

TUTORIAL:
PROGRAMACIÓN DE CONSOLAS DE 8 y 16 BITS, PARA NOVATOS
3ª PARTE : CÓDIGO

Por David Senabre, Marzo 2006

OBJETIVO

Muy bien! Ya has llegado lejos. Ya tenemos una idea general del asunto, ahora introduciré algunos conceptos un poco más avanzados, pero imprescindibles. Y con este tutorial termino la saga.

Hasta el momento se ha visto una visión general y anecdótica de la programación de consolas de 8 y 16 bits, fundamentos elementales de computadores, que son imprescindibles, y algunas "técnicas básicas".

Este tutorial será el último antes de pasar a la práctica. No quiero alargar más la clase teórica, pues considero que hay que pasar rápido a la acción o se corre el riesgo de perder el interés, o tirar la toalla. He intentado por eso condensar lo necesario.

Lo que voy a tratar ahora es:

- * Saltos
- * Bucles
- * Uso de índices
- * La pila
- * Subrutinas
- * Manejo de bits
- * Interrupciones
- * Algun ejemplo más de manejo de puertos (por ejemplo para leer el pad).

UN POCO MÁS SOBRE CPUs

Ya he dicho que una CPU se basa en ejecutar instrucciones, una tras otra, y que hay un registro especial llamado PC que apunta a la dirección de la siguiente instrucción.

Pero el programa no siempre debe seguir el mismo camino. A veces en función del momento, del resultado de una operación, o una acción del usuario, tendremos que ejecutar un código u otro.

Para permitir esto hay instrucciones de salto. Se basan en desviar el flujo de ejecución del programa a otra parte del mismo. Estas instrucciones cambian el valor del registro PC para que apunte a otro sitio, y siga ejecutando el programa desde allí.

A parte de este registro, existen al menos otros dos importantes. Uno de ellos es el de la pila. Apunta a una zona de memoria RAM que hace de pila. Ahora diré para qué es.

Y el otro registro es el registro de estado. Es un registro que sólo vale para leerlo normalmente y refleja el resultado de la última operación que la CPU ha hecho. Cada bit de este registro es un "flag", o bandera. Por ejemplo, hay un fln llamado Z que se pone a uno si la última operación ha dado como resultado cero. Si has hecho una resta, si el resultado es cero, Z se pondrá a 1 y lo podrás leer. Hay otros flags importantes como el de acarreo (C), el flag de resultado negativo (N), y otros que sólo están presentes en algunas CPUs.

Ejemplos habituales serían:

<u>Operación</u>	<u>Flags</u>
3 - 2 = 1	(Z = 0)(N = 0)
2 - 2 = 0	(Z = 1)(N = 0)
2 - 5 = -3	(Z = 0)(N = 1)

En el Z80, como en casi todas las CPUs, el carry (C) se pone a 1 si el resultado es negativo. Esto es lo normal. Pero ojo, en el 6502, el carry (C) se pone a 0 si el resultado de la resta es negativo.

Otro flag importante es el de desbordamiento (overflow). Se activa siempre que la última operación ha dado como resultado un valor que se sale del registro. Por ejemplo, una suma da un valor que no cabe en 8 bits, que requiere 9, y por tanto el bit más alto se pierde.

Es importante que recuerdes que no todas las instrucciones afectan a los flags. Cuando ya entres en harina, tendrás un documento con la lista de instrucciones y los flags que modifican, no te preocupes. Por ejemplo, la instrucción **iny** del 6502 aumenta el valor del registro Y en uno. Pero si lo desborda, al sumar uno cuando ya tenía su máximo valor posible, el registro se pondrá a cero de nuevo, y NO se reflejará en el flag de overflow. **Iny** sólo afecta al flag Z y N. En cambio las instrucciones de suma y resta afectan a Z, N, C y V (como se llama al de overflow).

SALTOS CONDICIONALES E INCONDICIONALES

Cuando queremos que al llegar a un punto la ejecución del programa se bifurque a otro lugar, usaremos un **salto incondicional**.

En el 6502 es "jmp"

En el Z80 es "jp"

Tan sencillo.

Pero a veces queremos que se salte SÓLO si se ha dado un caso en concreto. Por ejemplo, imagina que quieres que se salte sólo si el contenido de la posición de memoria \$0020 es igual al de posición de memoria \$0021.

