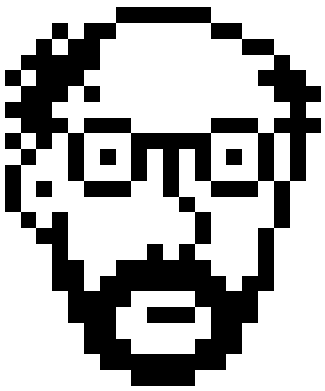
Videojuegos en lenguaje ensamblador para negados

Creación de videojuegos en ZX Spectrum de forma profesional para gente sin conocimientos previos de programación



Bienvenido

Estimado alumno, ha decidido entrar en el fascinante mundo de la programación de videojuegos en lenguaje ensamblador. Le felicitamos por su valentia, arrojo, y ganas de sufrimiento... estooo, por su coraje para crear videojuegos de forma profesional.

El curso estaba dividido originalmente en 728 entregas, de las cuales el 60% eran teoría sobre procesadores, acceso a memoria, y rutinas avanzadas de e/s, y el 40% restante ejercícios prácticos con no menos de 500 líneas de código. Se llamaba "Curso avanzado de programación en lenguaje ensamblador y resolución de criptogramas cuánticos". Aquel curso no conseguimos que ningún alumno pasara de la primera lección ("Algebra de Boole mediante integrales de 5º grado").

A continuación realizamos una remodelación del plan de estudios, obteniendo como resultado el "Curso de aprendizaje de programación en ensamblador para ZX Spectrum". El curso constaba de 177 entregas, y regalábamos un póster de Sir Clive Sinclair haciendo topless en Ibiza. Curisamente no se llegó a matricular ningún alumno, y eso que usamos el poster de Sir Clive como cartel anunciador.

En nuestro tercer intento realizamos un curso de 83 entregas titulado "Videojuegos en ensamblador para principiantes". Con el curso se regalaban vales para tomarse unas cervezas al principio de cada lección... y fué un total éxito. El bar de la academia no daba a basto, la pega es que los profesores también se hiban de cañas por falta de alumnos.

Es por ello que hemos realizado este curso "Videojuegos en lenguaje ensamblador para negados". No hay regalos, y las lecciones no pasarán de la docena.

Conocimientos previos

Para la realización de este cursillo es necesario:

- el kit de programación que adjuntamos
- tener manos para poder escribir
- saber leer

Da exactamente igual que haya programado o no alguna vez en su vida, tampoco que conozca como se usa un programa ensamblador, ni mucho menos que tenga graduado escolar (el que escribe estas líneas sólo tiene el certificado de escolaridad... quien hiba a pensar que luego sería ingeniero).

Al grano

Eso es todo, puede usted pasar a la primera lección y comenzar a realizar videojuegos profesionales con conocimientos nulos del lenguaje.

Un cordial saludo

Radastan

Ensamblando

Preparando el ensamblador

En su fantástico paquete de iniciación, con el texto de introducción, las postales de Doraemon, y un Z80 quemado por overclocking en nuestros laboratorios, debe usted encontrar un fichero ZIP denominado "kit_ensamblador.zip".

En caso contrario puede usted encontrar todo en:

http://www.bytemaniacos.com/ficheros/curso_asm/

Una vez descomprimido a un directorio encontrará varios ficheros, de los cuales sólo le interesan:

compilar.bat
programa.asm

Compilando

Si usted ejecuta "compilar.bat" advertirá que se crean dos nuevos ficheros, de los cuales sólo le interesa:

programa.tap

dicho fichero es un ejecutable que se puede usar con cualquier emulador de ZX Spectrum o reproducir con Winamp para cargarse en un ZX Spectrum real como si fuera una cinta. En este último caso necesitaría el plug-in:

http://www.winamp.com/plugins/details.php?id=9186

En la primera ejecución debería obtener la letra "o" en pantalla, la cual puede mover con las teclas "q a o p".

Editando sus programas

Si edita el fichero "programa.asm" con cualquier procesador de textos (ej. Notepad de Windows). Advertirá que dentro hay un montón de insultos sin sentido, denominado lenguaje ensamblador, y un texto sin pies ni cabeza tras ";", los cuales llamaremos comentarios.

Los programas sólo se componen de estos dos tipos de contenidos: lenguaje ensamblador y comentarios. Usted puede realizar programas sin comentarios, no son necesarios para el ensamblador ni para el propio programa, pero si para mantener su cordura en el desarrollo de su creación.

Le rogamos usar tantos comentarios como le sean precisos, no se corte.

Una vez editado su programa guarde una copia del fichero en otra parte con el nombre

que le venga en gana y sobreescriba el fichero programa.asm. Tras esto sólo tiene que ejecutar "compilar.bat" y ya tendrá un fichero TAP listo para su uso.

