Ni intento
ENTERTAINMENT KERNEL

Martín Villagra

17 de diciembre de 2014

Resumen

El objetivo del trabajo es construir un emulador de la videoconsola Nintendo Entertaiment System(también conocida como Family) que se ejecute sin ningún sistema operativo de por medio.

Índice general

Capítulo 1

Introducción

La consola NES[1]fue una de las consola más vendidas de todos los tiempos. En 2009, la misma fue nombrada la mejor consola de videojuegos de la historia por IGN[2]. Gracias a su popularidad y simplicidad muchas personas se interesaron en el funcionamiento de la consola, dando lugar a una abundante documentación al respecto.
Fue por estas dos razones que se dicidió emular está consola en particular.
Por otro lado, se eligió realizar un kernel para no depender de ningún sistema operativo. Esto introduce la posibilidad de ejecutar el emulador en máquinas más restringidas, que obviamente resultarían de menor costo que una computadora o la consola original.
Debido a la gran extensión del trabajo, se decidió seguir la filosofía de 'no reinventar la rueda'. El proyecto se construyó tomando como base otros kernels y emuladoras ya existentes, mencionados en la bibliografía (Capítulo **??**).

1.1. Estructuración

En un principio dividimos en dos partes principales el projecto.
Una es el **emulador** propiamente dicho, es decir la que se encarga de hacer todo lo que hacía internamente la consola.
La otra parte es el **kernel** encargada de inicializar todo lo necesario para que el emulador funcione, así como proveerle funciones de bajo nivel tales como escribir en pantalla o reservar memoria. Al no tener un sistema operativo detrás, funciones como malloc y free que cualquier programador de C supone siempre presentes deben ser implementadas por el kernel.

[1] `http://en.wikipedia.org/wiki/Nintendo_Entertainment_System`
[2] `http://uk.ign.com/top-25-consoles/1.html`

Capítulo 2

El Kernel

El diseño del kernel está muy lejos de lo que hoy en día se conoce como un sistema operativo completo. Se priorizó la simplicidad recortando todo lo que estaba de más, reduciendo al mínimo las capacidades del sistema.

2.1. Carácteristicas

- Dentro de la familia de arquitecturas x86 se eligió i686. Se prefirió esta por sobre x86_64 ya que tiene esta última no es soportada nativamente por GRUB(ver sección 2.3). Respecto al formato de los ejecutables se eligió el que implementan los sistemas Unix(como Linux), el ELF [1].
- Es un kernel monolítico, esto significa que todos los drivers y los servicios necesarios para el funcionamiento completo del sistema están incluídos en dentro del mismo kernel. Sistemas operativos como Linux no siguen esta filosofía, ya que implicaría cargar todos los drivers al incio del sistema, el cual tiene un costo de memoria muy alto. Esto no representa un problema para nosotros puesto que la cantidad de drivers y servicios ofrecidos por el kernel es muy reducida.
- Monotarea: Solo puede realizar una tarea a la vez, en nuestro caso el emulador. Para simplificar aún más incluso, el emulador es verdaderamente la *única* tarea que el kernel puede ejecutar. Podría verse al emulador como un solo programa standalone, que no necesita ningún sistema operativo para ejecutarse.
- Sin memoria virtual: Optamos por no incluir esta opción, para simplificar el sistema. Esto significa que los punteros contienen la dirección de memoria física y que es posible acceder y leer la memoria usada por otros procesos libremente.

[1] `http://en.wikipedia.org/wiki/Executable_and_Linkable_Format`

- Sin userspace: Normalemente un sistema operativo puede dividirse en tres capas de abstracción diferentes, el hardware, el kernel y finalmente el userspace, que es donde se ejecutan las aplicaciones como Chromium, Notepad, *etc*.

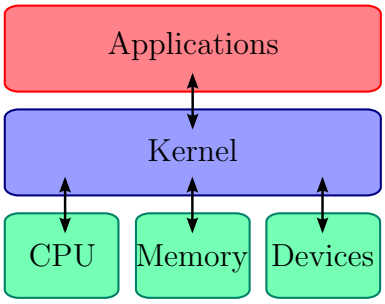


Figura 2.1: Layout estándar de un kernel.

En nuestro caso no tenemos userspace: el emulador, nuestro único programa a ejecutar, está embebido en el mismo kernel. Esto simplifica enormemente el diseño del kernel.