Deberás hacer una resta, y si el resultado es cero, eso es que eran iguales. Entonces deberás mirar el flag Z. Si está a 1, ya sabes que eran iguales. Para este tipo de cosas hay una instrucción máquina de comparación, pero en realidad funciona como si restaras uno del otro. Aquí usarías un **salto condicional**.

¿Te atreves con código?

En el 6502 (NES) sería:

```
Codigo1:
    lda    $0020
    cmp    $0030          ;comparar memoria con el acumulador (A)
                        ; lo que hace es restar el contenido de memoria
                        ; del acumulador (A - memoria)
    beq    Codigo2       ;salta a la etiqueta Codigo2 si Z = 1

    ; Aquí puede haber más instrucciones que se ejecutarán si
    ; Z = 0, porque no habría salto.
    ...

Codigo2:                  ;Aquí vendría el salto condicional, si Z=1
```

En el Z80 (Master System) sería:

```
Codigo1:
  ld  a,($0020)  ;cargar en registro A
  ld  b,($0030)  ;cargar en registro B
  cmp b          ;comparar registro A con B
                   ;lo que hace es restar A - B
  jp  z, Codigo2 ;salta si flan Z=1, o sea, si eran iguales
...
Codigo2:
```

BUCLES

Un bucle se basa en repetir una porción de código varias veces, haciendo lo mismo, o haciendo ligeros cambios "inteligentes".

Puedes quere repetir una cosa muchas veces, así que en vez de repetir todo el rato las mismas instrucciones haces algo como

```
Bucle_tonto:

  Instrucciones

  jmp  Buble_tonto
```

Esto sería un bucle. Pero si dentro de él no hay instrucciones de salto nunca se podría salir de ahí, porque "jmp" (o jp en Z80) es un salto incondicional. Siempre se volvería al principio de él. Se usa algunas veces, pero poco.

Lo normal es repetir algo un par de veces, por ejemplo 10. Esto es más interesante. Usamos un registro como contador para hacer esto. Mira este ejemplo, que como siempre aclara.

```
Bucle:
  ldx  #10          ;cargar 10 en X

  Instrucciones

  dex          ;restar 1 a X
  bne  Bucle      ;si el resultado no es cero, salta

  ; cuando X llegue a cero, el flag Z se pondrá a 1
  ; y por tanto la instrucción bne no saltará
```

Si te fijas, el bucle se repetirá 10 veces, y luego se seguirá.
En Z80 sería:

```
Bucle:
  ld  a,$10        ;cargar 10 en A

  Instrucciones

  dec  a          ;restar 1 a A
  jp  nz,Bucle    ;si el resultado no es cero, salta
```

Fijate bien cuando estés aprendiendo, en los bucles que se usan para cargar datos a la VRAM, por ejemplo, o para borrar la RAM al inicializar la consola (algo que está bien hacer, para asegurarse de que no hay datos residuales en ella). Para borrar la RAM se tiene que

escribir un "0" en cada posición de ella. Si la RAM es de 8Kb, habrá que escribir 8192 veces "0". Lo adecuado es usar un bucle que se repita 8192 veces, en lugar de escribir 8192 veces el mismo código, ¿no?

BUCLAS CON INDICES

Los índices son muy útiles. Serían como un valor que se suma a otro, para a memoria. Esto permite hacer un bucle en el que en cada vuelta, cambiando el valor del índice, se vaya cambiando la dirección de memoria a la que accedemos.

Por seguir con el ejemplo anterior, si quieres borrar una RAM de 8kb, no basta con escribir 8192 veces "0", sino que cada vez que lo escribas, deberás apuntar a la siguiente posición de memoria. Sino escribirías 8192 veces "0" en la misma posición, dejando las restantes 8191 como estaban :)

Imagina, por sencillez, que quieres borrar los primeros 256 bytes de la RAM. En la NES, la RAM está en las posiciones \$0000-\$0400. queremos borrar por tanto las posiciones \$0000-\$00FF (las primeras 256).

```
    ldy  #$0           ;cargar 256 en Y
    lda  #0           ;valor que vamos a escribir
Bucle:
    sta  $0000,y      ;escribir A en $0000 + Y
    iny                ; Y + 1
    bne  Bucle        ; si el resultado no es cero, salta

    ; cuando Y llegue a cero, el flag Z se pondrá a 1
    ; y por tanto la instrucción bne no saltará al bucle,
    ;y seguirá la ejecución del programa
```

La instrucción clave que debes entender es:

```
sta $0000,y
```

Lo que hace "**sta \$0000**" es almacenar el contenido del registro A en la posición de memoria de la CPU \$0000, que en la NES corresponde al inicio de la RAM.