Errores

Si usted teclea mal un comando de ensamblador, o ha realizado algo incorrectamente, verá que la compilación muestra un mensaje de error y el fichero programa.tap no se crea o no se actualiza con los cambios. En dicho caso el ensamblador proporcionará el número de línea donde se encuentra el error.

Un primer intento

Vamos a realizar una primera incursión en el lenguaje ensamblador.
No se asuste, será algo sencillo.

Localice la línea:

```
ld a, 79
```

está casi al final del programa.

Cambie el 79 por 81, grabe el fichero programa.asm, y vuelva a ensamblar, ahora en vez de una "o" debería poder mover una "q".

Ahora vamos a localizar la línea:

```
ld a, 1          ; borde azul, para actualización inmediata
```

está por el principio del programa.

Cambie el 1 por un 2, grabe el fichero programa.asm, y vuelva a ensamblar, ahora en vez de un borde azul debería obtener unborde de color rojo.

¿Ve que fácil?

En la próxima lección le enseñaremos desde cero como realizar un programa en ensamblador sin haber programado antes.

Comenzando

Nuestro primer programa

Bien, ya sabemos como ensamblar nuestro programa y estamos ansiosos por saber programar. En otros cursos se frenaría al alumno y se le daría unas nociones de álgebra digital, arquitectura del Z80, o fabricación de cócteles Molotov. Pero aquí no, vamos al grano.

Vamos a describir como se realiza lo básico, el esqueleto de un programa en ensamblador:

```
; Nuestro primer programa

        ORG 32768

      RET
```

Ya está, así de simple, si lo ensambla verá que el ZX Spectrum retornará a BASIC sin haber hecho ABSOLUTAMENTE NADA. ¿No es genial? ¡¡NADAAAAAAAAAAAA!!

¿Y el ORG 32768 ese? ¿para qué sirve? si se hace esta pregunta le daremos una respuesta muy sencilla: es la posición de memoria donde el programa va a estar y es requerida por el programa ensamblador. De momento no cambie dicha dirección, si lo hace verá que su programa puede que no funcione, deje esas cosas para otro día.

¿Y el RET? eso es ensamblador, de hecho la única línea de código que hay en el programa. Su función es muy simple: retornar. En este caso, como es la última línea, retornará al BASIC. Es más, nuestro programa es una subrutina que llamamos desde BASIC con randomize usr 32768 (no hace falta que recuerde esto).

Estructura de la memoria

Vamos a ver como es la estructura básica de la memoria de un ZX Spectrum:

00000 - Inicio de la ROM (sistema operativo BASIC)
16383 - Fin de la ROM

16384 - Inicio de la memoria de pantalla
23295 - Fin de la memoria de pantalla

23296 - Comienzo de la memoria libre
65535 - Fin de la memoria libre

Con esto tenemos lo básico para trabajar.

Como puede ver la ROM está entre 0 y 16383, ahí estan alojadas todas las subrutinas que BASIC usa, entre otras cosas. Poco a poco le iremos mostrando cuales son las subrutinas más interesantes y cómo se usan.

La pantalla está entre 16384 y 23295. Su estructura no la vamos a explicar poque no vamos a necesitarlo, le proporcionaremos directamente las rutinas para manejar gráficos y punto.

La memoria libre empieza a partir de 23296, pero nuestros programas no los comenzaremos hasta bastante después. ¿por qué? porque si lo hacemos ya no podemos usar las subrutinas de la ROM y volver al BASIC, puesto que el comienzo de este espacio se usa para datos del BASIC (y por ende de subrutinas de la ROM). Hemos escogido 32768 para empezar, lo que nos da 32K de espacio para nuestro programa, más que suficiente para un juego sencillo. ¿Y los 128K? nos olvidamos de ellos de momento, si consigue realizar un juego para 48K nos damos por satisfechos.

Una buena idea es plasmar el mapa de memoria en un papel. Dibuje un rectángulo alargado verticalmente, el cual debe dividir en 4 partes iguales. Cada parte se corresponden con 16K de memoria, haciendo 64K en total:

- El primer bloque sería la ROM
- El segundo la pantalla, los datos de la ROM, y el programa BASIC
- El tercer y cuarto bloque sería nuestro programa

Nuestro segundo programa

```
; Nuestro segundo programa

        ORG 32768

      ld a, 15          ; papel azul y tinta en blanco
      ld (23693),a      ; actualizamos la variable de atributos en pantalla
      ld (23624),a      ; actualizamos la variable de atributos en pantalla
                        ; (parte baja)
      call 3503

      RET
```

¿Y esto? ¿pero que co%$? tranquilo, vamos a ir poco a poco. Si lo ensambla y ejecuta verá que el fondo de la pantalla cambia a color azul, así de simple. Ahora se lo explicamos:

```
ld a,15
```