2.2. Funciones

Antes de hacer el kernel, hay que tener muy en claro que es exactamente lo que tiene que hacer. Surge la pregunta entonces ¿Qué necesita como mínimo nuestro emulador? Se muestra, en orden de importancia:

- Iniciar el sistema y ejecutar el emulador. Ver sección 2.3.
- Modificar los pixeles de la pantalla. Ver sección 2.4.
- Detectar pulsaciones del teclado. Ver sección 2.5.
- Poder reservar memoria dinámica. Ver sección 2.6.
- Cargar de alguna forma los juegos. Ver sección 2.7.

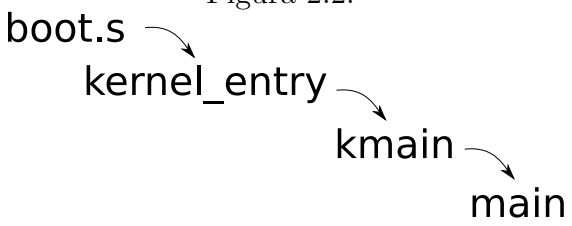
Pues bien, el kernel tiene que ser capaz de proveer funciones que faciliten cada una de estas tareas. En las siguientes secciones se detallaran cada una de ellas.

2.3. Booteo

Para evitar tener que lidiar con la BIOS y otras interfaces de bajo nivel, se eligió utilizar un bootloader ya existente y ampliamente usado: GRUB.

Existe un standard llamado Multiboot[2] que especifica como estructurar un kernel para que el mismo pueda ser cargado por GRUB(o por cualquier otro bootloader que implemente Multiboot).
En particular se debe elaborar un header al inicio del kernel. En esta cabecera se escribe por ejemplo que modo se prefiere (texto o consola) y valores para verificar integridad.
A su vez GRUB se comunica con la BIOS y recolecta información de la máquina que luego es recibida convenientemente por nuestro kernel.

Figura 2.2:



De esta forma al encender la máquina se iniciará GRUB, el mismo va a poder detectar y ejecutar el kernel. Ya iniciado por GRUB, la secuencia de arranque se puede ver en la figura 2.2

2.3.1. boot.s[3]

El problema es que al momento en que GRUB ejecuta el kernel el stack pointer no está inicializado por lo que no es posible que la función inicial sea en C. Debemos comenzar en assembler. Esta es la razón de la existencia del archivo boot.s, que contiene el comienzo de nuestro kernel. Allí se incluye el header de Multiboot, el espacio para el stack y una función, la primera en ser llamada al iniciar el sistema, que inicializa el stack pointer y llama a nuestra función kernel_entry (ya en C) que continua con la inicialización del sistema.

2.3.2. kernel_entry[4]

En está función se inicializan en el orden dado las siguientes funciones:

- Pantalla y consola: Se hace primero para poder ver cualquier error en los siguientes pasos.
- Tablas globales de descripción (GDT): Indican al procesador como reaccionar antes las distintas interrupciones (a que dirección saltar).
- Programmable Interrupt Counter (PIT): Configura el procesador para que llame a determinada función cada cierta frecuencia fija.
- Page Memory Management (PMM): Ver sección 2.6.
- Teclado: Ver sección 2.5.

[2] `http://en.wikipedia.org/wiki/Multiboot_Specification`
[3] Ubicado en kernel/arch/x86/generic/init/boot.s
[4] Ubicado en kernel/arch/x86/generic/init/kernel_entry.c

2.3.3. kmain[5]

En esta función inicia el sistema virtual de archivos (VFS, ver sección 2.7) y una pequeña librería de timers que no fue usada en el proyecto.

2.4. Pantalla

Se aprovechó la información brindada por GRUB. La misma nos provee un puntero a un buffer lineal de la pantalla, así como sus dimensiones y profundidad (bits por pixel). En la inicialización se calcula cuantos carácteres entran por línea y se configura la manera adecuada para escribir un pixel (dependiendo de la profundidad), entre otras cosas. Teniendo esto se implementaron diferentes funciones para modificar la pantalla (escribir una imagen, un carácter, hacer un clear, *etc*)[6].
El emulador obtiene el puntero al buffer lineal y va modificandolo con ayuda de la función setpixel, configurada previamente.
Respecto a imprimir texto, las funciones involucradas en el proceso se pueden ver en los siguientes diagramas:

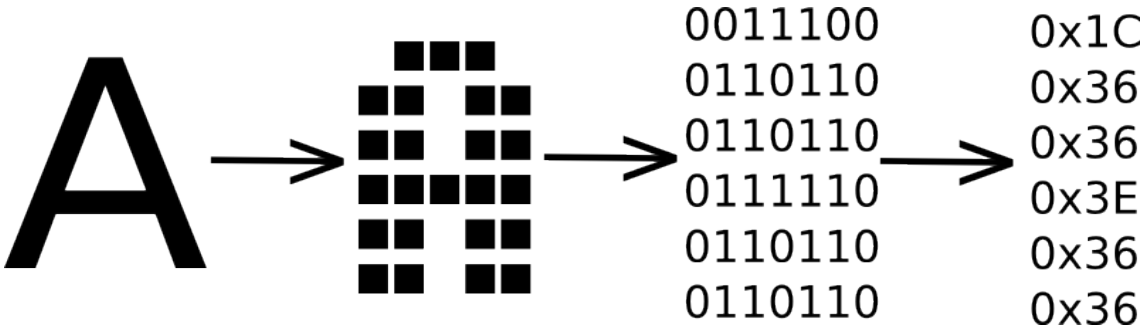
printf parsea la entrada, y por cada carácter a escribir llama a putchar. putchar calcula en que posición del buffer tiene que ir el carácter y llama a textmode_write, que guarda finalmente el carácter con la información del color en el buffer.

Este carácter no se muestra hasta que se llama a video_flush_console, el cual vuelca el buffer de la consola a pantalla, carácter por carácter.

2.4.1. Fuente

Para escribir un carácter en pantalla, se guardó para cada uno de los posibles carácteres la información para dibujarlo.

[5]Ubicado en kernel/kmain.c
[6]Implementadas en kernel/graphics/video.c



Este fue el procedimiento para generar la fuente de 16x16 de la consola, el cual nos evita tener que realizar a mano cada carácter.

1. Se descargó de internet la fuente deseada, inspirada en la que usaban los juegos de la consola (formato ttf)
2. Se creó una imagen conteniendo todos los carácteres en una cuadricula (formato xfc)
3. Se exportó la imagen a formado RAW (extensión .data)
4. Se hizo makearray.cpp[7], que carga la imagen y para cada fila calcula una máscara de bits de sus pixeles y las muestra como un arreglo de C.
5. Se agregó el mismo al codigo fuente del kernel[8].

2.5. Teclado

Para detectar cuando se presiona una tecla, se configuró la interrupción IRQ1 para que llame a la función kbd_handler[9]. En la misma se lee del puerto 0x60 para obtener el scancode que contiene información sobre si se soltó o presionó una tecla y cual fue. La función getchar que proporciona el kernel se implementó usando una arreglo circular como buffer. Sin embargo esta función no es utilizada por el emulador ya que sólo necesita saber el estado de los controles en un momento dado.

2.6. Administración de Memoria

En primer lugar, se dividió la memoria disponible en páginas de 4KB cada una. Se decidió utilizar la librería liballoc, cuyo funcionamiento se explica más

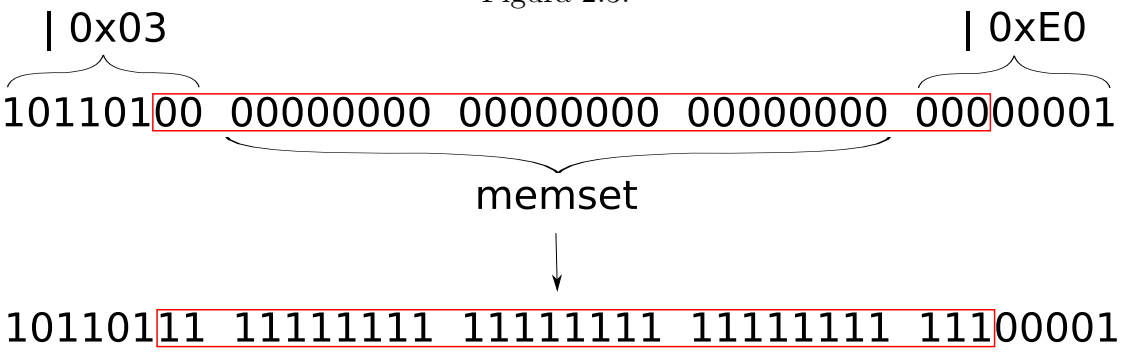
[7] en utils/fuentes
[8] graphics/rsrc/nesfont.h, usando en graphics/video.c
[9] kernel/arch/x86/generic/drivers/keyboard.c

