ballMovCount)     ; Carga en A los movimientos acumulados de la bola
inc     a                     ; Incrementa A
ld      (ballMovCount), a     ; Carga el valor en memoria
cp      d                     ; Lo compara con la inclinación
ret     nz                    ; Si no son iguales, sale. No se cambia la posición

; La posición debe cambiar
xor     a                     ; Pone A = 0 y activa el flag Z
ld      (ballMovCount), a     ; Pone los movimientos acumulados de la bola a 0

ret
```

Localizamos la etiqueta moveBall_up, y entre las líneas `JR Z, moveBall_upChg` y `CALL PreviousScan`, añadimos las siguientes líneas:

```
call   MoveBallY
jr     nz, moveBall_x
```

Evaluamos si se tiene que cambiar la posición Y de la bola, `CALL MoveBallY`, y de no ser así salta, `JR NZ, moveBall_x`.

Localizamos la etiqueta moveBall_down, y entre las líneas `JR Z, moveBall_downChg` y `CALL NextScan`, añadimos las siguientes líneas:

```
call   MoveBallY
jr     nz, moveBall_x
```

Evaluamos si se tiene que cambiar la posición Y de la bola, `CALL MoveBallY`, y de no ser así salta, `JR NZ, moveBall_x`.

Compilamos, cargamos en el emulador, y comprobamos que ahora cambian la inclinación y la velocidad.

Por último, vamos a hacer que cuando se marque un punto, se reinicien la velocidad y la inclinación de la bola.

Localizamos la rutina SetBallLeft, eliminamos la línea `AND $BF`, y la sustituimos por las siguientes:

```
and    $bf
and    $80
or     $31
```

Se queda con la dirección Y, `AND $80`, y pone dirección horizontal hacia la derecha, velocidad 3 e inclinación diagonal, `OR $31`.

Antes de la instrucción `RET`, añadimos las siguientes líneas:

```
ld     a, $00
ld     (ballMovCount), a
```

Ponemos A = 0, `LD A, $00`, y ponemos los movimientos de la bola a 0, `LD (ballMovCount), A`.

Localizamos la rutina SetBallRight y eliminamos la línea `OR $40` y la sustituimos por las siguientes:

```
or     $40
and    $80
or     $71
```

Se queda con la dirección Y, `AND $80`, y pone dirección horizontal hacia la izquierda, velocidad 3 e inclinación diagonal, `OR $11`.

Antes de la instrucción `RET`, añadimos las siguientes líneas:

```
ld     a, $00
ld     (ballMovCount), a
```

Ponemos A = 0, `LD A, $00`, y ponemos los movimientos de la bola a 0, `LD (ballMovCount), A`.

El aspecto final de ambas rutinas es el siguiente:

```
; -----------------------------------------------------------------------------
; Posiciona la bola a la izquierda.
; Altera el valor de los registros AF y HL.
; -----------------------------------------------------------------------------
SetBallLeft:
ld      hl, $4d60              ; Carga en HL la posición de la bola
ld      (ballPos), hl          ; Carga el valor en memoria
```

```
ld      a, $01                ; Carga 1 en A
ld      (ballRotation), a     ; Lo carga en memoria Rotación = 1
ld      a, (ballSetting)      ; Carga en A la dirección y velocidad de la bola
and     $80                   ; Se queda con la dirección Y
or      $31                   ; Pone dirección X a derecha, velocidad 3
                              ; e inclinación diagonal
ld      (ballSetting), a      ; Carga la nueva dirección de la bola en memoria
ld      a, $00
ld      (ballMovCount), a

ret

; ------------------------------------------------------------------------------
; Posiciona la bola a la derecha.
; Altera el valor de los registros AF y HL.
; ------------------------------------------------------------------------------
SetBallRight:
ld      hl, $4d7e             ; Carga en HL la posición de la bola
ld      (ballPos), hl         ; Carga el valor en memoria
ld      a, $ff                ; Carga -1 en A
ld      (ballRotation), a     ; Lo carga en memoria Rotación = -1
ld      a, (ballSetting)      ; Carga en A la dirección y velocidad de la bola
and     $80                   ; Se queda con la dirección Y
or      $71                   ; Pone dirección X a izquierda, velocidad 3
                              ; e inclinación diagonal
ld      (ballSetting), a      ; Carga la nueva dirección de la bola en memoria
ld      a, $00
ld      (ballMovCount), a

ret
```

Compilamos, cargamos en el emulador y vemos los resultados, que deben ser los esperados, aunque la bola va algo lenta, ¿o no?

¿Os habéis fijado que cuando la bola golpea en la parte más baja de la pala no cambia ni dirección vertical, ni inclinación, ni velocidad? ¿Sabéis a qué se debe? En el capítulo 10 veremos el por qué.

Paso 10: sonido y optimización

Sí, la bola va algo lenta. Esto es debido, en gran parte, a que el marcador se repinta en cada iteración del bucle principal, lo cual no es necesario.

El marcador solo se debería repintar cuando es borrado por la bola. Modificando este aspecto, vamos a ganar velocidad en la bola, ya que el tiempo de proceso en cada iteración del bucle principal se va a reducir.

Como es costumbre, creamos la carpeta Paso10 y copiamos los archivos Controls.asm, Game.asm, Main.asm, Sprite.asm y Video.asm desde la carpeta Paso09.

Lo primero es localizar el área de la pantalla dónde la bola borra el marcador, definiendo una serie de constantes en el archivo Sprite.asm, justo debajo de la constante POINTS_P2:

```
POINTS_X1_L:        EQU      $0c
POINTS_X1_R:        EQU      $0f
POINTS_X2_L:        EQU      $10
POINTS_X2_R:        EQU      $13
POINTS_Y_B:         EQU      $14
```

El significado de estas constantes, en orden de aparición, es:

- Columna en la que la bola empieza a borrar el marcador del jugador 1 por la izquierda.
- Columna en la que la bola empieza a borrar el marcador del jugador 1 por la derecha.
- Columna en la que la bola empieza a borrar el marcador del jugador 2 por la izquierda.
- Columna en la que la bola empieza a borrar el marcador del jugador 2 por la derecha.
- Tercio, línea y scanline en la que la bola empieza a borrar el marcador por la parte de abajo.

Una vez que hemos definido estas constantes, vamos a modificar las rutinas PrintPoints y ReprintPoints del archivo Video.asm, empezando por localizar la etiqueta printPoint_print, que vamos a sustituir por PrintPoint.

Dentro de la rutina PrintPoints, hay tres llamadas a printPoint_print, que vamos a sustituir por PrintPoint.

Compilamos, cargamos en el emulador y comprobamos que no hemos roto nada.

El siguiente paso es modificar la rutina ReprintPoints. En realidad, no la vamos a modificar, la vamos a borrar y a volver a implementar:

```
ReprintPoints:
ld     hl, (ballPos)
call   GetPtrY
cp     POINTS_Y_B
ret    nc
```

Cargamos la posición de la bola en HL, `LD HL, (ballPos)`, obtenemos tercio, línea y scanline de la posición de la bola, `CALL GetPtrY`, y lo comparamos con la posición donde la

bola empieza a borrar el marcador desde abajo, `CP POINTS_Y_B`. Si no hay acarreo, la bola pasa por debajo del marcador y sale, `RET NC`.

Si hay acarreo, según la coordenada Y de la bola, ésta podría borrar el marcador.

```
ld    a, l
and   $1f
cp    POINTS_X1_L
ret   c
jr    z, reprintPoint_1_print
```

Cargamos la línea y columna de la posición de la bola en A, `LD A, L`, nos quedamos con la columna, `AND $1F`, y lo comparamos con la coordenada X en la que se empieza a borrar el marcador del jugador 1 por la izquierda, `CP POINTS_X1_L`. Si hay acarreo, la bola pasa por la izquierda del marcador y sale, `RET C`. Si las dos coordenadas coinciden, la bola va a borrar el marcador del jugador 1, y salta para repintarlo, `JR Z, reprintPoint_1_print`.

Si no hemos salido, ni saltado, seguimos con las comprobaciones:

```
cp    POINTS_X2_R
jr    z, reprintPoint_2_print
ret   nc
```

Comparamos la coordenada X donde está la bola con la coordenada donde se empieza a borrar el marcador del jugador 2 por la derecha, `CP POINT_X2_R`. Si son iguales, salta a repintar el marcador del jugador 2, `JR Z, reprintPoint_2_print`. Si no salta y no hay acarreo, la bola pasa por la derecha y sale, `RET NC`.

Si no hemos saltado, ni hemos salido, seguimos con las comprobaciones:

```
reprintPoint_1:
cp    POINTS_X1_R
jr    c, reprintPoint_1_print
jr    nz, reprintPoint_2
```

Comparamos la coordenada X de la bola con la coordenada donde la bola empieza a borrar el marcador del jugador 1 por la derecha, `CP POINTS_X1_R`. Si hay acarreo, está borrando el marcador del jugador 1 y salta para repintarlo, `JR C, reprintPoint_1_print`. Si no son la misma coordenada, pasa por la derecha del marcador del jugador 1 y salta para comprobar si borra el marcador del jugador 2, `JR NZ, reprintPoint_2`.

Si está borrando el marcador del jugador 1, lo repinta:

```
reprintPoint_1_print:
ld    a, (p1points)
call  GetPointSprite
push  hl
```

Cargamos los puntos del jugador 1 en A, `LD A, (p1points)`, obtenemos la dirección del sprite a pintar, `CALL GetPointSprite`, y preservamos el valor, `PUSH HL`.

Empezamos pintando el primer dígito, las decenas:

```
ld    e, (hl)
inc   hl
ld    d, (hl)
ld    hl, POINTS_P1
call  PrintPoint
pop   hl
```

Cargamos en E la parte baja de la dirección de memoria del sprite del primer dígito, `LD E, (HL)`, apuntamos HL a la parte alta de la dirección, `INC HL`, cargamos la parte alta de la dirección en D, `LD D, (HL)`, cargamos en HL la dirección de memoria donde se pinta el marcador del jugador 1, `LD HL, POINTS_P1`, pintamos el primer dígito, `CALL PrintPoint`, y recuperamos el valor de HL, `POP HL`.

Terminamos pintando el segundo dígito:

```
inc   hl
inc   hl
ld    e, (hl)
inc   hl
ld    d, (hl)
ld    hl, POINTS_P1
inc   l
jr    PrintPoint
```

Apuntamos HL a la dirección de memoria del sprite del segundo dígito, `INC HL, INC HL`, cargamos la parte baja de la dirección en E, `LD E, (HL)`, apuntamos HL a la parte alta de la dirección, `INC HL`, la cargamos en D, `LD D, (HL)`, cargamos en HL la dirección de memoria dónde se pinta el marcador del jugador 1, `LD HL, POINTS_P1`, apuntamos HL a la dirección donde se pinta el segundo dígito, `INC L`, y pintamos el dígito y salimos, `JR PrintPoint`.

Posiblemente os estaréis preguntando, ¿cómo salimos? ¡Si no hay ningún RET!

Estaréis pensando que en lugar de JR PrintPoint, tendríamos que haber puesto:

```
call  PrintPoint
ret
```

Y efectivamente esto funciona, pero no es necesario. Además, de la forma que lo hemos implementado, ahorramos tiempo de proceso y bytes.

La última instrucción de PrintPoint es un RET, y este es el RET que utilizamos para salir, por eso podemos poner JR en lugar de CALL y RET.

Por eso, y porque no tenemos nada que tengamos que recuperar de la pila. Si hubiéramos dejado algo en la pila, los resultados serían impredecibles.

A continuación, podemos ver la diferencia de ciclos de reloj y bytes entre hacerlo de una manera o de otra:

Instrucción	Ciclos de reloj	Bytes
CALL PrintPoint	17	3
RET	10	1
JR PrintPoint	12	2

Nos hemos ahorrado 15 ciclos de reloj y 2 bytes.

También hemos cambiado la forma de repintar. Antes repintábamos los marcadores haciendo OR con lo que hubiera pintado en esa zona, y ahora directamente pintamos el marcador. El resultado es que al pintar el marcador borramos la bola, lo que puede producir algún parpadeo. Como estos parpadeos también existen en el arcade original, lo dejamos así...o podéis cambiarlo.

Vamos ahora a ver cómo repintamos el marcador del jugador 2:

```
reprintPoint_2:
cp     POINTS_X2_L
ret    c
```

En este punto, solo hay que comprobar que la bola no esté pasando entre los marcadores sin borrarlos. Comparamos con el límite izquierdo del marcador del jugador 2, `CP POINTS_X2_L`, y si hay acarreo sale pues pasa por la izquierda, `RET C`.

Si no ha salido, hay que repintar el marcador del jugador 2, lo cual es casi idéntico a lo que hacemos con el marcador del jugador 1, por lo que se marcan las diferencias sin entrar en el detalle:

```
reprintPoint_2_print:
ld     a, (p2points)
call   GetPointSprite
push   hl
; 1er dígito
ld     e, (hl)
inc    hl
ld     d, (hl)
ld     hl, POINTS_P2
call   PrintPoint
pop    hl
; 2° dígito
```

```
inc    hl
inc    hl
ld     e, (hl)
inc    hl
ld     d, (hl)
ld     hl, POINTS_P2
inc    l
jr     PrintPoint
```

Siendo el aspecto final de la rutina el siguiente:

```
; ------------------------------------------------------------------------------
; Repinta el marcador.
; Cada número consta de 1 byte de ancho por 16 de alto.
; Altera el valor de los registros AF, BC, DE y HL.
; ------------------------------------------------------------------------------
ReprintPoints:
ld      hl, (ballPos)               ; Carga la posición de la bola en HL
call    GetPtrY                     ; Obtiene tercio, línea y scanline de esta posición
cp      POINTS_Y_B                  ; Compara con la posición Y donde
                                    ; empieza a borrar el marcador
ret     nc                          ; Si no hay acarreo, paso por debajo y sale
; Si llega aquí la bola podría borrar el marcador, según su posición Y
ld      a, l                        ; Carga línea y columna de la posición
                                    ; de la bola en A
and     $1f                         ; Se queda con la columna
cp      POINTS_X1_L                 ; Compara con la posición donde la bola borrar el
                                    ; marcador del jugador 1 por la izquierda
ret     c                           ; Si hay acarreo pasa por la izquierda y sale
jr      z, reprintPoint_1_print     ; Si coinciden, la bola va a borrar el marcador
                                    ; y repinta
; Sigue con las comprobaciones
cp      POINTS_X2_R                 ; Compara la coordenada X de la bola con la
                                    ; posición donde borra el marcador 2 por la derecha
jr      z, reprintPoint_2_print     ; Si son iguales, repinta el marcador
ret     nc                          ; Si no hay acarreo, pasa por la derecha y sale
; Resto de comprobaciones para averiguar si borra el marcador 1
reprintPoint_1:
cp      POINTS_X1_R                 ; Compara la coordenada X de la bola con la
                                    ; posición donde borra el marcador 1 por la derecha
jr      c, reprintPoint_1_print     ; Si hay acarreo, borra el marcador y repinta
```

```
jr      nz, reprintPoint_2          ; Si no es 0 para por la derecha del marcador 1
                                    ; y salta
; Repinta el marcador del jugador 1
reprintPoint_1_print:
ld      a, (p1points)               ; Carga en A la puntuación del jugador 1
call    GetPointSprite              ; Obtiene la dirección del sprite a pintar
push    hl                          ; Preserva el valor de HL
ld      e, (hl)                     ; Carga en E la parte baja de la dirección
                                    ; del sprite
inc     hl                          ; Apunta HL a la parte alta de la dirección
ld      d, (hl)                     ; La carga en D
ld      hl, POINTS_P1               ; Carga en HL la dirección dónde se pinta el
                                    ; marcador 1
call   PrintPoint                   ; Pinta el primer dígito
pop     hl                          ; Recupera el valor de HL
inc     hl
inc     hl                          ; Apunta HL al sprite del segundo dígito
ld      e, (hl)                     ; Carga la parte baja de la dirección en E
inc     hl                          ; A punta HL a la parte alta de la dirección
ld      d, (hl)                     ; La carga en D
ld      hl, POINTS_P1               ; Carga en HL la dirección dónde se pinta el
                                    ; marcador 1
inc     l                           ; Apunta a la dirección dónde se pinta el segundo
                                    ; dígito
jr      PrintPoint                  ; Pinta el dígito y sale
; Resto de comprobaciones para averiguar si borra el marcador 2
reprintPoint_2:
cp      POINTS_X2_L                 ; Compara la coordenada X de la bola con la
                                    ; posición donde borra el marcador 2 por la
                                    ; izquierda
ret     c                           ; Si hay acarreo, pasa por la izquierda y sale
; Repinta el marcador del jugador 2
reprintPoint_2_print:
ld      a, (p2points)               ; Carga en A la puntuación del jugador 2
call    GetPointSprite              ; Obtiene la dirección del sprite a pintar
push    hl                          ; Preserva el valor de HL
ld      e, (hl)                     ; Carga en E la parte baja de la dirección del
                                    ; sprite
inc     hl                          ; Apunta HL a la parte alta de la dirección
ld      d, (hl)                     ; La carga en D
ld      hl, POINTS_P2               ; Carga en HL la dirección dónde se pinta el
```

```
                                ; marcador 2
call    PrintPoint              ; Pinta el primer dígito
pop     hl                      ; Recupera el valor de HL
inc     hl
inc     hl                      ; Apunta HL al sprite del segundo dígito
ld      e, (hl)                 ; Carga la parte baja de la dirección en E
inc     hl                      ; A punta HL a la parte alta de la dirección
ld      d, (hl)                 ; La carga en D
ld      hl, POINTS_P2           ; Carga en HL la dirección dónde se pinta el
                                ; marcador 2
inc     l                       ; Apunta a la dirección dónde se pinta el segundo
                                ; dígito
jr      PrintPoint              ; Pinta el dígito y sale
```

Compilamos, cargamos en el emulador, y vemos el resultado.

Podemos ver que la bola ahora va más rápida, incluso cuándo tiene que ir lento. También, si nos fijamos cuando es el jugador 2 el que marca el tanto y la bola debe salir por la derecha, parte de la misma se ve durante un corto espacio de tiempo en la izquierda.

Si hacemos memoria, cuando marcamos un punto la pelota sale desde el campo del jugador que ha ganado el punto. Eso nos lleva a la conclusión de que el problema está en la rutina SetBallRight, y más concretamente, en la primera línea:

```
ld    hl, $4d7f
```

Según esta línea, posicionamos la pelota en tercio el 1, scanline 5, línea 3, columna 31.

Además, dos líneas más abajo, cambiamos la rotación de la bola, poniéndola a -1:

```
ld    a, $ff
ld    (ballRotation), a
```

Ahora, si buscamos el sprite correspondiente a esta rotación, vemos que es el siguiente:

```
db    $00, $78    ; +7/$07    00000000 01111000       -1/$ff
```

Por lo que la columna 31 la pintamos en blanco, y en la 32 pintamos $78. Pero es que la columna 32 no existe: las columnas en total son 32, pero van de la 0 a la 31. Al pintar en la 32, estamos pintando en la columna 0.

Una vez visto esto, la solución es sencilla. Editamos la primera línea de la rutina SetBallRight, para posicionar la bola en la columna 30:

```
ld    hl, $4d7e
```

Compilamos, cargamos en el emulador, y vemos que este problema ha quedado resuelto.

Y ahora vamos a cambiar la velocidad de la bola, para que no vaya tan rápida.

La configuración de la bola la tenemos guardada en ballSetting, en el archivo Sprite.asm:

```
; Velocidad y dirección de la bola.
; bits 0 a 3: movimientos de la bola para que cambie la posición Y.
;            Valores f = semiplano, 2 = semi diagonal, 1 = diagonal
; bits 4 y 5: velocidad de la bola: 1 muy rápido, 2 rápido, 3 lento
; bit 6: dirección X: 0 derecha / 1 izquierda
; bit 7: dirección Y: 0 arriba / 1 abajo
ballSetting:
db     $31    ; 0011 0001
```

Según vemos en los comentarios, la velocidad de la bola se configura en los bits 4 y 5. Sería tan sencillo como que la velocidad 2 sea muy rápido, la 3 rápido, y la… En 2 bits solo podemos especificar valores del 0 a 3, y el resto de bits lo tenemos ocupados.

Vamos a "robar" un bit a la inclinación de la bola. Como resultado, podremos reducir la velocidad de la bola, y como contraprestación, cuando la bola vaya plana, va a ir un poco más inclinada:

```
; Velocidad y dirección de la bola.
; bits 0 a 2: Movimientos de la bola para que cambie la posición Y.
;            Valores 7 = semiplano, 2 = semi diagonal, 1 = diagonal
; bits 3 y 5: velocidad de la bola: 2 muy rápido, 3 rápido, 4 lento
; bit 6: dirección X: 0 derecha / 1 izquierda
; bit 7: dirección Y: 0 arriba / 1 abajo
ballSetting:
db     $21    ; 0010 0001
```

Y ahora hay tres rutinas que tenemos que cambiar:

- **CheckCrossY en Game.asm**: en esta rutina asignamos inclinación y velocidad de la bola, dependiendo de en qué parte de la pala golpea.
- **MoveBallY en Game.asm**: en esta rutina evaluamos si los movimientos acumulados de la bola han alcanzado los necesarios para cambiar la coordenada Y de la misma.
- **SetBallLeft y SetBallRight en Game.asm**: en estas rutinas reiniciamos la configuración de la bola.
- **Loop en Main.asm**: al inicio de esta rutina, verificamos si se ha llegado al número de iteraciones del bucle, necesarias para mover la bola.

Empezamos por CheckCrossY en Game.asm. Localizamos la etiqueta checkCrossY_1_5, y después la línea OR $31:

```
or     $31    ; Hacia arriba, velocidad 3 e inclinación diagonal
```

Según la nueva definición, vamos a poner velocidad 4 e inclinación diagonal:

0010 0001

Los bits marcados en rojo especifican la velocidad, y los marcados en verde, la inclinación. La línea OR $31 debe quedar de la siguiente manera:

```
or    $21
```

Localizamos la etiqueta checkCrossY_2_5 y ponemos velocidad 3, inclinación semi diagonal:

0001 1010

Modificamos la línea:

```
or    $22   ; Hacia arriba, velocidad 2 e inclinación semi diagonal
```

Y la dejamos como:

```
or    $1a
```

Localizamos la etiqueta checkCrossY_3_5 y ponemos velocidad 2, inclinación semi plana:

0001 0111

Modificamos la línea:

```
or    $1f   ; Hacia arriba/abajo, velocidad 1 e inclinación semi plana
```

Y la dejamos como:

```
or    $17
```

Localizamos la etiqueta checkCrossY_4_5 y ponemos velocidad 3, inclinación semi diagonal:

1001 1010

Modificamos la línea:

```
or    $a2   ; Hacia abajo, velocidad 2 e inclinación semi diagonal
```

Y la dejamos como:

```
or    $9a
```

Localizamos la etiqueta checkCrossY_5_5 y ponemos velocidad 4, inclinación diagonal:

1010 0001

Modificamos la línea:

```
or    $b1   ; Hacia abajo, velocidad 3 e inclinación diagonal
```

Y la dejamos como:

```
or    $a1
```

Con esto hemos acabado con la parte más laboriosa de la modificación.

Localizamos la etiqueta MoveBallY, y modificamos la segunda línea:

```
and   $0f
```

Y la dejamos como:

```
and   $07
```

Con $0f ahora obtendríamos la inclinación y el primer bit de la velocidad. Con $07 sólo obtenemos la inclinación.

Modificamos el reinicio de la configuración de la bola en las rutinas SetBallLeft y SetBallRight.

En SetBallLeft modificamos la línea:

```
or $31       ; Pone dirección X a derecha, velocidad 3,
             ; inclinación diagonal
```

Y la dejamos como:

```
or    $21
```

En SetBallRight modificamos la línea:

```
or    $71    ; Pone dirección X a izquierda, velocidad 3,

             ; inclinación diagonal
```

Y la dejamos como:

```
or    $61
```

Vamos a terminar modificando el código de la etiqueta Loop de Main.asm.

A partir de la segunda línea, nos encontramos 4 instrucciones RRCA. Quitamos una, para rotar sólo 3 veces y dejar en los bits 0, 1 y 2, la velocidad de la bola.