A esto llamamos una instrucción de carga, coje un valor (en este caso 15) y lo introduce en el registro A. Y usted se acaba de perder, no sabe lo que son los registros. No nos hace falta aún, imagine que ha metido el valor 15 en una hoja de papel que hay dentro del ordenador, y que dicha hoja la usa para lo que usted quiera.

```
ld (23693),a
```

Esto es otra instrucción de carga, pero más rara todavia, lo que hace es introducir esa hoja de papel que hemos escrito (a) en la posición de memoria 23693. Por un casual dicha dirección es la que usa el ordenador para almacenar los colores con que se escribe en BASIC en pantalla.

```
ld (23624),a
```

Otra vez lo mismo, pero en esta ocasión hemos actualizado la parte inferior de la pantalla. BASIC es así de latazo, divide la pantalla en dos partes: programa y editor. Observe una cosa curiosa, lo que vamos a utilizar siempre está a la derecha de la instrucción en ensamblador, y el destino a la izquierda.

```
call 3503
```

¡Ajá! ¡eso es para llamar por teléfono a una línea erótica! no, pero casi. call realiza una llamada, si, pero a una subrutina (un programita que hace algo y tiene al final un RET). 3503 es la dirección de memoria donde está la subrutina que vamos a llamar, es decir estamos llamando a una rutina de la ROM, que corresponde en este caso a un "borrar pantalla" (CLS en BASIC).

Como hemos cambiado los colores con los que escribe el BASIC y hemos llamado a la instrucción CLS... pues la pantalla cambia de color. Este programa en BASIC equivale a:

```
20 ink 1
30 paper 1
40 cls
```

Pero como podemos ver no se corresponde el valor del código de color, ¿el motivo? pues que estamos metiendo el valor de tinta y papel en un sólo número mediante la siguiente tabla:

```
- TINTA -

NEGRO             0     00000000
AZUL              1     00000001
ROJO              2     00000010
MAGENTA           3     00000011
VERDE             4     00000100
AZULADO           5     00000101
AMARILLO          6     00000110
BLANCO            7     00000111

- PAPEL -

NEGRO             0     00000000
AZUL              8     00001000
ROJO              16    00010000
MAGENTA           24    00011000
VERDE             32    00100000
AZULADO           40    00101000
AMARILLO          48    00110000
BLANCO            56    00111000
```

Ej. queremos tinta azul y papel amarillo: 1 + 48 = 49, que es el valor a introducir en "a"

Nuestro tercer programa

Ahora escriba esto:

```
; Nuestro tercer programa

        ORG 32768

      ld a, 1           ; borde azul, para actualización inmediata
      out (254), a

      RET
```

¡Alaaaaaa! ¡no hemos usado ninguna subrutina y hace algo! Pues si, si ensambla y ejecuta este programa verá que el borde la pantalla se vuelve azul, como puede ver de pequeño me gustaban mucho Los Pitufos.

Hemos cargado en "a" el valor 1, que en esta ocasión si se corresponde con el código de colores del BASIC... ¿el motivo? que aquí sólo estamos cambiando un color y no dos, en el anterior ejemplo teníamos que meter en un sólo número dos valores.

La novedad es la sentencia out (254), a. out sirve para sacar por un puerto de comunicación un valor determinado, en este caso introduce en el puerto 254 el valor de "a". Dicho puerto se corresponde con el borde la pantalla, sólo tiene que saber eso.

Nuestro cuarto programa

```
; nuestro cuarto programa

        ORG 32768

      ld a, 15          ; papel azul y tinta en blanco
      ld (23693),a      ; actualizamos la variable de atributos en pantalla
      ld (23624),a      ; actualizamos la variable de atributos en pantalla
                        ; (parte baja)
      call 3503
      ld a, 1           ; borde azul, para actualización inmediata
      out (254), a

      RET
```

Bien, hemos unido todos los programas anteriores y ya tenemos algo "tangible", el fruto de nuestros esfuerzos. Con esto ya podemos preparar la pantalla con los colores que queramos.

Subrutinas

Previo

Tras los múltiples mensajes de protesta, manifestaciones, y dos intentos de asesinato, hemos decidido remodelar esta sección del curso para facilitar aún más el aprendizaje de nuestros alumnos. Rogamos disculpen las molestias ocasionadas, y los daños mentales y psicológicos que puedan ocasionar el presente curso en la mente de nuestros lectores.

Para aquellos que no estén acostumbrados a un ejercicio mental prolongado recomendamos leer una línea cada 30 minutos, y ver el programa Ana Rosa para limpiar la mente antes de cada capítulo.

Esta edición está homologada por la Real academia de la Lengua, habiendo sido revisada personalmente por Jaime Tejedor "Fernado Fernan" Gómez. Gracias por su colaboración.

Subrutinas

No hay nada más liante que empezar a escribir un programa e ir añadiendo líneas, ya que llega un punto en el que uno ya no sabe ni por cual parte está escribiendo, ni a donde conduce todo. Para ello lo más adecuado es usar subrutinas, es decir, dividir el programa en secciones claramente diferenciadas y que podamos reaprovechar en otros programas.