Lo que hace "**sta \$0000,y**" es almacenarlo en \$0000 más el valor del registro Y, luego como Y empieza valiendo 0, el primer "**sta \$0000,y**" escribirá en \$0000. Después "**iny**" incrementa el valor de Y en 1. Como Y vale 1, el "**bne bucle**" no salta.

En la siguiente iteración, se escribirá en \$0001 (\$0000 + Y, que vale 1), en la siguiente en \$0002,... y así hasta que se escriba en la posición \$00FF. Entonces, el "**iny**", al sumar 1 a FF, lo que hace es volver a poner Y a 0, porque los registros de la NES son de 8 bits, y FF es el máximo valor que se puede representar con 8 bits (recuerda que cada dígito hexadecimal representa 4 bits, y que FF es 11111111). Al ser cero, el flag Z se pone a 1, el **bne** ya no salta.

Así que debes retener 2 cosas. El uso de índices, y el hecho de que si un registro tiene el máximo valor que puede almacenar (esto es, con todos sus bits a 1), al sumarle 1, vuelve a cero. Si el registro es de 16 bits, pues podría almacenar hasta \$FFFF, después, pasará a \$0000. Esto se llama **desbordamiento**, y cuando ocurre activa un "flag" en el registro de estado de la CPU, que indica que ha ocurrido un desbordamiento en la última operación. Por supuesto podrías usar ese flag para comprobar que en efecto ha ocurrido, pero en este ejemplo sencillo, basta con esperar a que Y vuelva a ser 0, ya que la única manera de que eso ocurra sumándole 1, es desbordándolo.

En la Master System yo para hacer esto no uso índices. Lo que hago es usar un registro como puntero, y lo voy incrementando, como hago con Y en este ejemplo de la NES. No es lo mejor, pero lo voy a poner para dar otro ejemplo de cómo hacer las cosas, y así aprender nuevas técnicas.

```

;Clear RAM $C000-DFFF
;-----
ld bc,$C000 ;apuntar al comienzo de RAM
-:
ld a,$0 ;vamos a escribir ceros
ld (bc),a ;escribir en la RAM
inc c ;apuntar a la siguiente direccion
jp nz,- ;repetir FF veces
inc b ;y luego incrementar el byte alto
ld a,$E0 ;hasta llegar a $E0
cp b ;eso significa $E000
jp nz,-
; en $E000 ya no tenemos que escribir, porque la RAM
; acaba e $DFFF

```

Básicamente aprovecho que el Z80 puede cargar valores de 16 bits usando 2 registros emparejados, en este caso el registro B y el C. En B se almacena el byte alto (\$C0), y en el C el bajo (\$00). Juntos almacenan el valor \$C000. Esta es la dirección de comienzo de la RAM, y usaré el par de registros como puntero. Después de cada escritura, aumento el valor de C para apuntar a la siguiente dirección, y cuando ya he barrido FF posiciones, y C vuelve a ser cero, aumento el registro B que almacena el byte alto de la dirección, y así hasta llegar al final. Si todo esto es nuevo para ti no te preocupes si no lo entiendes bien ahora mismo.

LA PILA

La pila es una zona de memoria RAM que se dedica a una función específica; hacer de pila. Y es exactamente lo que dice el nombre.

Puedes "empujar" un valor a la pila, y "sacarlo" después. Pero es similar a una pila de libros, de tal forma que lo último que has dejado en ella es lo primero que sacas de ella.

Es importante tener claro que si metes una cosa, y luego otras 3, para sacar la primera tendrás que sacar las 3 que hay encima.

En el 6502 sólo puedes "empujar" el registro A y el registro de estado (el que tiene los flags, vamos). Si quieres empujar el registro X e Y, tendrás que hacer una pirula.

```

Salvar registros en la pila en un 6502
pha ;empujar A a la pila
txa ;transferir X a A
pha ;empujar A (o sea, lo que contenia X)
tya ;transferir Y a A
pha ;empujar A (o sea, lo que contenia Y)

```

si quieres sacar el valor que tenía X al empujarlo, tendrás que sacar primero el que tenía Y, que fue el último en entrar en la pila.

```
pla      ;sacar el valor de la cima de la pila.  
pla      ;sacar el siguiente. Corresponde al que tenia X.  
pla      ;y sacar el primero que metimos. El que tenia A.
```

Sólo recuerda eso, último en entrar primero en salir. No puedes acceder a otro elemento que no sea el de la cima de la pila, el último en entrar.