```
rrca

rrca

rrca

rrca
```

Como ahora tenemos 3 bits para la velocidad, en lugar de dos, modificamos la línea siguiente, que es:

```
and   $03
```

Y la dejamos como:

```
and   $07
```

Compilamos, cargamos en el emulador, y comprobamos que la velocidad de la bola es ahora más llevadera, en detrimento de la inclinación.

Optimización de ScanKeys

Ahora es el momento de optimizar la rutina ScanKeys, tal y como anunciamos en el paso 2.

En la rutina ScanKeys hay cuatro instrucciones BIT, dos `BIT $00, A`, y otras dos `BIT $01, A`. Con las instrucciones BIT comprobamos el estado de un BIT en concreto de un registro, sin alterar el valor de dicho registro; cada instrucción BIT ocupa 2 bytes y tarda 8 ciclos de reloj.

Vamos a sustituir las instrucciones BIT por AND, ahorrándonos un ciclo de reloj en cada una. Sustituimos las instrucciones `BIT $00, A` por `AND $01`, las instrucciones `BIT $01, A` por

`AND $02`. Con esta modificación vamos a ahorrar 4 ciclos de reloj, aunque vamos a alterar el valor del registro A que, en este caso, no importa.

Optimización de Cls

En el paso 3 comentamos que la rutina Cls se podía optimizar ahorrándonos 8 ciclos de reloj y 4 bytes.

Vamos a recordar cómo es la rutina actualmente:

```
; ----------------------------------------------------------------------------
; Limpia la pantalla, tinta 7, fondo 0.
; Altera el valor de los registros AF, BC, DE y HL.
; ----------------------------------------------------------------------------
Cls:
; Limpia los píxeles de la pantalla
ld      hl, $4000     ; Carga en HL el inicio de la VideoRAM
ld      (hl), $00     ; Limpia los píxeles de esa dirección
ld      de, $4001     ; Carga en DE la siguiente posición de la VideoRAM
ld      bc, $17ff     ; 6143 repeticiones
ldir                  ; Limpia todos los píxeles de la VideoRAM

; Pone la tinta en blanco y el fondo en negro
ld      hl, $5800     ; Carga en HL el inicio del área de atributos
ld      (hl), $07     ; Lo pone con la tinta en blanco y el fondo en negro
ld      de, $5801     ; Carga en DE la siguiente posición del área de atributos
ld      bc, $2ff      ; 767 repeticiones
ldir                  ; Asigna el valor a toda el área de atributos

ret
```

La primera parte de la rutina limpia los píxeles, y la segunda asigna los colores a la pantalla. Es en esta segunda parte donde vamos a realizar la optimización.

Una vez ejecutado el primer LDIR, HL vale $57FF y DE vale $5800. Cargar un valor de 16 bits en un registro de 16 bits consume 10 ciclos de reloj y 3 bytes, por lo que haciendo `LD HL, $5800` y `LD DE, $5801`, consumimos 20 ciclos de reloj y 6 bytes.

Como podemos ver, HL y DE valen uno menos de lo que necesitamos para asignar los atributos a la pantalla, por lo que lo único que necesitamos es incrementar su valor en uno, y es ahí donde vamos a conseguir la optimización; vamos a sustituir `LD HL, $5800` y `LD DE, $5801` por `INC HL` e `INC DE`. Incrementar un registro de 16 bits consume 6 ciclos de reloj y ocupa un byte, por lo que el coste total será de 12 ciclos de reloj y 2 bytes, frente a los 20 ciclos de reloj y 6 bytes actuales, logrando un ahorro de 8 ciclos de reloj y 4 bytes.

El aspecto final de la rutina es:

```
; ----------------------------------------------------------------------------
; Limpia la pantalla, tinta 7, fondo 0.
```

```
; Altera el valor de los registros AF, BC, DE y HL.
; -----------------------------------------------------------------------------
Cls:
; Limpia los píxeles de la pantalla
ld      hl, $4000       ; Carga en HL el inicio de la VideoRAM
ld      (hl), $00       ; Limpia los píxeles de esa dirección
ld      de, $4001       ; Carga en DE la siguiente posición de la VideoRAM
ld      bc, $17ff       ; 6143 repeticiones
ldir                    ; Limpia todos los píxeles de la VideoRAM

; Pone la tinta en blanco y el fondo en negro
inc     hl              ; Apunta HL al inicio del área de atributos
ld      (hl), $07       ; Lo pone con la tinta en blanco y el fondo en negro
inc     de              ; Apunta DE a la siguiente posición del área de atributos
ld      bc, $2ff        ; 767 repeticiones
ldir                    ; Asigna el valor a toda el área de atributos

ret
```

Optimización de MoveBall

En el paso 5 comentamos que se podían ahorrar 5 bytes y 2 ciclos de reloj, lo cual vamos a conseguir modificando cinco líneas del conjunto de rutinas MoveBall, que se encuentran en el archivo Game.asm. En concreto vamos a sustituir las cinco líneas `JR moveBall_end` por `RET`; JR ocupa 2 bytes y tarda 12 ciclos de reloj, mientras que RET ocupa 1 byte y tarda 10 ciclos de reloj.

Como podemos observar, en la etiqueta MoveBall_end sólo hay una instrucción, RET, de ahí que podamos sustituir todos los JR moveBall_end por RET.

Hemos dicho que sólo ahorramos 2 ciclos de reloj, lo cual debido a que cada vez que se llama a MoveBall, sólo se ejecuta uno de los JR, por eso solo se ahorran 2 ciclos y no 10, aunque sí se ahorran 5 bytes.

Los JR que vamos a sustituir, los encontramos como última línea de las etiquetas:

- moveBall_right.
- moveBall_rightLast.
- moveBall_rightChg.
- moveBall_left.
- moveBall_leftLast.

La etiqueta movelBall_end se puede eliminar, pero no el RET que la sigue, aunque la etiqueta en sí no ocupa nada.

Optimización de ReprintLine

En el paso 6 comentamos que se podían ahorrar 5 bytes y 22 ciclos de reloj, lo cual vamos a conseguir modificando ocho líneas de la rutina ReprintLine del archivo Video.asm.

Lo primero que vamos a hacer es localizar la etiqueta `reprintLine_loopCont` y la vamos a mover tres líneas más abajo, justo encima de `Call NextScan`.

El siguiente paso es localizar la línea `LD C, LINE` y borrar las tres líneas siguientes:

```
jr     ReprintLine_loopCont
ReprintLine_00:
ld     c, ZERO
```

El siguiente paso es localizar las líneas `JR C, reprintLine_00` y `JR Z, reprintLine_00` y sustituimos `reprintLine_00` por `reprintLine_loopCont`.

El último paso nos lleva al primero. Localizamos la nueva ubicación de la etiqueta `reprintLine_loopCont`, y cuatro líneas más arriba eliminamos `LD C, LINE`. Dos líneas más debajo de la línea eliminada, sustituimos `OR C` por `OR LINE`.

¿Qué hemos hecho?

El objetivo final de la rutina es repintar la parte de la línea central que se ha borrado, sin borrar la parte de la bola que hay donde se tiene que repintar, para lo cual obtenemos los píxeles que hay en pantalla y los mezclamos con la parte de la línea que hay que pintar, y ahí está la cuestión; si lo que hay que repintar de la línea es la parte que va a ZERO (blanco), no es necesario repintarla.

El aspecto final de la rutina es el siguiente:

```
; -------------------------------------------------------------------------------
; Repinta la línea central.
; Altera el valor de los registros AF, B y HL.
; -------------------------------------------------------------------------------
ReprintLine:
ld      hl, (ballPos)               ; Carga en HL la posición de la bola
ld      a, l                        ; Carga la línea y columna en A
and     $e0                         ; Se queda con la línea
or      $10                         ; Pone la columna a 16 ($10)
ld      l, a                        ; Carga el valor en L. HL = Posición inicial

ld      b, $06                      ; Se repintan 6 scanlines
reprintLine_loop:
ld      a, h                        ; Carga tercio y scanline en A
and     $07                         ; Se queda con el scanline
; Si está en los scanlines 0 o 7 pinta ZERO
; Si está en los scanlines 1, 2, 3, 4, 5 o 6 pinta LINE
cp      $01                         ; Comprueba si está en scanline 1 o superior
jr      c, reprintLine_loopCont     ; Si está por debajo, salta
cp      $07                         ; Comprueba si está en scanline 7
jr      z, reprintLine_loopCont     ; Si es así, salta
```

```
ld      a, (hl)                     ; Obtiene los pixeles de la posición actual
or      LINE                        ; Los mezcla con C
ld      (hl), a                     ; Pinta el resultado en la posición actual
reprintLine_loopCont:
call    NextScan                    ; Obtiene el scanline siguiente
djnz    reprintLine_loop            ; Hasta que B = 0

ret
```

Optimización de GetPointSprite

En el paso 8 comentamos que podríamos ahorrar 2 bytes y unos cuantos ciclos de reloj implementando la rutina GetPointSprite de otra manera, lo que vamos a hacer es no usar un bucle.

Actualmente, esta rutina tarda más cuanto mayor sea la puntuación de los jugadores. Mientras el máximo de puntos sea 15 no se aprecia el problema, pero si son 99 o 255, entonces ahí sí que tenemos un problema, tal y como pudimos apreciar cuando hicimos la pruebas y el partido no se paraba al llegar a 15 puntos.

Según la definición de los sprites, cada uno está a 4 bytes del otro, es por eso que lo que hacemos es un bucle partiendo de la dirección de Cero y sumando 4 bytes por cada punto que tiene el jugador del que vamos a pintar el marcador. En realidad, hacer esto sería lo mismo que multiplicar los puntos del jugador por 4, y sumarle el resulto a la dirección del sprite Cero. De esta manera siempre va a tardar los mismo, sean 0 o 99 puntos, nos ahorramos 2 bytes y unos cuantos ciclos de reloj.

Recordemos que en GetPointSprite, recibimos en A la puntuación, y devolvemos en HL la dirección del sprite a pintar.

¿Cómo multiplicamos por 4 si el Z80 no tiene una instrucción para multiplicar?

Multiplicar no es más que sumar un número tantas veces como dice el multiplicador, o lo que es lo mismo, multiplicar un número por cuatro, sería igual a:

2*4 = 2+2+2+2 = 8

Esto lo podríamos hacer con un bucle, pero lo vamos a simplificar aún más, ya que para multiplicar un número por 4, solo nos hace falta hacer dos sumas:

3*4 = 3+3 = 6   6+6 = 12

Es decir, sumamos el número a sí mismo, y el resultado lo sumamos a si mismo, y ya tenemos hecha la multiplicación por 4. Si ese resultado lo sumamos a si mismo, ya tendríamos la multiplicación por 8, y si seguimos así por 16, 32, 64... o lo que es lo mismo $n*2^n$.

Tenemos dos maneras de implementar GetPointSprite sin necesidad de modificar nada más: con un marcador de hasta 61 puntos o un marcador de hasta 99 puntos.

Vamos con la primera implementación, con un marcador de hasta 61 puntos (61 * 4 = 244 = 1 byte)

```
; ------------------------------------------------------------------------------
; Obtiene el sprite correspondiente a pintar en el marcador.
; Entrada:     A -> puntuación.
; Salida:      HL -> Dirección del sprite a pintar.
; Altera el valor de los registros AF, BC y HL.
; ------------------------------------------------------------------------------
GetPointSprite:
; HASTA 61 PUNTOS
ld             hl, Cero                     ; Carga en HL la dirección del sprite del 0
; Cada sprite está del anterior a 4 bytes
add            a, a                         ; Multiplica A * 2
add            a, a                         ; Multiplica A * 2 = A original por 4
ld             b, ZERO
ld             c, a                         ; Carga el valor de A en BC
add            hl, bc                       ; Se lo suma a HL
ret
```

En este caso, la puntuación máxima sería 61 que al multiplicarlo por 4 da 244, resultado que nos cabe en un byte y por tanto podemos usar el registro A para realizar la multiplicación por 4. Esta rutina ocupa 10 bytes y tarda 50 ciclos de reloj.

Si una partida de Pong a 61 puntos se nos hace corta la podemos hacer a 99, la rutina ocuparía lo mismo que la anterior, pero tardaría 64 ciclos de reloj (en este caso las sumas hay que hacerlas con un registro de 16bits ya que 99*4 = 396 = 2 bytes).