Para empezar vamos a coger la rutina que ya conocemos para cambiar el borde la pantalla:

```
; Nuestro tercer programa

        ORG 32768

      ld a, 1           ; borde azul, para actualización inmediata
      out (254), a

      RET
```

Hasta aquí todo bien, es muy corta... pero es un latazo tener que escribir el mismo código para cada vez que actualicemos el color del borde. Por ello vamos a dividir el programa en la rutina principal, que llamará a la subrutina, y la subrutina en si:

```
; ------------------------------------------
; Programa: Nuestra primera subrutina
; ------------------------------------------

        ORG 32768

; ------------------------------------------
; Rutina principal
; ------------------------------------------

      ld a, 1           ; borde azul, para actualización inmediata
      call borde

      RET

; ------------------------------------------
```

```
; SUBRUTINA PARA CAMBIAR EL BORDE DE PANTALLA
; ENTRADAS: a, es el color del borde
; SALIDAS: ninguna
; ----------------------------------------

borde out (254), a
      ret
```

¡Que no cunda el pánico! vamos a explicar lo que hemos hecho. En primer lugar hemos puesto unos recuadros de texto para diferenciar bien cada parte del código, explicando en su interior que demonios es lo que hay a continuación, no son impresindibles... pero quedan muy bonitos y ayudan bastante a diferenciar cada parte del programa.

Antes de la rutina principal hemos separado el ORG, ya que se trata de una orden para que el ensamblador sepa a partir de qué punto de la memoria empieza el programa.

En la rutina principal tenemos una cosa nueva que no hemos visto antes: "call borde". Suena francamente mal, y su función es llamar a una subrutina llamada "borde". Antes de CALL hemos cargado "a" con el valor 1 (el color del borde) y para terminar hemos realizado un RET que finaliza el programa. Cuando realizamos la llamada a la subrutina no alteramos nada en la memoria, por eso la subrutina verá que "a" tiene un valor que puede usar.

Si nos fijamos en el programa tras CALL hemos usado un nombre y no una dirección, a eso se le llama "etiqueta" en ensamblador. Las etiquetas las usamos nosotros por comodidad, y el ensamblador se encarga de traducirlo por nosotros a una dirección de memoria.

La subrutina la hemos emplazado después del programa principal. ¿Cómo la definimos? simplemente tecleando un nombre de 5 letras de extensión delante de una instrucción:

```
borde out (254), a
```

En esta línea hemos hecho dos cosas: definir una etiqueta y ejecutar una instrucción.

De esta forma se crean las subrutinas, con una etiqueta que sirva como referencia para su llamada, y el código de la misma terminando con un ret para RETornar. Fácil, ¿no?

Podemos usar cuantas subrutinas queramos, siempre por debajo de la rutina principal, y como podeis ver es conveniente que antes de cada subrutina indiquemos para lo que sirve, además de las entradas que usa y las salidas que proporciona (si las hubiera).

Registros y bits

¡NOOOOO! ¡TEORIA NOOOOO! HARAKIRIIIIIIII! Calma, es algo muy simple y escueto.

Hemos visto que existe un registro "a" para usarlo a modo de papel, pero nuestro ordenador es más listo y tiene varios "papeles":

a

b
c

d
e

h
l

Los hemos separado aposta por pares, ¿el motivo? que algunas instrucciones en ensamblador las usan de dos en dos. Por ejemplo para usarlas como direcciones de memoria. La hoja más importante es "a", es la hoja principal de trabajo y con la que más vamos a trabajar.

Todos los registros, por separado, pueden tener un valor entre 0 y 255 en valor decimal, o un valor entre:

00000000

y

11111111

en código binario. Nosotros entendemos el decimal, y la CPU del ordenador sólo entiende el binario. El programa ensamblador se encarga de "traducir" nuestro lenguaje al suyo. No obstante, entender el binario puede resultar útil en muchas ocasiones para realizar ciertas triquiñuelas.

Nuestra segunda subrutina

Para nuestra segunda subrutina vamos a realizar algo más complicado, nos vamos a crear un CLS sin usar ninguna rutina de la ROM:

```
; ----------------------------------------
; SUBRUTINA PARA HACER UN CLS SIN USAR LA ROM
; ENTRADAS: a, es el color de tinta/papel
; SALIDAS: pone el mapa de pantalla a 0
; ----------------------------------------

sucls ld hl, 22528
      ld de, 22529
      ld bc, 767
      ld (hl),a
      ldir
      ld hl, 16384
      ld de, 16385
      ld bc, 6143
      ld (hl),l
      ldir
      ret
```

Antes de ver 10 veces seguidas la película de JFK, para planear perfectamente el asesinato del autor de este curso, esperamos que se nos de una oportunidad (si, tengo doble

personalidad, Gollum no es el único... sssssss).