En Z80 puedes empujar cualquier registro, pero en pares de registros.

```
Push  af  
Push  bc  
Push  de  
Push  hl
```

Y recuperarlos haciendo;

```
Pop   hl  
Pop   de  
Pop   bc  
Pop   af
```

Recuerda, siempre en sentido inverso. Último libro en dejar, primer libro en coger,...

Lo de la pila puede parecer de lo más estúpido para un novato. ¿Para qué una pila si tenemos una preciosa memoria RAM a la que podemos acceder como nos de la gana?

Primer motivo, por comodidad. Sí, para algunas cosas es muy comoda. Por ejemplo, si necesitas usar unos registros, pero a su vez, cuando acabe el proceso que quieres hacer, necesitas restablecer su antiguo valor, empujas los registros a la pila, y luego los recuperas en orden inverso, como en el ejemplo de arriba. Es mucho más rápido y cómodo que meterlos en algún sitio cualquiera de la RAM, y como tienes que guardarlos y sacarlos uno tras otro, no importa que sólo puedas acceder al último elemento. Mejor, de hecho (se entiende con práctica).

Segundo motivo, permite usar subrutinas, una de las cosas más potentes de ensamblador. Sin subrutinas no irás muy lejos.

Así que un respeto a la pila, que da mucho juego. La pila es Dios :-)

SUBRUTINAS

Las subrutinas son igual que cualquier trozo de código, un conjunto de instrucciones. Pero con una peculiaridad. Se llama desde varios sitios del programa, y cuando finaliza, vuelve al lugar de donde se ha llamado. Esto permite que un código que hace cosas habituales como leer el estado del mando, pueda llamarse desde distintos sitios de la ROM.

Para hacer esto con lo que sabemos hasta ahora, habría que hacer un salto, condicional o incondicional. Pero cuando acabemos la subrutina, ¿cómo sabrá la CPU donde tiene que volver? Es decir, ¿desde dónde se llamó, para poder continuar la ejecución desde allí? No sabrá.

Para eso, hay que usar una instrucción de salto nueva, el salto a subrutina. Es igual que un salto normal, pero deja en la pila la dirección desde donde se llamó a la subrutina. Cuando acabe la subrutina, deberás usar una instrucción especial de "retorno de subrutina", y entonces la CPU sacará de la pila la dirección desde donde se llamó, la cargará en su registro PC, y continuará la

ejecución desde la instrucción siguiente al salto a subrutina... ¿esta claro? Por si acaso, vamos con un ejemplo;

```

    jsr  Mi_Subrutina      ;saltamos en modo subrutina

    ; después de volver de la subrutina se seguirá
    ; desde este punto.

    Instrucciones
    Jmp Codigo

Mi_Subrutina:

    Instrucciones      ;hacemos nuestras cosas

    rts                ;y regresamos

Codigo:

    Instrucciones

```

Recuerda que si tocas la pila dentro de una subrutina debes dejarla igual que estaba cuando se empezó a ejecutar, porque sino la CPU no podrá recuperar la dirección de retorno que estaba en la cima al comienzo de la subrutina.

Si tienes que salvar registros, algo muy habitual, antes de el "rts" sacalos todos de la pila y dejala como estaba. Si empujaste 4 cosas, deberás sacar 4 cosas antes de volver.

En Z80 es igual, sólo que la llamada a subrutina es "call", y el retorno de subrutina es "ret".

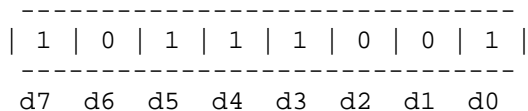
MANEJO DE BITS

Debes familiarizarte con ciertas instrucciones de manejo de bits. Muchas veces nos interesa trabajar no sobre registros enteros, sino sobre algún bit en concreto, o algunos bits. Hay dos cosas básicas que debes conocer; los desplazamientos, y las máscaras.

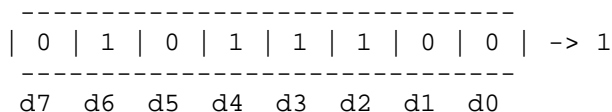
Desplazamientos

Desplazar es correr todos los bits en una dirección. Hay varios tipos de desplazamientos (principalmente rotación, desplazamiento aritmético, y desplazamiento lógico), pero tranquilo que la base es la misma. Vamos a ilustrar primero el esquema general.