```
; ------------------------------------------------------------------------------
; Obtiene el sprite correspondiente a pintar en el marcador.
; Entrada:     A -> puntuación.
; Salida:      HL -> Dirección del sprite a pintar.
; Altera el valor de los registros AF, BC y HL.
; ------------------------------------------------------------------------------
GetPointSprite:
; HASTA 255 PUNTOS, 99 SI NO SE CAMBIA LA RUTINA DE IMPRESIÓN DEL MARCADOR
ld             h, ZERO
ld             l, a                 ; Carga en HL los puntos
; Cada sprite está del anterior a 4 bytes
add            hl, hl               ; Multiplica HL * 2
add            hl, hl               ; Multiplica HL * 2 = HL original por 4
ld             bc, Cero             ; Carga en BC la dirección del sprite del 0
add            hl, bc               ; Lo suma a HL para calcular donde está el sprite
                                    ; que corresponde a la puntuación
ret
```

Si queremos una puntuación mayor de 99 hay que modificar la rutina de impresión de los marcadores pues ahora solo imprime dos dígitos, y tener en cuenta que estas implementaciones de GetPointSprite tampoco serían válidas (posiblemente habría que repensar todo, empezando por la forma de declarar los sprites).

Optimización PrintPoints y ReprintPoints

¡Pero oye! Si ReprintPoints la acabamos de implementar de nuevo, al inicio de este capítulo.

Bueno, en realidad hemos añadido una parte para que repinte el marcador solo cuando sea necesario, pero hemos heredado alguna cosilla de la implementación original.

En el paso 8 comentamos que podríamos ahorrar 2 bytes y 12 ciclos de reloj haciendo una pequeña modificación en la rutina PrintPoints. Pues bien, estamos de enhorabuena ya que en realidad nos vamos a ahorrar 33 bytes y 178 ciclos de reloj; los cambios que vamos realizar en PrintPoints, los vamos a realizar también en ReprintPoints.

En la tercera línea de PrintPoints encontramos `PUSH HL`, y esta es la primera línea que vamos a cambiar de lugar, ya que preservamos el valor del registro HL antes de tiempo. Cortamos esta línea y la pegamos tres líneas más abajo, justo antes de cargar la dirección de memoria donde se pintan los puntos del jugador 1 en HL, `LD HL, POINTS_P1`. El motivo de preservar el valor del registro HL es justamente esta instrucción.

Una vez que llamamos a pintar el punto, recuperamos el valor de HL, POP HL, e incrementamos HL dos veces para apuntarlo a la parte baja de la dirección donde está el segundo dígito. Pues bien, como hemos preservado HL después de posicionarnos en la parte alta de la dirección del primer dígito, ahora vamos a quitar uno de estos dos `INC HL`: nos acabamos de ahorrar 1 byte y 6 ciclos de reloj.

Esta misma modificación tenemos que hacerla al pintar el marcador del jugador 2 y en la rutina ReprintPoints. En total ahorramos 4 bytes y 24 ciclos de reloj.

Spirax comentó otra optimización que podríamos hacer, con la cual podremos quitar cuatro instrucciones `INC L`, ahorrando otros 4 bytes y 16 ciclos de reloj.

Tanto en PrintPoints como en ReprintPoints, al dibujar el segundo dígito de los marcadores hacemos los siguiente.

```
ld hl, POINTS_P1
inc l


ld hl, POINTS_P2
inc l
```

Como esto lo hacemos tanto en PrintPoints como en ReprintPoints, en realidad hacemos cuatro veces INC L, y lo podemos evitar de la siguiente manera:

```
ld hl, POINTS_P1 + 1


ld hl, POINTS_P2 + 1
```

De esta forma apuntamos directamente HL a la posición donde se dibuja el segundo dígito, y nos ahorramos los `INC L`.

Y ahora nos vamos a ahorrar 25 bytes y 138 ciclos de reloj más gracias otra vez a Spirax.

En la parte final de la rutina ReprintPoints encontramos la etiqueta `reprintPoint2_print`, y justo por encima de esta etiqueta la instrucción `RET C`. Bien, vamos a borrar la etiqueta `reprintPoint2_print` y todo lo que tiene por debajo hasta el final de la rutina. Después de `RET C` vamos a incluir `JR printPoint2_print`.

En una implementación anterior, `PrintPoints` y `ReprintPoints` pintaban de distinta manera, pues `ReprintPoints` hacía `OR` con los píxeles de la pantalla, pero este ya no es el caso, por lo que vamos a utilizar el código que pinta el marcador del jugador 2 para repintarlo, y nos vamos a ahorrar 25 bytes y 138 ciclos de reloj.

La etiqueta `printPoint2_print` no existe, por lo que vamos a incluirla. Buscamos la etiqueta `PrintPoints`, como vemos primero pinta el marcador del jugador 1, y una vez que ha finaliza pinta el marcador del jugador 2, que empieza justo debajo de la segunda llamada a `PrintPoint`. Pues es ahí, justo debajo del segundo `CALL PrintPoint`, donde vamos a añadir la etiqueta `printPoint_2_print`.

¡Muchas gracias Spirax!

El aspecto final de las rutinas es el siguiente:

```
; -----------------------------------------------------------------------------
; Pinta el marcador.
; Cada número consta de 1 byte de ancho por 16 de alto.
; Altera el valor de los registros AF, BC, DE y HL.
; -----------------------------------------------------------------------------
PrintPoints:
ld      a, (p1points)          ; Carga en A los puntos del jugador 1
call    GetPointSprite         ; Obtiene el sprite a pintar en el marcador
; 1er dígito del jugador 1
ld      e, (hl)                ; Carga en E la parte baja de la dirección
                               ; donde está el primer dígito
inc     hl                     ; Apunta HL a la parte alta de la dirección
                               ; donde está el primer dígito
ld      d, (hl)                ; y la carga en D
push    hl                     ; Preserva el valor de HL
ld      hl, POINTS_P1          ; Carga en HL la dirección de memoria donde se pintan
                               ; los puntos del jugador 1
call    PrintPoint             ; Pinta el primer dígito del marcador del jugador 1
pop     hl                     ; Recupera el valor de HL
; 2º dígito del jugador 1
inc     hl                     ; Apunta HL a la parte baja de la dirección
                               ; donde está el segundo dígito
ld      e, (hl)                ; y la carga en E
inc     hl                     ; Apunta HL a la parte alta de la dirección
```

```
                              ; donde está el segundo dígito
ld      d, (hl)               ; y la carga en D
; Spirax
ld      hl, POINTS_P1 + 1     ; Carga en HL la dirección de memoria donde se pinta
                              ; el segundo dígito de los puntos del jugador 1
call    PrintPoint            ; Pinta el segundo dígito del marcador del jugador 1

printPoint_2_print:
; 1er dígito del jugador 2
ld      a, (p2points)         ; Carga en A los puntos del jugador 2
call    GetPointSprite        ; Obtiene el sprite a pintar en el marcador
ld      e, (hl)               ; Carga en E la parte baja de la dirección
                              ; donde está el primer dígito
inc     hl                    ; Apunta HL a la parte alta de la dirección
                              ; donde está el primer dígito
ld      d, (hl)               ; y la carga en D
push    hl                    ; Preserva el valor de HL
ld      hl, POINTS_P2         ; Carga en HL la dirección de memoria donde se pintan
                              ; los puntos del jugador 2
call    PrintPoint            ; Pinta el primer dígito del marcador del jugador 2
pop     hl                    ; Recupera el valor de HL
; 2º dígito del jugador 2
inc     hl                    ; Apunta HL a la parte baja de la dirección
                              ; donde está el segundo dígito
ld      e, (hl)               ; y la carga en E
inc     hl                    ; Apunta HL a la parte alta de la dirección
                              ; donde está el segundo dígito
ld      d, (hl)               ; y la carga en D
; Spirax
ld      hl, POINTS_P2 + 1     ; Carga en HL la dirección de memoria donde se pinta
                              ; el segundo dígito de los puntos del jugador 2
; Pinta el segundo dígito del marcador del jugador 2
PrintPoint:
ld      b, $10                ; Cada dígito son 1 byte por 16 (scanlines)
printPoint_printLoop:
ld      a, (de)               ; Carga en A el byte a pintar
ld      (hl), a               ; Pinta el byte
inc     de                    ; Apunta DE al siguiente byte
call    NextScan              ; Apunta HL al siguiente scanline
djnz    printPoint_printLoop  ; Hasta que B = 0
```

```
ret

; ----------------------------------------------------------------------------
; Repinta el marcador.
; Cada número consta de 1 byte de ancho por 16 de alto.
; Altera el valor de los registros AF, BC, DE y HL.
; ----------------------------------------------------------------------------
ReprintPoints:
ld      hl, (ballPos)         ; Carga en HL la posición de la bola
call    GetPtrY        ; Obtiene tercio, línea y scanline de la posición de la bola
cp      POINTS_Y_B            ; Lo compara con el límite inferior del marcador
ret     nc                    ; Si no hay acarreo, pasa por debajo y sale
ld      a, l                  ; Carga en A la línea y columna de la posición de la bola
and     $1f                   ; Se queda con la columna
cp      POINTS_X1_L           ; Lo compara con el límite izquierdo del marcador 1
ret     c                     ; Si hay acarreo, pasa por la izquierda y sale
jr      z, reprintPoint_1_print      ; Si es 0, está justo en el margen izquierdo
                                     ; y salta para pintar

cp      POINTS_X2_R           ; Lo compara con el límite derecho de marcador 2
jr      z, printPoint_2_print ; Si es 0, está justo en el margen derecho
                              ; y salta para pintar
ret     nc                    ; Si no hay acarreo, pasa por la derecha y sale

reprintPoint_1:
cp      POINTS_X1_R           ; Lo compara con el límite derecho de marcador 1
jr      c, reprintPoint_1_print      ; Si hay acarreo, pasa por el marcador 1
                                     ; y salta para pintar
jr      nz, reprintPoint_2    ; Si no es cero, pasa por la derecha
                              ; y salta para comprobar paso por marcador 2
reprintPoint_1_print:
ld      a, (p1points)         ; Carga en A los puntos del jugador 1
call    GetPointSprite        ; Obtiene el sprite a pintar en el marcador
; 1er dígito
ld      e, (hl)               ; Carga en E la parte baja de la dirección donde
                              ; está el primer dígito
inc     hl                    ; Apunta HL a la parte alta de la dirección donde
                              ; está el primer dígito
ld      d, (hl)               ; y la carga en D
push    hl                    ; Preserva el valor de HL
ld      hl, POINTS_P1         ; Carga en HL la dirección de memoria donde se pintan
```

```
                              ; los puntos del jugador 1
call    PrintPoint            ; Pinta el primer dígito del marcador del jugador 1
pop     hl                    ; Recupera el valor de HL
; 2° dígito
inc     hl                    ; Apunta HL a la parte baja de la dirección donde
                              ; está el segundo dígito
ld      e, (hl)               ; y la carga en E
inc     hl                    ; Apunta HL a la parte alta de la dirección donde
                              ; está el segundo dígito
ld      d, (hl)               ; y la carga en D
ld      hl, POINTS_P1 + 1     ; Carga en HL la dirección de memoria donde se pinta
                              ; el segundo dígito de los puntos del jugador 1
jr      PrintPoint            ; Pinta el segundo dígito del marcado del jugador 2
reprintPoint_2:
cp      POINTS_X2_L           ; Lo compara con el límite derecho de marcador 2
ret     c                     ; Si hay acarreo, pasa por la izquierda y sale
; Spirax
jr      printPoint_2_print    ; Pinta el marcador del jugador
```

Compilamos, cargamos en el emulador y comprobamos que todo sigue funcionando.

Bug del golpeo de la bola en la parte baja de la pala

Es el momento de arreglar un bug que arrastramos desde que implementamos el cambio de velocidad e inclinación de la bola en base a en que parte de la pala golpea. Cuando la bola golpea en el último scanline de la pala, no cambia inclinación, ni velocidad, ni dirección vertical. ¿A qué se debe?

El motivo está en la forma en la que implementamos la detección de colisiones. Antes de evaluar en que parte de la pala golpea, evaluamos si golpea en la pala, y aquí está el error; cuando golpea en el último scanline de la pala sale de la rutina, con el flag Z activado indicando que hay colisión, pero sin evaluar en que parte de la pala golpea.

Abrimos el archivo Video.asm y localizamos la etiqueta CheckCrossY. Quince líneas más abajo nos encontramos con esto.

```
ret     nc      ; Si no hay acarreo la bola pasa por debajo
                ; de la pala o colisiona en el último scanline.
                ; En este último caso ya se ha activado el flag Z
```

Si leemos atentamente los comentarios, salimos de la rutina si no hay acarreo (flag Z desactivado = no hay colisión). El problema es qué si no hay acarreo, el resultado puede ser mayor o igual a 0. Es decir, si el resultado es 0, salimos de la rutina con el flag Z activado (hay colisión) sin evaluar en que parte de la pala ha golpeado.

Para solucionar este aspecto vamos a hacer una doble comprobación y añadir una nueva etiqueta a la que saltar.

El código actual de la parte que vamos a tocar es el siguiente.

```
ret     nc      ; Si no hay acarreo la bola pasa por debajo
                ; de la pala o colisiona en el último scanline.
                ; En este último caso ya se ha activado el flag Z

; Dependiendo de donde sea la colisión, se asigna grado de inclinación
; y velocidad a la bola
ld      a, c    ; Carga la posición del penúltimo scanline de la pala en A
```

Vamos a añadir una línea antes de `RET NC` y una etiqueta antes de `LD A, C`, dejando el código de la siguiente manera.