Si teclamos la rutina dada, tal cual, no hacemos nada. Esto es una subrutina, y hace falta una rutina principal que la llame (o al menos usar un ORG antes para usarla directamente como rutina). La forma de llamar a esta subrutina es muy simple:

```
      call sucls
```

Es decir, el programa más básico que incluya la subrutina sería:

```
; ------------------------------------------
; Programa: Nuestra segunda subrutina
; ------------------------------------------

        ORG 32768

; ------------------------------------------
; Rutina principal
; ------------------------------------------

      call sucls

      RET

; ------------------------------------------
; SUBRUTINA PARA HACER UN CLS SIN USAR LA ROM
; ENTRADAS: a, es el color de tinta/papel
; SALIDAS: pone el mapa de pantalla a 0
; ------------------------------------------

sucls ld hl, 22528
      ld de, 22529
      ld bc, 767
      ld (hl),a
      ldir
      ld hl, 16384
      ld de, 16385
      ld bc, 6143
      ld (hl),l
      ldir
      ret
```

Ahora pasamos a la subrutina en si. Lo que más llama la atención es que usamos muchos registros nuevos, antes sólo usábamos "a", y además por parejas (a eso se le llama registro de 16 bits). Para entenderlo todo empezaremos por la instrucción que menos conocemos, LDIR.

LDIR realiza la acción siguiente: coge el valor de la posición de memoria apuntada por HL y la copia a la dirección de memoria apuntada por DE, tras lo cual incrementa HL y DE (es decir pasa a la siguiente posición). Esta operación se repite tantas veces como se indique en BC (en nuestro caso 767 veces). En pocas palabras, se trata de una instrucción que copia una parte de la memoria a otra. Pensad en ella como si fuera otra subrutina (de hecho podemos llamar desde una subrutina a otras subrutinas).

En nuestro ejemplo hay dos partes de la memoria a borrar: los pixels y los atributos (colores). Los atributos es lo primero que borramos, y esta parte empieza en la posición

de memoria 22528, con una longitud de 768 bytes:

```
ld hl, 22528
ld de, 22529
ld bc, 767
ld (hl),a
ldir
```

Aprovechando la instrucción LDIR hemos copiado con la instrucción:

```
ld (hl),a
```

el valor de "a" a la posición de memoria que apunta HL:

```
ld hl, 22528
```

es decir, 22528 que es el primer byte de la zona de atributos en pantalla. Con ello, si hacemos que LDIR copie continuamente el valor del byte anterior al siguiente (es decir, que copie consecutivamente "a" de una posición a la siguiente) llenaremos todo el espacio de atributos. No tenemos que realizar esta acción 768 veces, que es lo que ocupan los atributos, sino 767 veces, ya que el primer byte de la zona de atributos ya lo hemos puesto con el valor "a" nosotros.

La segunda parte borra la zona de pixels a 0:

```
ld hl, 16384
ld de, 16385
ld bc, 6143
ld (hl),l
ldir
```

La zona de pixels empieza en 16384 y tiene una extensión de 6144 bytes. En esta ocasión hemos usado una argucia para evitar tener que usar "a"... como queremos poner todo a 0, y como casualmente 16384 en binario es 01000000 00000000... y hemos hecho:

```
ld hl, 16384
```

para indicar la posición de memoria donde va a comenzar LDIR... pues reaprovechamos "L". Si no quisiéramos usar esa argucia, podríamos usar:

```
ld hl, 16384
ld de, 16385
ld bc, 6143
ld (hl),b
ldir
```

Pero previamente tendríamos que poner en "b" el valor 0 si queremos borrar todos los pixels (o 255 si quisiéramos tenerlos todos activados).

Ese es el funcionamiento de la subrutina que hemos preparado, pero aún cuando no sepamos como funciona la podemos usar.
Basta preparar los registros que necesita la subrutina a nuestro antojo y realizar el CALL correspondiente. En este caso si realizamos el call y ponemos en "a" antes el color que

queremos para el borrado de la pantalla, no tenemos porqué saber como funciona la subrutina.