Si hacemos un desplazamiento de una posición a derechas de:



El registro quedaría así:



Observa que el bit d0 se ha perdido, al salir del registro por la derecha, y que se ha introducido un cero por la izquierda, al bit d7, para llenar el hueco.

Siguiendo con el mismo ejemplo, la diferencia entre los diferentes tipos de desplazamientos a derechas radica en factores como:

- ¿Qué vamos a introducir por la izquierda para llenar el hueco?
- ¿Dónde va a parar el bit de la derecha que sale del registro?
- ¿Deben meterse por la izquierda los bits que salen por la derecha?
- ¿Deben perderse?

Como normal general, es suficiente si recuerdas:

Un **desplazamiento lógico** trata todos los bits del registro como una simple secuencia de bits, no como si fuera un número, y por tanto no tiene en cuenta su signo, y no distingue signos. Llena los huecos con ceros. El ejemplo de arriba es un ejemplo de desplazamiento lógico.

Un **desplazamiento aritmético** preserva el signo del registro, que suele estar indicado por el bit más significativo (d7 en este caso). Se usa cuando se quiere considerar los bits del registro como un número. La diferencia con el ejemplo anterior sería

El registro quedaría así:

```
-----  
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | -> 1  
-----  
d7  d6  d5  d4  d3  d2  d1  d0
```

La diferencia es que se ha retenido d7 en su posición.

Cuando es a izquierdas, suele coincidir con el desplazamiento lógico.

El principal uso de este desplazamiento, que creo importante saber, es el de multiplicar el número almacenado en el registro por 2,4,8,16, o cualquier potencia de 2. Cada posición a izquierda que desplaces el registro multiplicará su valor por 2, y cada posición a la derecha, lo dividirá por 2. Es muchísimo más rápido que hacer una multiplicación real. Así que siempre que tengas que multiplicar o dividir por 2,4,6,8,16,32,128,etc... usa desplazamientos.

Una **rotación** es distinta, se basa en meter los bits que sacamos de uno de los extremos, por el otro. Eso sí, habitualmente pasan primero por el flag de carry (que como recordarás, se almacena en el registro de estado de la CPU). Partimos del mismo ejemplo:

```
-----  
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |   | 0 |  
-----  
d7  d6  d5  d4  d3  d2  d1  d0   C (bit de carry)
```

Si rotamos una posición a izquierda, el bit d7 va al carry, y el bit que tenía el carry entra para llenar el hueco. Quedaría así:

```
-----  
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |   | 1 |  
-----  
d7  d6  d5  d4  d3  d2  d1  d0   C
```

Si rotamos otra vez, lo mismo:

```
-----  
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |   | 0 |  
-----  
d7  d6  d5  d4  d3  d2  d1  d0   C
```

La NES sólo dispone de desplazamiento lógico, y rotación. Sin embargo es un poco lioso, porque las instrucciones de desplazamiento son **ASL**

(aritmético a izquierda) y **LSR** (lógico a derechas). Pero desde mi punto de vista, como un desplazamiento aritmético a izquierda es lo mismo que uno lógico, considero que no tiene desplazamiento aritmético como tal. Las instrucciones de rotación son **ROR** y **ROL**.

La Master System es liosa, en cambio, porque dispones de todos los desplazamientos y rotaciones que he mencionado, y más. En concreto tiene 13 instrucciones de desplazamientos, que son variaciones de los explicados. Así que para que no te lies, te diré cuales se corresponden con los que hemos visto. Los desplazamientos lógicos, **SRL** y **SLL**, los desplazamientos aritméticos, **SRA** y **SLA**, y las rotaciones **RR** y **RL**. Del resto no te preocupes. Son para gente experimentada que los requiere para propósitos específicos.

Máscaras

Enmascarar es retener una parte de un registro, dejando el resto de bits que no nos interesan a cero. Es algo muy sencillo pero tan útil que he decidido mencionarlo. Un nibble son 4 bits, y cada registro de 8 bits está compuesto de 2 nibbles. El nibble bajo son los 4 bits menos significativos. Si quieres retener el nibble bajo del registro A en el 6502, haríamos

```
AND #0F
```

La máscara sería el valor `0F`. Al hacer la operación lógica **AND**, todos los bits de la máscara que son cero, fuerzan el resultado a ser cero. Luego retendremos el nibble bajo sea cual sea su valor. Si quieres retener el nibble alto, haríamos `AND #F0`.