```
jr      z, checkCrossY_eval   ; Si es cero, choca en el último scanline
ret     nc                    ; Si no hay acarreo la bola pasa por debajo y sale.

; Dependiendo de donde sea la colisión, se asigna grado de inclinación
; y velocidad a la bola
checkCrossY_eval:
ld      a, c                  ; Carga la posición del penúltimo scanline de la pala en A
```

Incluso ese `JR Z, checkCrossY_eval` lo podríamos cambiar por `JR Z, checkCrossY_5_5`, pues sabemos que ha golpeado en la parte inferior de la pala (probad de las dos maneras).

Compilamos, cargamos en el emulador y comprobamos que hemos arreglado el bug.

Sonido

Y abordamos el penúltimo paso; vamos a implementar efectos de sonido cuando la bola golpea con los laterales, las palas, o cuando se marque algún punto.

Añadimos el archivo Sound.asm, y añadimos las contantes y rutinas necesarias para nuestros efectos de sonido, que van a ser los sonidos que se van a reproducir cuando la bola rebota contra los distintos elementos.

Vamos a definir tres sonidos distintos:

- Cuando se marca un punto.
- Cuando la bola choca con una pala.
- Cuando la bola choca con el borde.

Para cada sonido tenemos que definir la nota y la frecuencia. La frecuencia es el tiempo que va a durar la nota, y la vamos a identificar con el sufijo FQ.

```
; Punto
C_3:        EQU $0D07
C_3_FQ:     EQU $0082 / $10

; Pala
C_4:        EQU $066E
```

```
C_4_FQ:      EQU $0105 / $10


; Rebote
C_5:         EQU $0326
C_5_FQ:      EQU $020B / $10
```

Todos los sonidos que vamos a usar son DO, aunque en distintas escalas; a mayor escala, el sonido es más agudo.

Las frecuencias especificadas son las que hacen que la nota dure un segundo, es por eso que las dividimos por 16. Si las multiplicáramos por 2, la nota duraría 2 segundos.

A cada nota, en cada escala, le corresponde una frecuencia propia. En el apéndice 1 se muestran sendas tablas con frecuencias y notas, en decimal, hexadecimal, y código ensamblador.

La siguiente constante que vamos a ver, es la dirección de memoria donde está alojada la rutina BEEPER de la ROM:

```
BEEPER:      EQU $03B5
```

Esta rutina recibe en HL la nota y en DE la duración, y altera el valor de los registros AF, BC, DE, HL e IX, además de otro aspecto que veremos más adelante.

Debido a que la rutina BEEPER de la ROM altera tantos registros, es recomendable no llamarla directamente; vamos a implementar una rutina que lo haga.

La rutina que vamos a implementar, recibe en A el tipo de sonido a emitir, 1 = punto, 2 = pala, 3 = borde, y no altera el valor de ningún registro:

```
PlaySound:
push  de
push  hl
```

Preservamos el valor de los registros DE, `PUSH DE`, y HL, `PUSH HL`.

```
cp    $01
jr    z, playSound_point
```

Comprobamos si el sonido a reproducir es de tipo 1 (punto), `CP $01`, y de ser así saltamos, `JR Z, playSound_point`.

```
cp    $02
jr    z, playSound_paddle
```

Si el sonido no es de tipo 1, comprobamos si es de tipo 2 (pala), `CP $02`, y de ser así saltamos, `JR Z, playSound_paddle`.

Si el sonido no es de tipo 1, ni de tipo 2, es de tipo 3 (borde):

```
ld    hl, C_5
ld    de, C_5_FQ
```

```
jr    beep
```

Cargamos en HL la nota, `LD HL, C_5`, cargamos en DE la frecuencia (duración), `LD DE, C_5_FQ`, y saltamos a reproducir el sonido, `JR beep`.

Si el sonido es de tipo 1 o 2, hacemos lo mismo, pero con los valores de cada sonido:

```
playSound_point:
ld    hl, C_3
ld    de, C_3_FQ
jr    beep

playSound_paddle:
ld    hl, C_4
ld    de, C_4_FQ
```

Nos ahorramos el último JR, ya que justo después viene la rutina que reproduce el sonido:

```
beep:
push  af
push  bc
push  ix
call  BEEPER
pop   ix
pop   bc
pop   af

pop   hl
pop   de

ret
```

Preservamos los valores de AF, `PUSH AF`, de BC, `PUSH BC`, y de IX, `PUSH IX`. Llamamos a la rutina de la ROM, `CALL BEEPER`, y recuperamos los valores de IX, `POP IX`, de BC, `POP BC`, de AF, `POP AF`, de HL, `POP HL`, y de DE, `POP DE`. Los valores de HL y DE los preservamos al principio de la rutina PlaySound. Por último, salimos, `RET`.

El aspecto final del archivo Sound.asm, es el siguiente:

```
; ------------------------------------------------------------------------------
; Sound
; Fichero con los sonidos
; ------------------------------------------------------------------------------
; Punto
```

```
C_3:   EQU $0D07
C_3_FQ: EQU $0082 / $10

; Pala
C_4:   EQU $066E
C_4_FQ: EQU $0105 / $10

; Rebote
C_5:   EQU $0326
C_5_FQ: EQU $020B / $10

; -----------------------------------------------------------------------------
; Rutina beeper de la ROM.
;
; Entrada:     HL      ->      Nota.
;              DE      ->      Duración.
;
; Altera el valor de los registros AF, BC, DE, HL e IX.
; -----------------------------------------------------------------------------
BEEPER:        EQU $03B5

; -----------------------------------------------------------------------------
; Reproduce el sonido de los rebotes.
; Entrada:      A       ->      Tipo de rebote.     1. Punto
;                                                   2. Pala
;                                                   3. Borde
; -----------------------------------------------------------------------------
PlaySound:
; Preserva el valor de los registros
push    de
push    hl

cp      $01                         ; Evalúa si se emite el sonido de Punto
jr      z, playSound_point          ; Si es así salta

cp      $02                         ; Evalúa si se emite el sonido de Pala
jr      z, playSound_paddle         ; Si es así salta

; Se emite el sonido de Borde
ld      hl, C_5                     ; Carga en HL la nota
ld      de, C_5_FQ                  ; Carga en DE la duración (frecuencia)
```

```
jr      beep                        ; Salta a emitir el sonido

; Se emite el sonido de Punto
playSound_point:
ld      hl, C_3                     ; Carga en HL la nota
ld      de, C_3_FQ                  ; Carga en DE la duración (frecuencia)
jr      beep                        ; Salta a emitir el sonido

; Se emite el sonido de Pala
playSound_paddle:
ld      hl, C_4                     ; Carga en HL la nota
ld      de, C_4_FQ                  ; Carga en DE la duración (frecuencia)

; Hace sonar la nota
beep:
; Preserva el valor de los registros ya que la rutina BEEPER de la ROM los altera
push    af
push    bc
push    ix

call    BEEPER                      ; Llama a la rutina BEEPER de la ROM

; Recupera el valor de los registros
pop     ix
pop     bc
pop     af

pop     hl
pop     de

ret
```

Para acabar, tenemos que llamar a nuestra nueva rutina para emitir los sonidos de los rebotes de la bola.

Abrimos el archivo Game.asm y localizamos la etiqueta checkBallCross_right. Vamos a añadir dos líneas entre la línea `RET NZ`, y la línea `LD A, (ballSetting)`:

```
ld    a, $02
call  PlaySound
```

Carga el tipo de sonido en A, `LD A, $02`, y emite el sonido, `CALL PlaySound`.

Localizamos la etiqueta checkBallCross_left. Vamos a añadir las mismas dos líneas de antes entre la línea `RET NZ`, y la línea `LD A, (ballSetting)`:

```
ld    a, $02
call  PlaySound
```

Localizamos la etiqueta moveBall_upChg. Justo debajo de la misma, añadimos dos líneas casi iguales a las anteriores:

```
ld    a, $03
call  PlaySound
```

Localizamos la etiqueta moveBall_downChg. Justo debajo de la misma, añadimos las dos líneas anteriores:

```
ld    a, $03
call  PlaySound
```

Localizamos la etiqueta moveBall_rightChg, y justo debajo añadimos:

```
ld    a, $01
call  PlaySound
```

Cinco líneas más abajo localizamos CALL SetBallLeft. Debajo añadimos:

```
ld    a, $03
call  PlaySound
```

Localizamos la etiqueta moveBall_leftChg, y justo debajo añadimos:

```
ld    a, $01
call  PlaySound
```

Cinco líneas más abajo localizamos CALL SetBallRight. Debajo añadimos:

```
ld    a, $03
call  PlaySound
```

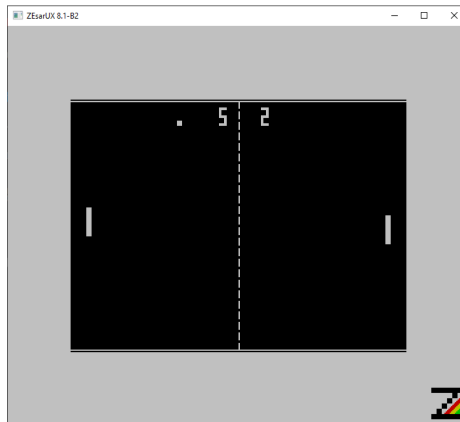
Por último, abrimos el archivo Main.asm, localizamos la rutina Loop y justo encima añadimos las siguientes líneas:

```
ld    a, $03
call  PlaySound
```

Nos vamos al final del fichero, y en la parte de los “includes”, incluimos el archivo Sound.asm:

```
include     "Sound.asm"
```

Si todo ha ido bien, hemos llegado al final. Compilamos, cargamos en el emulador y…



¿Qué le pasa al borde? ¿Por qué es blanco? Bueno, ya advertimos que la rutina BEEPER de la ROM altera muchas cosas, y una de ellas es el color del borde, aunque tiene fácil solución.

Por suerte, tenemos una variable de sistema donde podemos guardar el color del borde. En esta variable se guardan también los atributos de la pantalla inferior. El fondo de dicha pantalla es el color del borde.

Abrimos el archivo Video.asm y al inicio del mismo declaramos una constante con la dirección de memoria de dicha variable del sistema:

```
BORDCR:      EQU $5c48
```

Localizamos la rutina Cls, y antes de la línea `LD HL, $5800`, añadimos:

```
ld     a, $07       ; Fondo negro, tinta blanca
```

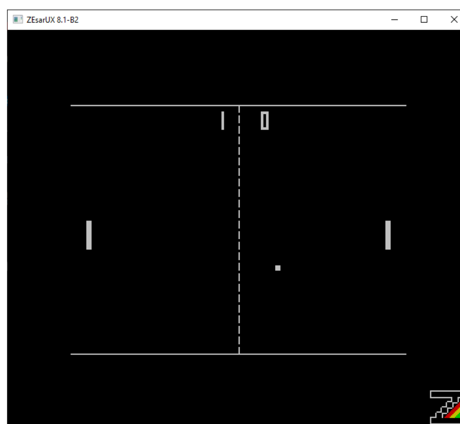
Modificamos la línea LD (HL), $07 dejándola así:

```
ld     (hl), a
```

Por último, antes de RET, añadimos:

```
ld     (BORDCR), a
```

Compilamos, cargamos en el emulador, y ahora sí. ¿Hemos terminado nuestro PorompomPong?



Todavía nos falta una última cosa por hacer. ¿Es compatible nuestro programa con el modelo de 16 Kb? Pues todavía no, pero como no trabajamos con interrupciones, es muy sencillo hacerlo compatible.

Vamos a abrir el archivo Main.asm, vamos a localizar las directivas ORG y END, y vamos a sustituir $8000 por $5dad en el caso de ORG. En el caso de END, vamos a sustituir $8000 por Main, que es la etiqueta de entrada al programa.

Si ahora compilamos y cargamos en el emulador con el modelo 16K, nuestro programa es compatible.

Si nos fijamos bien, podemos observar que se ha perdido algo de velocidad. Esta pérdida es debida a que los segundos 16 KiB del ZX Spectrum, que es donde ahora cargamos el programa, es lo que se llama memoria contenida, y está compartida con la ULA; uando la ULA trabaja, todo se para.

Vamos a volver a cambiar la velocidad a la que va la bola.

Abrimos el archivo Sprite.asm, localizamos la etiqueta ballSetting, comentamos la línea db $21 y escribimos justo debajo:

```
; or  $21
or    $19
```

Ahora la bola se inicia a velocidad 3, que va a ser la más lenta.

Abrimos el archivo Game.asm, localizamos la etiqueta SetBallLeft, comentamos la línea 7, y escribimos justo debajo:

```
; or  $21
or    $19
```

Ahora, cuando reiniciamos la bola para que salga por la izquierda de la pantalla, se inicia a velocidad 3.

Localizamos la etiqueta SetBallRight, comentamos la línea 7, y escribimos justo debajo:

```
; db  $61
db    $59
```

Ahora, cuando reiniciamos la bola para que salga por la derecha de la pantalla, se inicia a velocidad 3.

Localizamos la etiqueta checkCrossY_1_5, comentamos la línea 7, y escribimos justo debajo:

```
; or  $21
or    $19
```

Ahora la velocidad de la bola es 3 en lugar de 4.

Localizamos la etiqueta checkCrossY_2_5, comentamos la línea 7, y escribimos justo debajo:

```
; or  $1a
or    $12
```

Ahora la velocidad de la bola es 2 en lugar de 3.

Localizamos la etiqueta checkCrossY_3_5, comentamos la línea 7, y escribimos justo debajo:

```
; or  $17
or    $0f
```

Ahora la velocidad de la bola es 1 en lugar de 2.

Localizamos la etiqueta checkCrossY_4_5, comentamos la línea 7, y escribimos justo debajo:

```
; or  $9a
or    $92
```

Ahora la velocidad de la bola es 2 en lugar de 3.

Localizamos la etiqueta checkCrossY_5_5, comentamos la línea 3, y escribimos justo debajo:

```
; or  $a1
or    $99
```

Ahora la velocidad de la bola es 3 en lugar de 4.

Compilamos, probamos en el emulador y hemos terminado.

Paso 11: optimización parte 2

Como hemos comentado anteriormente, Spirax señaló varias optimizaciones. Hemos dejado para el final una que mostró para la rutina de sonido y otra que se ha implementado siguiendo la que planteo para la rutina ReprintPoints.

Como de costumbre, creamos una carpeta llamada Paso11 y copiamos en ella todos los archivos .asm de la carpeta Paso10.

Optimización de PlaySound

La primera optimización la vamos a realizar en la rutina de sonido, en concreto en la manera en la que evaluamos el sonido que tenemos que emitir.

Abrimos el archivo Sound.asm y localizamos la etiqueta PlaySound, cuyas primeras líneas son:

```
PlaySound:
; Preserva el valor de los registros
push    de
push    hl
cp      $01                   ; Evalúa si se emite el sonido de Punto

jr      z, playSound_point    ; Si el resultado es 0, el valor de A era 1 y emite el
                              ; sonido del punto
cp      $02                   ; Evalúa si se emite el sonido de Pala

jr      z, playSound_paddle   ; Si el resultado es 0, el valor de A era 2 y emite el
                              ; sonido de choque con la pala
```

En esta rutina utilizamos `CP $01` y `CP $02` para comprobar que sonido hay que emitir. Cada instrucción CP ocupa 2 bytes y tarda 7 ciclos de reloj; vamos a sustituir estas instrucciones por `DEC A`, que ocupa 1 byte y tarda 4 ciclos de reloj, por lo que nos vamos a ahorrar 2 bytes y 6 ciclos de reloj. Al contrario de `CP`, `DEC` si altera el valor del registro A, pero dado que lo que tenemos en A es el tipo de sonido a emitir, no nos afecta.

Veamos como queda el inicio de la rutina.

```
PlaySound:
; Preserva el valor de los registros
push    de
push    hl


; Spirax
dec     a                     ; Evalúa si se emite el sonido de Punto
jr      z, playSound_point    ; Si el resultado es 0, el valor de A era 1 y emite el
                              ; sonido del punto
; Spirax
dec     a                     ; Evalúa si se emite el sonido de Pala
jr      z, playSound_paddle   ; Si el resultado es 0, el valor de A era 2 y emite el
```

```
; sonido de choque con la pala
```

Primero preserva el valor de DE, `PUSH DE`, luego el de HL, `PUSH HL`, y a continuación decrementa A, `DEC A`. Si A era 1, el resultado de la operación es 0 y salta a emitir el sonido, `JR Z, PlaySound_point`.

Si A no era uno, seguimos con las comprobaciones; decrementa A, `DEC A`, y si el resultado de la operación es 0 salta a emitir el sonido, `JR Z, PlaySound_paddle`. Si salta a reproducir el sonido es porque inicialmente A valía 2, con el primer decremento vale 1 y con este segundo decremento vale 0.

Si no ha saltado, la rutina sigue tal y como estaba y emite el sonido del punto.

Este es el momento de compilar, cargar en el emulador y comprobar que todo sigue funcionando.

Optimización de ReprintPoints

Con esta optimización vamos a ahorrar 20 bytes y 107 ciclos de reloj. Para lograr este ahorro vamos a aplicar el mismo método que aplicamos en la entrega anterior, siguiendo los comentarios de Spirax, vamos a seguir por el camino que nos marcó.

Como recordaréis del capítulo anterior, pusimos una etiqueta para que el pintado del marcador del jugador 2 se pudiera llamar de manera independiente; vamos a hacer los mismo con el marcador del jugador 1. Con esta modificación, los marcadores van a tardar algo más en pintarse (solo se pintan al inicio de la partida y al marcar un punto), pero vamos a simplificar la rutina ReprintPoints, ahorrando bytes y ciclos de reloj, y eliminando código redundante.

Vamos a empezar modificando la rutina PrintPoints para que se pueda llamar de manera independiente al pintado de los marcadores de ambos jugadores.

Abrimos el archivo Video.asm y localizamos la etiqueta PrintPoints; justo debajo de ella agregamos otra etiqueta; es la que vamos a llamar para pintar el marcador del jugador 2:

```
printPoint_1_print:
```

Entre las etiquetas PrintPoints y printPoint_1_print vamos a añadir las llamadas a pintar el marcador de cada jugador:

```
call    printPoint_1_print    ; Pinta el marcador del jugador 1

jr      printPoint_2_print    ; Pinta el marcador del jugador 2
```

Lo primero que hacemos es llamar a pintar el marcador del jugador 1, `CALL printPoint_1_print`, y luego saltar a pintar el marcador del jugador 2, `JR printPoint_1_print`.

Ya solo queda un cambio en PrintPoints, hay que añadir `RET` justo antes de la etiqueta printPoint_2_print, para que `CALL printPoint_1_print` salga correctamente; recordad que el resto de saltos salen por el `RET` de PrintPoint.

Añadimos `RET` antes de printPoint_2_print:

```
ret

printPoint_2_print:
```

Con esto hemos acabado con las modificaciones necesarias en PrintPoints, y como vemos no hemos ahorrado nada, al contrario, hemos añadido código, añadiendo bytes y ciclos de reloj.

Vamos ahora con el ahorro, para ello localizamos la etiqueta reprintPoint_1_print y la borramos. También borramos las líneas que la siguen hasta llegar a la etiqueta reprintPoint_2, está ultima etiqueta no la borramos.

Localizamos la etiqueta ReprintPoints y nueve líneas más abajo encontramos la instrucción JR Z, reprintPoint_1_print. Dado que esta etiqueta ya no existe, hay que cambiar esta línea y dejarla como sigue:

```
jr      z, printPoint_1_print
```

Ahora Localizamos la etiqueta reprintPoint_1 y vamos a terminar con las modificaciones.

El código actual de esta etiqueta, una vez borrada toda la parte de reprintPoint_1_print es el siguiente:

```
reprintPoint_1:
cp      POINTS_X1_R           ; Lo compara con el límite derecho de marcador 1
jr      c, printPoint_1_print ; Si hay acarreo, pasa por el marcador 1
                              ; y salta para pintar
jr      nz, reprintPoint_2    ; Si no es cero, pasa por la derecha
                              ; y salta para comprobar paso por marcador 2
```

Tenemos que cambiar la doble comprobación; dado que la etiqueta reprintPoint_2 esta ahora justo debajo de la línea `JR NZ, reprintPoint_2`, ese salto ya no es necesario, pero si que es necesario comprobar si es cero, en cuyo caso hay que pintar el marcador del jugador 1, `JR Z, printPoint_1_print`, y cambiar el salto de `JR C, reprintPoint_1_print` por `JR C, printPoint_1_print,` por lo tanto, el código quedaría así:

```
reprintPoint_1:
cp      POINTS_X1_R           ; Lo compara con el límite derecho de marcador 1
jr      z, printPoint_1_print
jr      c, printPoint_1_print ; Si es 0 o hay acarreo, pasa por el marcador 1
                              ; y salta para pintar
```

El aspecto final de las rutinas PrintPoints y ReprintPoints es el siguiente:

```
; -----------------------------------------------------------------------------
; Pinta el marcador.
; Cada número consta de 1 byte de ancho por 16 de alto.
; Altera el valor de los registros AF, BC, DE y HL.
; -----------------------------------------------------------------------------
PrintPoints:
call    printPoint_1_print    ; Pinta el marcador del jugador 1
jr      printPoint_2_print    ; Pinta el marcador del jugador 2

printPoint_1_print:
```

```
ld      a, (p1points)  ; Carga en A los puntos del jugador 1
call    GetPointSprite ; Obtiene el sprite a pintar en el marcador
; 1er dígito del jugador 1
ld      e, (hl)                ; Carga en E la parte baja de la dirección
                               ; donde está el primer dígito
inc     hl                     ; Apunta HL a la parte alta de la dirección
                               ; donde está el primer dígito
ld      d, (hl)                ; y la carga en D
push    hl                     ; Preserva el valor de HL
ld      hl, POINTS_P1          ; Carga en HL la dirección de memoria donde se pintan
                               ; los puntos del jugador 1

call    PrintPoint             ; Pinta el primer dígito del marcador del jugador 1


pop     hl                     ; Recupera el valor de HL
; 2º dígito del jugador 1
inc     hl                     ; Apunta HL a la parte baja de la dirección
                               ; donde está el segundo dígito
ld      e, (hl)                ; y la carga en E
inc     hl                     ; Apunta HL a la parte alta de la dirección
                               ; donde está el segundo dígito
ld      d, (hl)                ; y la carga en D
; Spirax
ld      hl, POINTS_P1 + 1      ; Carga en HL la dirección de memoria donde se pinta
                               ; el segundo dígito de los puntos del jugador 1
call    PrintPoint             ; Pinta el segundo dígito del marcador del jugador 1

ret

printPoint_2_print:
; 1er dígito del jugador 2
ld      a, (p2points)          ; Carga en A los puntos del jugador 2
call    GetPointSprite         ; Obtiene el sprite a pintar en el marcador
ld      e, (hl)                ; Carga en E la parte baja de la dirección
                               ; donde está el primer dígito
inc     hl                     ; Apunta HL a la parte alta de la dirección
                               ; donde está el primer dígito
ld      d, (hl)                ; y la carga en D
push    hl                     ; Preserva el valor de HL
ld      hl, POINTS_P2          ; Carga en HL la dirección de memoria donde se pintan
                               ; los puntos del jugador 2
```

```
call    PrintPoint            ; Pinta el primer dígito del marcador del jugador 2

pop     hl                    ; Recupera el valor de HL
; 2º dígito del jugador 2
inc     hl                    ; Apunta HL a la parte baja de la dirección
                              ; donde está el segundo dígito
ld      e, (hl)               ; y la carga en E
inc     hl                    ; Apunta HL a la parte alta de la dirección
                              ; donde está el segundo dígito
ld      d, (hl)               ; y la carga en D
; Spirax
ld      hl, POINTS_P2 + 1     ; Carga en HL la dirección de memoria donde se pinta
                              ; el segundo dígito de los puntos del jugador 2
; Pinta el segundo dígito del marcador del jugador 2

PrintPoint:
ld      b, $10                ; Cada dígito son 1 byte por 16 (scanlines)

printPoint_printLoop:
ld      a, (de)               ; Carga en A el byte a pintar
ld      (hl), a               ; Pinta el byte
inc     de                    ; Apunta DE al siguiente byte
call    NextScan              ; Apunta HL al siguiente scanline
djnz    printPoint_printLoop  ; Hasta que B = 0

ret

; ------------------------------------------------------------------------------
; Repinta el marcador.
; Cada número consta de 1 byte de ancho por 16 de alto.
; Altera el valor de los registros AF, BC, DE y HL.
; ------------------------------------------------------------------------------
ReprintPoints:
ld      hl, (ballPos)         ; Carga en HL la posición de la bola
call    GetPtrY               ; Obtiene tercio, línea y scanline de la posición
                              ; de la bola
cp      POINTS_Y_B            ; Lo compara con el límite inferior del marcador
ret     nc                    ; Si no hay acarreo, pasa por debajo y sale
ld      a, l                  ; Carga en A la línea y columna de la posición de la bola
and     $1f                   ; Se queda con la columna
cp      POINTS_X1_L           ; Lo compara con el límite izquierdo del marcador 1
```