Pasamos a la acción

Bien, ya estamos en disposición de usar cualquier subrutina, aún cuando no sepamos su funcionamiento, ya que nos basta con saber que parámetros (registros de entrada) necesita. ¿No nos cree? pase y vea:

```
; ------------------------------------------
; Prueba de subrutinas usando sprites
;
; Radastan
; ------------------------------------------

        ORG 32768

; ------------------------------------------
; Rutina principal
; ------------------------------------------

      ld a, 15          ; papel azul y tinta en blanco
      call sucls
      ld a, 1           ; borde azul, para actualización inmediata
      call borde

      ld d, 7           ; posición vertical del sprite
      ld e, 7           ; posición horizontal del sprite
      ld hl, cubo       ; sprite a usar
      call print
      ret

; ------------------------------------------
; SUBRUTINA PARA HACER UN CLS SIN USAR LA ROM
; ENTRADAS: a, es el color de tinta/papel
; SALIDAS: pone el mapa de pantalla a 0
; Ver tabla de color al final del programa
; ------------------------------------------

sucls ld hl, 22528
      ld de 22529
      ld bc, 767
      ld (hl),a
      ldir
      ld hl, 16384
      ld de, 16385
      ld bc, 6143
      ld (hl),l
      ldir
      ret

; ------------------------------------------
; SUBRUTINA PARA CAMBIAR EL BORDE DE PANTALLA
; ENTRADAS: a, es el color del borde
; SALIDAS: ninguna
; ------------------------------------------

borde out (254), a
      ret

; ------------------------------------------
```

```
; RUTINA DE IMPRESION DE UN SPRITE 16x16 PIXELS
; CON ATRIBUTOS EN CUALQUIER POSICION DE CARACTER
; ENTRADAS:
; D será la posición del cursor vertical en caracteres
; E será la posición del cursor horizontal en caracteres
; HL es la posición de memoria donde tenemos el sprite
; SALIDAS: se escribe en el mapa de pantalla
; ADVERTENCIAS: no comprueba límites de pantalla
; -----------------------------------------

print
      push de           ; salvamos los valores vertical y horizontal
      push de           ; salvamos los valores vertical y horizontal
      push de           ; salvamos los valores vertical y horizontal
      call cdrw         ; calculamos dirección de pantalla
      ld b, 8
      call draw
      pop de            ; recuperamos el valor horizontal
      inc d             ; incrementamos una línea
      call cdrw
      ld b, 8
      call draw
                        ; Ahora imprimimos los atributos
      pop de            ; recuperamos el valor horizontal
      call cdrw
      call catr
      call colr
      pop de            ; recuperamos el valor horizontal
      inc d             ; incrementamos una línea
      call cdrw
      call catr
      call colr

      ret

draw
      ld a,(hl)           ; hl indica la posición del sprite en memoria
      ld (de),a           ; de indica la posición de pantalla
      inc hl
      inc e
      ld a,(hl)           ; esta parte imprime el segundo byte
      ld (de),a
      inc hl
      dec e
      inc d
      djnz draw           ; decrementa B y si es cero deja de saltar a draw
      ret

colr
      ld a,(hl)
      ld (de),a
      inc e
      inc hl
      ld a,(hl)
      ld (de),a
      dec e
      inc hl
      ret

cdrw
      ld a, d           ; recuperamos el valor vertical
      and 7             ; nos quedamos con la posición en el tercio
      rrca
      rrca
```

```
      rrca              ; rotamos para dejar su valor en múltiplos de 32 (linea)
      and 224           ; borramos el resto de bits por si las moscas
      or e              ; sumamos el valor horizontal
      ld e, a           ; e preparado
      ld a, d
      and 24            ; modificamos según el tercio de pantalla
      or 64             ; nos posicionamos a partir de 16384
                        ; (16384=64+0 en dos bytes)
      ld d, a           ; d preparado
      ret

catr
      ld a,d
      rra
      rra
      rra               ; dividimos por 32
      and 3             ; nos quedamos con los tres bits bajos
      or 88             ; apuntamos al comienzo del mapa de atributos
      ld d,a            ; ya tenemos d listo, e no hay que cambiarlo
      ret

; ----------------------------------------
; sprite de un cubo
; los cuatro últimos bytes son el color
; se pueden usar sprites creados con SevenuP
; ----------------------------------------

cubo  defb 255,255,128,1,128,1,128,1,128,1,128,1,128,1,128,1
      defb 128,1,128,1,128,1,128,1,128,1,128,1,255,255
      defb 12,13,14,15

; - TINTA -
;
; NEGRO           0     00000000
; AZUL            1     00000001
; ROJO            2     00000010
; MAGENTA         3     00000011
; VERDE           4     00000100
; AZULADO         5     00000101
; AMARILLO        6     00000110
; BLANCO          7     00000111
;
; - PAPEL -
;
; NEGRO           0     00000000
; AZUL            8     00001000
; ROJO            16    00010000
; MAGENTA         24    00011000
; VERDE           32    00100000
; AZULADO         40    00101000
; AMARILLO        48    00110000
; BLANCO          56    00111000

; ----------------------------------------
; FIN DEL PROGRAMA
; ----------------------------------------
```

¿PERO TE HAS VUELTO LOCO? Para nada, si observa el programa verá que la rutina principal la entiende a la perfección, aunque no entienda nada de lo que cada subrutina está haciendo. Esa es la gracia del lenguaje ensamblador, se pueden usar subrutinas hechas por otros.
No intente comprender lo que hace la subrutina de impresión de sprites, sólo lea los

comentarios de la cabecera, como verá es muy simple:

; D será la posición del cursor vertical en caracteres
; E será la posición del cursor horizontal en caracteres
; HL es la posición de memoria donde tenemos el sprite

En este ejemplo hemos dado otra vuelta de tuerca, y el sprite también lo ponemos en el código del programa mediante "defb". Esta instrucción sirve para introducir valores de byte, tantos como queramos, y realmente tampoco es una instrucción de ensamblador, es otra instrucción usada por el programa ensamblador para facilitarnos la vida.