Y si quieres algo más complicado como retener los bits 5 y 6, pues escribes en binario la máscara, con esos bits a 1, y el resto a cero. `01100000`, que en hexadecimal es `60`

A veces lo que interesa no es eliminar algunos bits, sino activarlos. Es decir, puede interesarnos poner el bit 7 de un registro a 1, por ejemplo. Eso lo haríamos con otra especie de máscara, pero usando la operación lógica **OR**.

```
ORA #80
```

`80` en binario es `10000000`. Al hacer la operación **OR**, todos los bits de la máscara que tiene valor 1, fuerzan que sea 1 el del resultado.

Ejemplos, por si acaso.

Registro A del 6502

```
-----  
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |  
-----  
d7  d6  d5  d4  d3  d2  d1  d0
```

Retener los bits 7,6 y 1.

```
AND #C1
```

Activar los bits 1 y 0.

```
OR #03
```

Desactivar el bit 7

```
AND #7F
```

En este último caso lo que hacemos es enmascarar todo menos el bit 7.

INTERRUPCIONES

Las instrucciones debes verlas como una especie de subrutinas. La única diferencia es qué motiva que se llamen.

Una subrutina es llamada por una instrucción ("jsr" en 6502, "call" en Z80, "bsr" en el motorola 68000, lo que sea...), una interrupción es llamada por un evento externo a la CPU que ocurre y que reclama su atención. Esa es la diferencia, pero es muy importante distinguirlas muy bien.

Hay que tener claro que una interrupción puede aparecer en cualquier momento. Algo externo a la CPU necesita algo, y le solicita una interrupción. Si la CPU la acepta, detiene la ejecución del programa y salta a una subrutina especial que da servicio a la interrupción, es decir, que se usa para atender la necesidad de esa interrupción. Después de ejecutarse, se vuelve de ella como de una subrutina normal (aunque con otra instrucción diferente), y se continúa desde la instrucción que se estaba ejecutando cuando se pidió la interrupción.

LA INTERRUPTIÓN MÁGICA. EL VBLANK.

En el mundo de las consolas de 8 y 16 bits hay una interrupción clave, que es la de refresco vertical. Muy resumidamente, la pantalla de un televisor dibuja una imagen 50 veces por segundo (PAL) o 60 veces por segundo (NTSC). Cuando termina de dibujar una imagen completa (cuadro), pasa un tiempo durante el cual se prepara para dibujar el siguiente cuadro. Esto se debe a que el televisor dibuja la imagen usando un haz de electrones, que al impactar sobre la pantalla crea los píxels (o puntos luminosos). El haz es como un chorro de luz muy finito, que va barriando la pantalla, en líneas horizontales, y va bajando tras alcanzar el final de cada línea, hasta llegar a la parte inferior de la pantalla. Entonces el haz tiene que volver a la parte superior, para poder empezar un nuevo cuadro.

Tal y como expliqué en "", los juegos de la época usaban el refresco de la pantalla como sincronización, para ajustar la velocidad del juego, y conseguir así que sea la misma en cualquier televisión del mismo tipo (PAL o NTSC). Por tanto, el juego procesará una serie de cosas, como la velocidad y la posición de los sprites, la vida del personaje, el scroll del nivel, los ítems, trayectorias y colisiones, por ejemplo, y esperará a que el televisor termine de dibujar el cuadro actual, para actualizar la posición de los objetos en la pantalla.

Por tanto, se actualiza la imagen mientras la pantalla se prepara para dibujar el siguiente cuadro, y no está dibujando nada. Hay un motivo importante por el cuál hacer esto en ese momento, y no hacerlo cuando se está a mitad de dibujar un cuadro. Si lo intentas, seguramente verás que la imagen se vuelve corrupta, que aparecen errores gráficos. Principalmente debido a que el chip de vídeo está más ocupado cuando el televisor está dibujando la imagen, que cuando está en el periodo en el que no está dibujando nada. Por eso, cuando accedamos a la VRAM, sobretodo si es para transferir bastante información, hay que asegurarse de hacerlo durante el periodo de Vertical Blank (VBlank), que es como se llama al momento en el que el haz del televisor está posicionándose para empezar un nuevo cuadro.

Todo lo que hagas durante el VBlank no provocará fallos gráficos en la imagen.