```
ret     c                     ; Si hay acarreo, pasa por la izquierda y sale
jr      z, printPoint_1_print ; Si es 0, está justo en el margen izquierdo
                              ; y salta para pintar

cp      POINTS_X2_R           ; Lo compara con el límite derecho de marcador 2
jr      z, printPoint_2_print ; Si es 0, está justo en el margen derecho
                              ; y salta para pintar
ret     nc                    ; Si no hay acarreo, pasa por la derecha y sale

reprintPoint_1:
cp      POINTS_X1_R           ; Lo compara con el límite derecho de marcador 1
jr      z, printPoint_1_print
jr      c, printPoint_1_print ; Si es 0 o hay acarreo, pasa por el marcador 1
                              ; y salta para pintar
reprintPoint_2:
cp      POINTS_X2_L           ; Lo compara con el límite derecho de marcador 2
ret     c                     ; Si hay acarreo, pasa por la izquierda y sale
; Spirax
jr      printPoint_2_print    ; Pinta el marcador del jugador 2
```

Si comparamos la implementación de ReprintPoints con la implementación que hicimos de esta rutina en el capítulo anterior, podemos observar que la rutina se ha simplificado significativamente, quedando prácticamente reducida a las comprobaciones que incluimos para que el marcador solo se repintase cuando es necesario.

Ahora ya solo queda compilar, cargar en el emulador y comprobar que todo sigue funcionando.

Hemos terminado. ¿O queréis añadir una pantalla de carga?

Paso 12: pantalla de carga

Hemos dejado para el final la inclusión de una pantalla de carga para nuestro PorompomPong.

Como siempre, creamos una nueva carpeta llamada Paso12 y copiamos todos los archivos .asm desde la carpeta Paso11.

En esta ocasión vamos a cambiar de emulador, vamos a usar Retro Virtual Machine. ¿Por qué este cambio? Vamos a realizar un cargador BASIC personalizado para añadir nuestra pantalla de carga, y Retro Virtual Machine me parece que se acerca visualmente más a lo que usábamos de pequeños, Computone incluido.

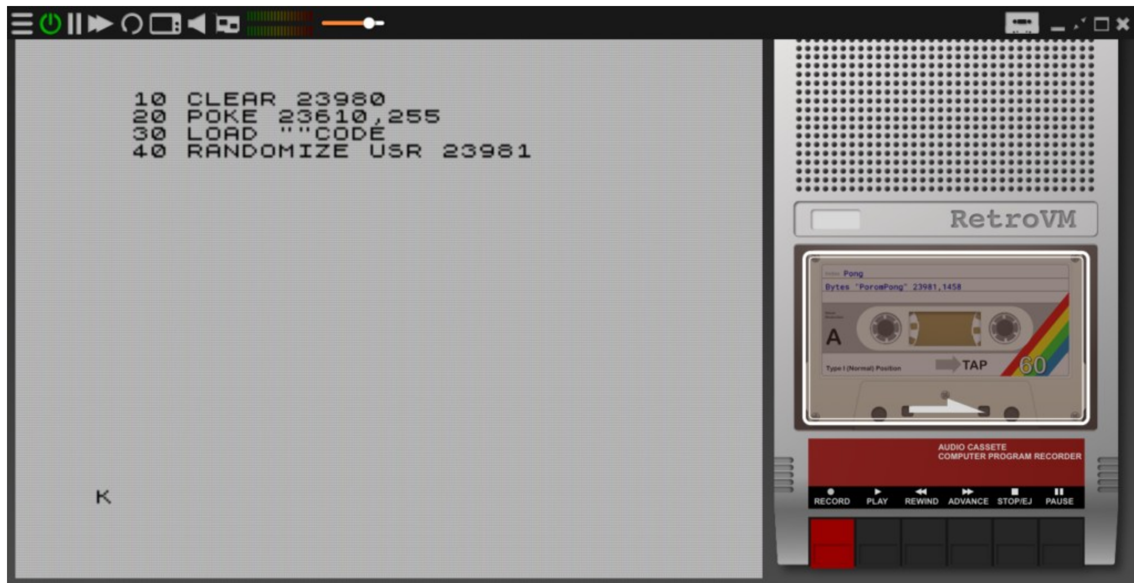
Antes de nada, debéis descargar y descomprimir la pantalla de carga que he preparado o hacer una vosotros. No soy grafista, así que no esperéis gran cosa, aunque cumple con el cometido de esta entrega.

Implementamos nuestro cargador

Para crear el cargador no vamos a partir desde cero, vamos a modificar el cargador que crea PASMO con la opción --tapbas.

Cargamos nuestro programa en Retro Virtual Machine, pero en lugar de cargarlo con **LOAD""**, lo vamos a cargar con **MERGE""**. Para poder escribir **MERGE**, si estáis trabajando en un PC, tenéis que pulsar las teclas Control y Mayúsculas para pasar a modo extendido, y luego Control + T para escribir **MERGE**. Después de **MERGE** hay que poner "", esto lo logramos pulsando la tecla P con la tecla Control pulsada. Un vez que tenemos escrito **MERGE""**, pulsamos Enter y ya podemos cargar el cargador que nos crea PASMO.

Cuando carga el primer bloque, paramos el reproductor y pulsamos Enter para quitar el mensaje 0 OK, 0:1 y ver el código del cargador.



Lo primero que vamos a hacer el editar la primera línea, para lo cual pulsamos el cursor arriba y una vez seleccionada, Mayúsculas + 1 para editar.

Vamos a cambiar la dirección donde se inicia el programa ya que, al meter más BASIC, es necesario cargar el programa en una posición de memoria más alta. Vamos a cambiar el **CLEAR 23980**, por **CLEAR 24059**. Seguidamente vamos a cambiar la línea 40 para poner la dirección de memoria donde cargar el programa, de manera que **RANDOMIZE USR 23981** la dejamos como **RANDOMIZE USR 24060**.

Ahora vamos a ampliar la línea 20. Editamos la línea, nos vamos al final y agregamos dos puntos, pulsando Control + Z. Seguido a los dos puntos, vamos a añadir **POKE 23624, 0**, que se obtiene con Mayúsculas + O y la coma con la coma. Con este POKE ponemos tinta y fondo negro en la línea de comandos del ZX Spectrum, también ponemos en negro el borde; la dirección 23624 es donde esta la variable de sistema que en la entrega 0x0A llamamos BORDCR.

Seguimos en la línea 20 y vamos a poner otro **POKE** (no olvidéis los dos puntos) para poner un valor en la variable de sistema donde se ponen los atributos permanentes de la pantalla, **POKE 23693, 0**, para poner la tinta en negro y el fondo en negro en toda la pantalla.

Por último, ponemos otros dos puntos y **CLS**, que obtenemos pulsando la V.

```
  10 CLEAR 24059
  20>POKE 23610,255: POKE 23624,
0: POKE 23693,0: CLS
  30 LOAD ""CODE
  40 RANDOMIZE USR 24060
```

Vamos a modificar la línea 30, y antes de **LOAD ""CODE**, vamos a añadir **LOAD ""SCREEN$:** (LOAD se obtiene pulsando la J y **SCREEN$** pulsando Control + K en modo extendido).

Después de **SCREEN$**, vamos a añadir otro **POKE** para que no se muestren en pantalla el resto de bloques que se van a cargar, y de esta forma que no borren parte de nuestra pantalla de carga. Este **POKE** en concreto lo he tomado prestado de uno de los vídeos de AsteroideZX; **POKE 23739, 111**.

Es el momento de grabar los cambios en una cinta, en este caso concreto la vamos a llamar PongCargador.tap, ya que vamos a crear otras dos cintas: una para la pantalla de carga y otra para el programa.

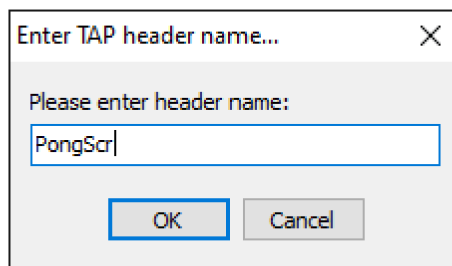
Para grabar ponemos **SAVE "PoromPong" LINE 10**. **SAVE** se obtiene en la S y **LINE** se obtiene pulsando Control + 3 en modo extendido. Con esto grabamos el programa en cinta y después, al cargarlo con **LOAD ""**, se ejecuta desde la línea 10.

```
  10 CLEAR 24059
  20 POKE 23610,255: POKE 23624,
0: POKE 23693,0: CLS
  30>LOAD ""SCREEN$ : POKE 23739
,111: LOAD ""CODE
  40 RANDOMIZE USR 24060
```

Reiniciamos el emulador y probamos lo grabado, veréis que se va a auto ejecutar y se queda esperando a que se siga cargando el resto del programa. Pulsamos Escape para parar la ejecución y pulsamos B, luego 7 y Enter, para poner el borde en blanco. Luego en modo extendido Control + X, luego 7 y Enter, y ya tenemos la tinta de la pantalla principal en blanco.

Añadimos la pantalla de carga

La pantalla de carga la he diseñado con [ZX PaintBrush]() y la he exportado como PongPantalla.tap; me ha pedido que ponga un nombre al bloque y he puesto PongScr.



Es muy importante exportar como tap y no grabar como tap. Si grabamos como tap no nos pide el nombre de la cabecera y luego no cargaría.

Ahora vamos a concatenar el cargador y la pantalla a través de Copy, con el símbolo de sistema (en mi caso que es Windows); no olvidéis situaros en el directorio de trabajo.

```
copy /b PongCargador.tap+PongPantalla.tap Pong.tap
```

Hemos concatenado los archivos PongCargador.tap y PongPantalla.tap y el resultado lo hemos escrito en Pong.tap.

El siguiente paso es cargar en el emulador el archivo Pong.tap para ver si carga nuestra pantalla.

Incluimos nuestro PorompomPong

Ahora solo queda compilar nuestro PorompomPong con PASMO, sin que nos cree el cargador BASIC.

Antes de nada abrimos el archivo Main.asm y cambiamos en la primera línea el **ORG $5dad** por **ORG $5dfc (24060)**, que es la dirección donde se debe cargar nuestro programa según hemos indicado en el cargador.

Es el momento de compilar con PASMO pero cambiando el --tapbas que veníamos usando hasta ahora por --tap.

```
pasmo --name PoromPong --tap Main.asm PongBytes.tap
```

Con esto compilamos pero PASMO no nos crea un cargador BASIC.

Por último vamos a concatenar los tres archivos en uno solo y comprobar si nuestro PorompomPong sigue funcionado.

```
copy /b PongCargador.tap+PongPantalla.tap+PongBytes.tap Pong.tap
```

Si todo ha ido bien, cargamos en el emulador y después de la pantalla de carga, se cargará nuestro programa y todo listo para jugar.

Apéndice 1: frecuencias y notas

A continuación, se muestran los valores para las frecuencias y las notas de 8 escalas distintas. Cuanto menor es la escala, es más grave, y cuanto mayor, más aguda.

Los valores de las frecuencias se han obtenido de Dilwyn Jones Sinclair QL Pages.

Estas frecuencias indican la duración de la nota, que en estos casos son de un segundo.

Para calcular cada nota, la fórmula es la siguiente:

```
(437500/frecuencia)-30.125    ; . es punto decimal
```

Esta fórmula se ha tomado del libro “Programación Avanzada del ZX Spectrum”, de Steve Kramer. Podéis encontrar dicha fórmula en la página 18.

A continuación, se muestra una tabla con frecuencias y notas, en decimal, hexadecimal y código ensamblador.

		Frecuencia, 1 segundo, cargar en DE			Nota, cargar en HL		
		Decimal	Hexadecimal	PASMO Z80 Assembler	Decimal	Hexadecimal	PASMO Z80 Assembler
C	0	16,35	0010	C_0_FQ: EQU $0010	26728,28	6868	C_0: EQU $6868
Cs	0	17,32	0011	Cs_0_FQ: EQU $0011	25229,69	628D	Cs_0: EQU $628D
D	0	18,35	0012	D_0_FQ: EQU $0012	23811,84	5D03	D_0: EQU $5D03
Ds	0	19,45	0013	Ds_0_FQ: EQU $0013	22463,45	57BF	Ds_0: EQU $57BF
E	0	20,60	0014	E_0_FQ: EQU $0014	21207,74	52D7	E_0: EQU $52D7
F	0	21,83	0015	F_0_FQ: EQU $0015	20011,10	4E2B	F_0: EQU $4E2B
Fs	0	23,12	0017	Fs_0_FQ: EQU $0017	18892,89	49CC	Fs_0: EQU $49CC
G	0	24,50	0018	G_0_FQ: EQU $0018	17827,02	45A3	G_0: EQU $45A3
Gs	0	25,96	0019	Gs_0_FQ: EQU $0019	16822,73	41B6	Gs_0: EQU $41B6
A	0	27,50	001B	A_0_FQ: EQU $001B	15878,97	3E06	A_0: EQU $3E06
As	0	29,14	001D	As_0_FQ: EQU $001D	14983,60	3A87	As_0: EQU $3A87
B	0	30,87	001E	B_0_FQ: EQU $001E	14142,21	373E	B_0: EQU $373E
C	1	32,70	0020	C_1_FQ: EQU $0020	13349,08	3425	C_1: EQU $3425
Cs	1	34,65	0022	Cs_1_FQ: EQU $0022	12596,14	3134	Cs_1: EQU $3134
D	1	36,71	0024	D_1_FQ: EQU $0024	11887,61	2E6F	D_1: EQU $2E6F
Ds	1	38,89	0026	Ds_1_FQ: EQU $0026	11219,55	2BD3	Ds_1: EQU $2BD3
E	1	41,20	0029	E_1_FQ: EQU $0029	10588,81	295C	E_1: EQU $295C
F	1	43,65	002B	F_1_FQ: EQU $002B	9992,78	2708	F_1: EQU $2708
Fs	1	46,25	002E	Fs_1_FQ: EQU $002E	9429,33	24D5	Fs_1: EQU $24D5
G	1	49,00	0031	G_1_FQ: EQU $0031	8898,45	22C2	G_1: EQU $22C2
Gs	1	51,91	0033	Gs_1_FQ: EQU $0033	8397,92	20CD	Gs_1: EQU $20CD
A	1	55,00	0037	A_1_FQ: EQU $0037	7924,42	1EF4	A_1: EQU $1EF4
As	1	58,27	003A	As_1_FQ: EQU $003A	7478,03	1D36	As_1: EQU $1D36
B	1	61,74	003D	B_1_FQ: EQU $003D	7056,04	1B90	B_1: EQU $1B90

C	2	65,41	0041	C_2_FQ: EQU $0041	6658,45	1A02	C_2: EQU $1A02
Cs	2	69,30	0045	Cs_2_FQ: EQU $0045	6283,01	188B	Cs_2: EQU $188B
D	2	73,42	0049	D_2_FQ: EQU $0049	5928,74	1728	D_2: EQU $1728
Ds	2	77,78	004D	Ds_2_FQ: EQU $004D	5594,71	15DA	Ds_2: EQU $15DA
E	2	82,41	0052	E_2_FQ: EQU $0052	5278,70	149E	E_2: EQU $149E
F	2	87,31	0057	F_2_FQ: EQU $0057	4980,76	1374	F_2: EQU $1374
Fs	2	92,50	005C	Fs_2_FQ: EQU $005C	4699,60	125B	Fs_2: EQU $125B
G	2	98,00	0062	G_2_FQ: EQU $0062	4434,16	1152	G_2: EQU $1152
Gs	2	103,80	0067	Gs_2_FQ: EQU $0067	4184,71	1058	Gs_2: EQU $1058
A	2	110,00	006E	A_2_FQ: EQU $006E	3947,15	0F6B	A_2: EQU $0F6B
As	2	116,00	0074	As_2_FQ: EQU $0074	3741,43	0E9D	As_2: EQU $0E9D
B	2	123,50	007B	B_2_FQ: EQU $007B	3512,39	0DB8	B_2: EQU $0DB8
C	3	130,80	0082	C_3_FQ: EQU $0082	3314,68	0CF2	C_3: EQU $0CF2
Cs	3	138,60	008A	Cs_3_FQ: EQU $008A	3126,44	0C36	Cs_3: EQU $0C36
D	3	146,80	0092	D_3_FQ: EQU $0092	2950,12	0B86	D_3: EQU $0B86
Ds	3	155,60	009B	Ds_3_FQ: EQU $009B	2781,57	0ADD	Ds_3: EQU $0ADD
E	3	164,80	00A4	E_3_FQ: EQU $00A4	2624,61	0A40	E_3: EQU $0A40
F	3	174,60	00AE	F_3_FQ: EQU $00AE	2475,60	09AB	F_3: EQU $09AB
Fs	3	185,00	00B9	Fs_3_FQ: EQU $00B9	2334,74	091E	Fs_3: EQU $091E
G	3	196,00	00C4	G_3_FQ: EQU $00C4	2202,02	089A	G_3: EQU $089A
Gs	3	207,70	00CF	Gs_3_FQ: EQU $00CF	2076,28	081C	Gs_3: EQU $081C
A	3	220,00	00DC	A_3_FQ: EQU $00DC	1958,51	07A6	A_3: EQU $07A6
As	3	233,10	00E9	As_3_FQ: EQU $00E9	1846,75	0736	As_3: EQU $0736
B	3	246,90	00F6	B_3_FQ: EQU $00F6	1741,85	06CD	B_3: EQU $06CD
C	4	261,60	0105	C_4_FQ: EQU $0105	1642,28	066A	C_4: EQU $066A
Cs	4	277,20	0115	Cs_4_FQ: EQU $0115	1548,16	060C	Cs_4: EQU $060C
D	4	293,70	0125	D_4_FQ: EQU $0125	1459,49	05B3	D_4: EQU $05B3
Ds	4	311,10	0137	Ds_4_FQ: EQU $0137	1376,18	0560	Ds_4: EQU $0560
E	4	329,60	0149	E_4_FQ: EQU $0149	1297,24	0511	E_4: EQU $0511
F	4	349,20	015D	F_4_FQ: EQU $015D	1222,74	04C6	F_4: EQU $04C6
Fs	4	370,00	0172	Fs_4_FQ: EQU $0172	1152,31	0480	Fs_4: EQU $0480
G	4	392,00	0188	G_4_FQ: EQU $0188	1085,95	043D	G_4: EQU $043D
Gs	4	415,30	019F	Gs_4_FQ: EQU $019F	1023,33	03FF	Gs_4: EQU $03FF
A	4	440,00	01B8	A_4_FQ: EQU $01B8	964,19	03C4	A_4: EQU $03C4
As	4	466,20	01D2	As_4_FQ: EQU $01D2	908,31	038C	As_4: EQU $038C
B	4	493,90	01ED	B_4_FQ: EQU $01ED	855,68	0357	B_4: EQU $0357
C	5	523,30	020B	C_5_FQ: EQU $020B	805,92	0325	C_5: EQU $0325
Cs	5	554,40	022A	Cs_5_FQ: EQU $022A	759,02	02F7	Cs_5: EQU $02F7
D	5	587,30	024B	D_5_FQ: EQU $024B	714,81	02CA	D_5: EQU $02CA
Ds	5	622,30	026E	Ds_5_FQ: EQU $026E	672,91	02A0	Ds_5: EQU $02A0

E	5	659,30	0293	E_5_FQ: EQU $0293	633,46	0279	E_5: EQU $0279
F	5	698,50	02BA	F_5_FQ: EQU $02BA	596,22	0254	F_5: EQU $0254
Fs	5	740,00	02E4	Fs_5_FQ: EQU $02E4	561,09	0231	Fs_5: EQU $0231
G	5	784,00	0310	G_5_FQ: EQU $0310	527,91	020F	G_5: EQU $020F
Gs	5	830,00	033E	Gs_5_FQ: EQU $033E	496,98	01F0	Gs_5: EQU $01F0
A	5	880,00	0370	A_5_FQ: EQU $0370	467,03	01D3	A_5: EQU $01D3
As	5	932,30	03A4	As_5_FQ: EQU $03A4	439,14	01B7	As_5: EQU $01B7
B	5	987,80	03DB	B_5_FQ: EQU $03DB	412,78	019C	B_5: EQU $019C
C	6	1047,00	0417	C_6_FQ: EQU $0417	387,74	0183	C_6: EQU $0183
Cs	6	1109,00	0455	Cs_6_FQ: EQU $0455	364,37	016C	Cs_6: EQU $016C
D	6	1175,00	0497	D_6_FQ: EQU $0497	342,22	0156	D_6: EQU $0156
Ds	6	1245,00	04DD	Ds_6_FQ: EQU $04DD	321,28	0141	Ds_6: EQU $0141
E	6	1319,00	0527	E_6_FQ: EQU $0527	301,57	012D	E_6: EQU $012D
F	6	1397,00	0575	F_6_FQ: EQU $0575	283,05	011B	F_6: EQU $011B
Fs	6	1480,00	05C8	Fs_6_FQ: EQU $05C8	265,48	0109	Fs_6: EQU $0109
G	6	1568,00	0620	G_6_FQ: EQU $0620	248,89	00F8	G_6: EQU $00F8
Gs	6	1661,00	067D	Gs_6_FQ: EQU $067D	233,27	00E9	Gs_6: EQU $00E9
A	6	1760,00	06E0	A_6_FQ: EQU $06E0	218,45	00DA	A_6: EQU $00DA
As	6	1865,00	0749	As_6_FQ: EQU $0749	204,46	00CC	As_6: EQU $00CC
B	6	1976,00	07B8	B_6_FQ: EQU $07B8	191,28	00BF	B_6: EQU $00BF
C	7	2093,00	082D	C_7_FQ: EQU $082D	178,91	00B2	C_7: EQU $00B2
Cs	7	2217,00	08A9	Cs_7_FQ: EQU $08A9	167,21	00A7	Cs_7: EQU $00A7
D	7	2349,00	092D	D_7_FQ: EQU $092D	156,12	009C	D_7: EQU $009C
Ds	7	2489,00	09B9	Ds_7_FQ: EQU $09B9	145,65	0091	Ds_7: EQU $0091
E	7	2637,00	0A4D	E_7_FQ: EQU $0A4D	135,78	0087	E_7: EQU $0087
F	7	2794,00	0AEA	F_7_FQ: EQU $0AEA	126,46	007E	F_7: EQU $007E
Fs	7	2960,00	0B90	Fs_7_FQ: EQU $0B90	117,68	0075	Fs_7: EQU $0075
G	7	3136,00	0C40	G_7_FQ: EQU $0C40	109,38	006D	G_7: EQU $006D
Gs	7	3322,00	0CFA	Gs_7_FQ: EQU $0CFA	101,57	0065	Gs_7: EQU $0065
A	7	3520,00	0DC0	A_7_FQ: EQU $0DC0	94,16	005E	A_7: EQU $005E
As	7	3729,00	0E91	As_7_FQ: EQU $0E91	87,20	0057	As_7: EQU $0057
B	7	3951,00	0F6F	B_7_FQ: EQU $0F6F	80,61	0050	B_7: EQU $0050
C	8	4186,00	105A	C_8_FQ: EQU $105A	74,39	004A	C_8: EQU $004A
Cs	8	4435,00	1153	Cs_8_FQ: EQU $1153	68,52	0044	Cs_8: EQU $0044
D	8	4699,00	125B	D_8_FQ: EQU $125B	62,98	003E	D_8: EQU $003E
Ds	8	4978,00	1372	Ds_8_FQ: EQU $1372	57,76	0039	Ds_8: EQU $0039
E	8	5274,00	149A	E_8_FQ: EQU $149A	52,83	0034	E_8: EQU $0034
F	8	5588,00	15D4	F_8_FQ: EQU $15D4	48,17	0030	F_8: EQU $0030
Fs	8	5920,00	1720	Fs_8_FQ: EQU $1720	43,78	002B	Fs_8: EQU $002B
G	8	6272,00	1880	G_8_FQ: EQU $1880	39,63	0027	G_8: EQU $0027
G	8	6645,00	19F5	Gs_8_FQ: EQU $19F5	35,71	0023	Gs_8: EQU $0023

s							
A	8	7040,00	1B80	A_8_FQ: EQU $1B80	32,02	0020	A_8: EQU $0020
As	8	7459,00	1D23	As_8_FQ: EQU $1D23	28,53	001C	As_8: EQU $001C
B	8	7902,00	1EDE	B_8_FQ: EQU $1EDE	25,24	0019	B_8: EQU $0019

Bibliografía

Curso de Ensamblador Z80 de Compiler Software de Santiago Romero

https://wiki.speccy.org/cursos/ensamblador/indice

Santiago Romero

Dilwyn Jones Sinclair QL Pages

http://www.dilwyn.me.uk/

http://www.dilwyn.me.uk/docs/articles/beeps.pdf

Programación avanzada del ZX Spectrum, Rutinas de la ROM y Sistema Operativo

© Steve Kramer, 1984

© Ediciones Anaya Multimedia, S.A., 1985