Si usamos el programa SevenuP y creamos un sprite de 16x16 pixels podemos obtener el código necesario para cambiar el sprite "cubo" por el que queramos. No obstante no es el momento de comprender esta parte, eso lo dejamos para más adelante.

Lo dicho, no intente comprender la subrutina de sprites, simplemente úsela.

Saltos

Vuelve a cantarla Sam

Ya sabe crear subrutinas, pero no sabe que hacer para que se repitan una y otra vez o que se ejecuten como reacción a algo. De igual forma le gustaría crear un juego que hiciera algo, no sólo mostrar cosas bonitas en pantalla y volver al BASIC... necesita crear bucles (no, no son espacio-temporales ni estamos en el día de la marmota) y también realizar acciones en base a una respuesta (es decir, un salto condicional).

Los saltos son la espina dorsal de nuestros programas, pero también nuestras peor pesadilla. Hay varias formas de realizar saltos, le vamos a ir enseñando una a una empezando por la más sencilla.

Salto incondicional

La forma más sencilla de realizar un bucle es mediante un salto incondicional, es decir mediante la instrucción "JP" (jump). Veamos un ejemplo:

```
; ------------------------------------------
; Prueba de salto incondicional
; ------------------------------------------

        ORG 32768

; ------------------------------------------
; Rutina principal
; ------------------------------------------

inic1 jp inic1

      ret
```

Si tecleamos, ensamblamos, y ejecutamos el ejemplo el ordenador se quedará visualmente "colgado" (que lo haga de una soga depende enteramente de usted). Lo que estamos haciendo es saltar con la línea "jp inic1" a sí misma, ya que usamos la etiqueta inic1 como sitio a donde saltar y la hemos situado en la misma línea (a esto le llamamos un programa egocéntrico, sólo se presta atención a si mismo y pasa olímpicamente del usuario).

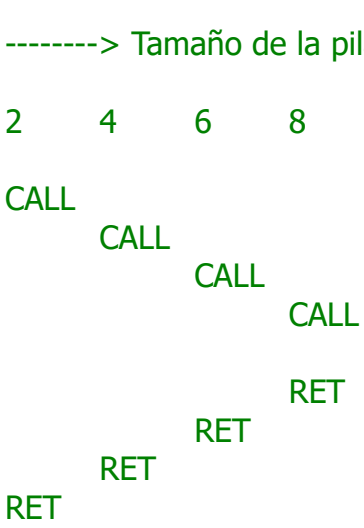
Es decir, si ponemos un salto incondicional antes del ret que apunte al inicio de la rutina principal, o a la parte que nos interesa repetir de forma indefinida, conseguiremos que nunca se retorne a BASIC. OJO, esto significa que nunca se ejecuta el RET y que si hemos realizado algún CALL y no hemos retornado la vamos a liar bien gorda... porque en cada CALL estamos usando la pila de saltos de la CPU, y si no retornamos la pila se llena indefinidamente ocupando toda la memoria y borrando el propio programa.

¿Pila? ¿no sabe lo que es la pila de la CPU? ¿no ha visto el anuncio de los conejitos de Duracell? ¿no tiene una "pila" para lavar en casa? ¿nunca ha realizado una pila de ropa? mejor le ponemos las pìlas...

La pila de la CPU

La pila de la CPU es un espacio de la memoria RAM que reserva la CPU para su uso. En este espacio se guardan las posiciones de memoria a donde se debe retornar tras cada CALL. Es decir, cada llamada a subrutina nos consume 2 bytes de memoria, y cuando retornamos con RET liberamos esos 2 bytes. Si realizamos CALL dentro de más CALL (subrutinas llamando a otras subrutinas) estaremos llenando más espacio de la pila.

Una imagen vale más que mil palabras:



Evidentemente no es exactamente así, ya que para usar nuestra rutina en ensamblador el BASIC ya ha usado algo de la pila, pero nos sirve para saber como funciona.

La pila se situa en la parte alta de la memoria, es decir parte de 65535 y crece hacia abajo. Ocupará tanta memoria como necesite, por lo que si vemos que nuestro juego empieza a hacer cosas raras o se jode por completo es que nos falta un RET en el bucle y la pila empieza a machacar toda la memoria. No olvide dejar algo de espacio para la pila.