La cuestión es... ¿cómo diablos sé cuando la televisión está en VBlank? En PAL, esto ocurre 50 veces por segundo, y dura muy poco tiempo, así que por casualidad no daremos con él. Y de nuevo con un ejemplo práctico se ve claramente la importancia de las interrupciones.

El chip de vídeo generará una interrupción al comienzo del Vblank, que puedes usar en tu programa para actualizar el contenido de la VRAM, y como sincronización para que el juego funcione siempre a la misma velocidad (ya que aunque acabe los cálculos muy rápido, como llegado a algún punto, tiene que esperar al VBlank, y sólo seguirá la ejecución después de que ocurra, la velocidad permanecerá fija).

Esta interrupción la puedes desactivar si quieres, diciendo al chip de video que no la genere. También puedes evitarla diciendo a la CPU que no responda a interrupciones, cosa que en la NES, por ejemplo, no puedes hacer, porque esta interrupción es de tipo NMI (non-maskeable interrupt) y no se puede deshabilitar en la CPU. Si le llega, tiene que responderla. Si activas la interrupción Vblank de la PPU, la CPU no puede no responder a ella, por lo que en la NES se activa y de desactiva en la PPU únicamente (usando el registro \$2000).

En la NES, activa la interrupción Vblank activando el bit más significativo (bit 7) del registro \$2000, y deja al programa en un bucle infinito como por ejemplo:

```
Instrucciones
lda #%10000000
sta $2000
Bucle:
    Jmp Bucle
```

Y ahora escribe una subrutina que hará de rutina de interrupción.

```
Nmi:
    Salvar todos los registros
    Instrucciones
    recuperar todos los registros

    Rti                ;volver de interrupcion
```

En vez de rts, se usa rti (retorno de rutina de interrupción). A este tipo de rutinas se les llama de servicio de interrupción, pero bueno, es sólo cuestión de nombres.

Es importantísimo que salves todos los registros en la pila, y luego los saques en orden inverso, porque una interrupción puede ocurrir en cualquier momento, cuando el programa esté procesando algo con algún registro. Por tanto, cuando se llegue a rti y se vuelva, el valor de los registros debe ser el mismo que cuando se activó la interrupción, so pena de una catástrofe :-)

Pero ¿cómo sabe la consola que subrutina debe llamar en caso de interrupción? En cada consola esto se define de una manera distinta. Además, hay diferentes tipos de interrupciones. Aunque en la NES las cosas son muy sencillas, porque con usar la VBlank es suficiente. La dirección de la subrutina que se debe llamar en caso de una NMI se le pone junto a otras 2 direcciones muy importantes, al final del mapa de memoria del 6502 (en este caso, al final del banco de ROM mapeado en \$C000-\$FFFF). Otra de las direcciones le dice al 6502 dónde empezar a

ejecutar código después de un reset, o al encenderse, y es por tanto muy importante también.

En la Master System hay un par de tipos de interrupción, aunque la VBlank y la del botón de pausa son las más habituales. Todas se pueden desactivar en la CPU. O sea que aunque le digas al VDP que genere interrupción, si en el Z80 deshabilitas las interrupciones, éste nunca las atenderá.

UN REGALITO. VAMOS A LEER EL PAD

Para acabar comentaré como leer el estado del pad en una Nes y una Master System, por si no te ha quedado claro el uso de los puertos. Cargar el registro A con información del pad, en la NES haríamos:

```
lda $4016
```

En la Master System:

```
in a,($DC)
```

El byte que hemos leído, y almacenado en el registro A, contiene el estado de los botones del pad. El formato de esta información, es decir, como interpretarla, varía de una consola a otra.

En la Master system, cada bit corresponde a un botón, incluido los 4 de dirección (arriba, abajo, izquierda y derecha). Por lo que el byte contiene el estado de los 6 botones (y deja 2 bits sin usar).

En el caso de la NES, el byte leído tiene información sólo de un botón, y su estado se refleja en el bit 0. Para leer el estado del siguiente botón, tendrás que volver a cargar el registro a con el puerto \$4016, y mirar de nuevo el bit 0. Y así hasta un total de 8 veces, que son los botones que tiene la NES (arriba, abajo, izquierda, derecha, A, B, start y select).

En la NES, para indicar al PAD que vamos a leer el estado de los botones, hay que escribir un 1 y luego un 0 en el puerto \$4016. Lo haremos antes de leer el primer boton, y cada vez que queramos empezar a leer desde el primer botón.

Todo esto se puede hacer de varias formas, pero yo voy a decir como hacerlo bien, utilizando los recursos que acabamos de aprender en este tutorial, es decir, usando bucles e indices.

El código que yo uso es:

```
;dar un pulso en el bit 0 de $4016
;eso indica al pad que vamos a leer los botones

ldx #1
stx $4016      ;escribimos un 1
dex           ;decrementamos x
stx $4016      ;escribimos un 0
ldy #0        ;indice = 0, para el bucle de ahora
;ahora cargamos los datos del pad a traves del mismo registro
;y almacenamos en una zona de RAM reservada a ello

LeerPad:
lda $4016      ;leer dato
and #1        ;importante, nos interesa solo el bit 0
sta A.BOTON,y ;almacenar en RAM + Y
iny           ;incrementar el indice, para ir barriendo las
              ;posiciones de memoria contiguas.

cpy #8        ;hemos copiado ya el indice 8?
bne LeerPad   ;si no es asi, seguir copiando

rts
```

A.BOTON es una zona de memoria de la NES. En la ROM, más arriba, habremos tenido que definirla, y yo lo que hago es definir 8 bytes consecutivos en la RAM, para almacenar el estado de los botones según llegan. Por ejemplo, las posiciones \$0008-\$000F de este modo:

```
A.BOTON   = $08
B.BOTON   = $09
SEL.BOTON = $0A
STA.BOTON = $0B
UP.BOTON  = $0C
DOWN.BOTON = $0D
LEFT.BOTON = $0E
RIGHT.BOTON = $0F
```

Como ves, usando el registro Y como índice, al ir sumándole 1 en cada paso por el bucle, se va accediendo cada vez a la posición de memoria siguiente.

Revisa mi demo si tienes dudas.

En la Master System basta con leer una vez el puerto del pad, ya que el estado de cada botón viene reflejado en uno de sus bits. Por ejemplo, el bit menos significativo representa la dirección arriba, el siguiente abajo, el siguiente izquierda, el siguiente derecha, el siguiente boton 1, y finalmente el bit 5 refleja el boton 2.

Es decir el byte leído tiene el formato siguiente:

xx21RLDU

Donde:

x - no interesa ahora.

Nº- nº del botón

Letra - U(up,arriba),D(down,abajo),L(left,izquierda),R(right,derecha)

El código que yo uso lo he escrito menos eficientemente que el de la NES, para que sea más claro y fácil de entender. En este código voy a hacer uso de rotaciones, que es algo muy útil que deberás de manejar bien. Lo de porqué complemento el byte leído lo explico bajo.

```
in a,($DC)      ;leer el estado del pad
cpl             ;complementar sus bits
ld b,a         ;y guardarlo en B
ld a,0         ;A sera un registro auxiliar

rr b           ;rotar B a traves del flag carry
rl a           ;e introducir por la izquierda en A
ld (Up),a      ;ahi tenemos el estado "Arriba"
ld a,0         ;reseteamos A
               ;para recibir el siguiente boton
rr b           ;volvemos a rotar al carry
rl a           ;y a introducir por la izquierda en A
ld (Down),a    ;y asi sucesivamente...
ld a,0

...            ;sigue para el resto de botones
```

En los documentos de cada consola encontrarás qué bit refleja el estado de qué botón. Te será de ayuda recordar que como normal general, un 0 indica botón pulsado, y un 1 botón no pulsado. Por este motivo complemento el byte leído del estado del pad. Así un bit a 1 indica botón pulsado, y viceversa, que es algo más natural para nosotros. Es sólo una sugerencia.

CONCLUSIÓN

(Vuelvo a repetir el cuento de siempre) El 6502 de la NES me parece más fácil de programar, y más adecuado para empezar. Tiene menos registros, y un direccionamiento indirecto más potente. Los puertos en la NES se acceden como si fueran direcciones de memoria, que al principio puede resultar más intuitivo. Aunque es cuestión de opiniones.

El Z80 de la Master System en cambio tiene sus ventajas. Es más potente en muchos aspectos, dispone de más registros, y permite usarlos por pares para trabajar con valores de 16 bits más cómodamente. El mapa de memoria de la Master System es todavía más sencillo que el de la NES, y los puertos no están dentro del mapa de memoria, sino que se accede a ellos a través de instrucciones especiales (**out** e **in**).