De igual forma un RET no ejecutado significa que los retornos no se corresponden... y que nuestro juego no funcionará. Ojito.

Salto relativo

Anteriormente hemos usado JP, pero también podemos usar JR de la misma forma:

```
; ------------------------------------------
; Prueba de salto incondicional
; ------------------------------------------

        ORG 32768

; ------------------------------------------
; Rutina principal
; ------------------------------------------

inic1 jr inic1

      ret
```

Como puede observar usar JP o JR parece lo mismo, de hecho realizan el mismo efecto...

pero no son iguales. JP es un salto absoluto a cualquier posición de memoria, mientras que JR es un salto relativo a una posición cercana (127 bytes de diferencia). JR es más rápida que JP, eso es lo importante y a tener en cuenta, usaremos JR mientras podamos y JP cuando no nos quede más remedio o no tengamos problemas de rapidez.

Salto condicional

Un salto condicional significa que sólo se realiza el salto si se da una condición, y no hay mejor forma de comprenderlo que con un ejemplo práctico. Vamos a coger un fragmento de la rutina de impresión de sprites:

```
draw
      ld a,(hl)           ; hl indica la posición del sprite en memoria
      ld (de),a           ; de indica la posición de pantalla
      inc hl
      inc e
      ld a,(hl)           ; esta parte imprime el segundo byte
      ld (de),a
      inc hl
      dec e
      inc d
      djnz draw           ; decrementa B y si es cero deja de saltar a draw
      ret
```

En este fragmento hemos tomado como base que antes de esta subrutina el registro "b" vale 8. La instrucción djnz lo que hace es decrementar "b", compararlo con 0 y si dicha comparación es afirmativa NO salta a la etiqueta indicada. En inglés se leería como: d-j-nz, es decir decrementa (d) y salta (j) si no es cero (nz) a la etiqueta.

En este ejemplo, como b vale 8 antes de comenzar, se realizan 8 ciclos desde la etiqueta draw a la instrucción ret. Es decir, imprimimos los 8 bytes que componen un carácter (en este caso la subrutina imprime dos caracteres a la vez para mayor rapidez).

DJNZ es muy similar en BASIC a un bucle FOR b=x to 0... next b.

Ahora veamos un salto condicional más sencillo:

```
; ------------------------------------------
; Prueba de salto condicional sencillo
; ------------------------------------------

        ORG 32768

; ------------------------------------------
; Rutina principal
; ------------------------------------------

      ld a, 5

inic1 dec a
      jr nz, inic1

      ret
```

Esto es más sencillo de comprender. Si tecleamos y ejecutamos el programa retornaremos al BASIC sin aparentemente haber realizado nada, pero la verdad es que habremos realizado 5 bucles con la instrucción jr. Hemos decrementado el registro "a" hasta que su valor fuese 0 y jr dejara de saltar, ya que jr ha estado comparando el valor de "a" con un número distinto de cero (nz).

Un registro con bandera

Estos no son los únicos tipos de salto condicional, hay más, todos afectados por las denominadas "banderas" del acumulador (el acumulador es el registro "a"). El registro "a" es el más importante de todos los de la CPU, ya que es con el que se opera y obtiene el resultado en la mayoría de los casos (por eso se le llama también acumulador).

Por ser tan importante está asociado al registro F... si, no se ha vuelto loco, existe un registro F que es el que almacena las banderas (flags en inglés) del registro A.

Las banderas son:

- signo (negativo/positivo)
- cero (que es el usado en djnz)
- acarreo (ej. cuando una suma excede el valor del registro)
- semi acarreo (mejor no pregunteis por el momento)
- paridad (cuando realizamos comparaciones entre registros)
- adición/substracción (no pregunteis igualmente)

Se puede aprovechar de las banderas del acumulador para poder realizar saltos realmente complejos.

Comparando que es gerundio

Las comparaciones se realizan con la instrucción CP, que hace uso de los flag de adición/substracción y cero:

```
cp b              ; compara a con b
jr z, salt1       ; si a es igual que b salta a salt1
cp c              ; compara a con c
jr z, salt2       ; si a es igual que c salta a salt2
```

Para los que sepan BASIC les es conocida esta forma de programar, pueden usar CP como usan IF..THEN en BASIC. De igual forma que se compara "a" con otro registro en igualdad se puede usar en comparación de tamaño:

```
cp b              ; compara a con b
jp p, salt1       ; si a es mayor que b salta a salt1
cp c              ; compara a con c
jp m, salt2       ; si a es menor que c salta a salt2
```

Como puede ver no es necesaria una explicación más detallada, es bien sencillo.

Y hasta aquí podemos resumir por encima lo que son los saltos, porque hay para todos los gustos: JP y JR se pueden combinar con Z (cero), NZ (no cero), etc... y a su vez tenemos la eficaz DJNZ que usa el registro "b" para realizar bucles.