adelante. La librería requiere que implementemos dos funciones: liballoc_alloc y liballoc_free[10].
liballoc_alloc recibe la cantidad de páginas contiguas que se necesitan reservar y debe devolver un puntero a la primer página.
liballoc_free es para liberar las páginas que reservamos anteriormente, recibe el puntero que habíamos devuelto y de alguna forma deberíamos marcar esas páginas como libres para ser reutilizadas.
Por cada página se guarda un bit, para saber si esa página está reservada (1) o libre (0). Como en cada entero podemos guardar 32 bits, podemos empaquetar todos los bits en $\lceil \frac{x}{32} \rceil = \lfloor \frac{x+31}{32} \rfloor$ enteros, donde x es la cantidad de páginas.
Utilizando máscaras de bits podemos realizar las operaciones necesarias más rápidamente. En particular se necesitaron tres funciones auxiliares[11].

- bitmap_set(l, r): Dado un intervalo en el bitset, setear todos los bits que estan dentro de él. Es usada para marcar las páginas como ocupadas.
- bitmap_clear(l, r): Análogo a la anterior, pero para limpiar los bits.
- fits_inside(x, v): Busca dentro de un entero v (32 bits) si hay x bits libres contiguos.

Por ejemplo, para implementar bitmap_set, primero se activaron los bits de los extremos usando máscaras de bits y luego los restantes mediante una lamada a memset. Se muestra un ejemplo en la figura 2.3 (usando 8 bits en lugar de 32).

Figura 2.3:



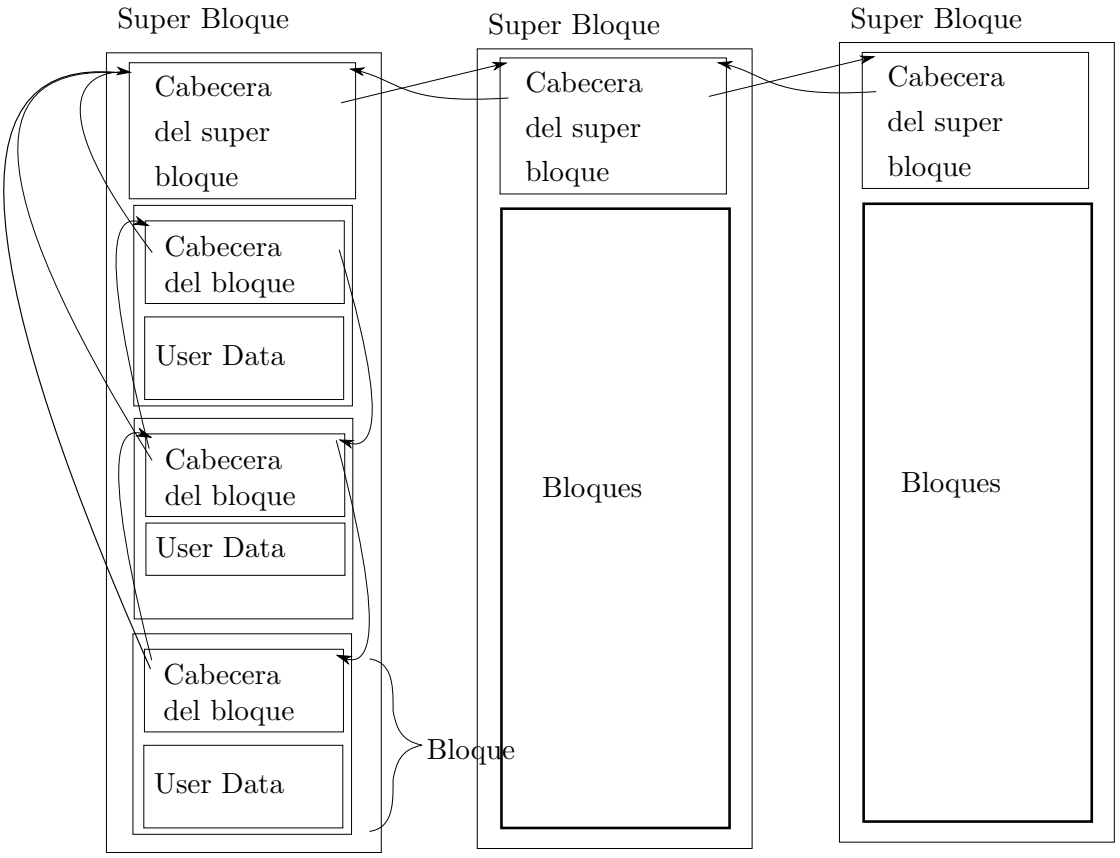
2.6.1. liballoc

Internamente para cada conjunto de páginas reservadas se crea un 'super bloque'. Cuando se llama a malloc se busca un super bloque con la suficiente memoria libre y se crea un bloque dentro de él, que va a contener la memoria que se va a otorgar al llamante.

[10] implementadas en kernel/arch/x86/pmm.c
[11] implementadas en kernel/arch/x86/pmm.c

Un dato interesante es que como estamos implementando malloc no podríamos pedir más memoria dinámica para guardar información sobre los bloques. La solución a esto fue guardar esta información directamente al comienzo de cada bloque. Para recorrer estas estructuras se usan listas enlazadas, los punteros al siguiente/anterior tambień se encuentran en las cabeceras de cada bloque, como se puede apreciar en la figura 2.4.

Figura 2.4:



Otra ventaja de este sistema es que cuando se recibe un puntero a la User Data, se puede acceder a su cabecera y obtener la información del bloque al que pertenece.

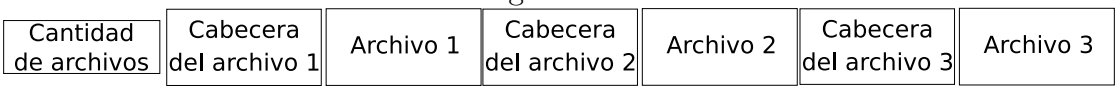
2.7. Sistema virtual de archivos

Necesitamos poder cargar los juegos desde algún lugar para poder emularlos. Como leer de discos puede ser relativamente complejo, se optó por un initrd[12]. Esto es básicamente un archivo que se carga directamente en la RAM al inicio del sistema. Afortunadamente la carga de este archivo es realizada enteramente

[12] `http://en.wikipedia.org/wiki/Initrd`

por GRUB antes de bootear el kernel. GRUB proporciona un puntero al inicio del archivo en RAM y el kernel se encarga de procesarlo.

Figura 2.5:



El formato del initrd que se utilizó se ejemplifica en la figura 2.5. El mismo consta simplemente de los archivos colocados uno tras otro,junto con una cabecera que contiene el tamaño, nombre y un valor para verificar integridad. Para crear este initrd se programó initrd_gen[13], el cual recibe los archivos y crea el la imagen lista para ser cargada por GRUB. En el archivo initrd.c[14] se implementó la inicialización del initrd, junto con sus procedimientos para leer y listar los archivos.

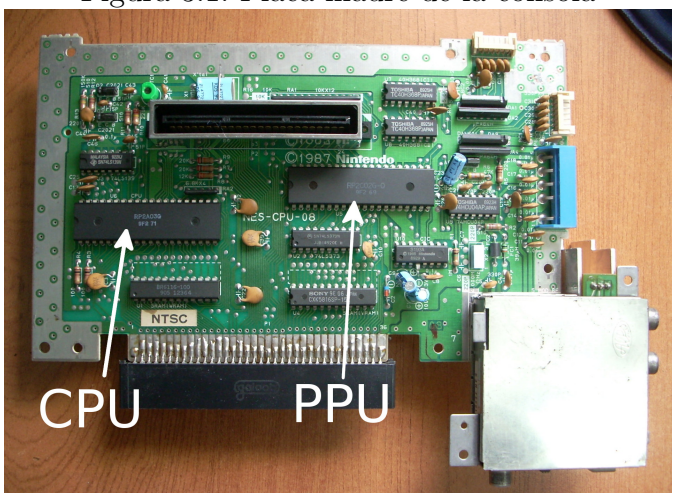
[13] utils/initrd_gen.c
[14] kernel/fs/initrd.c

Capítulo 3

Emulador

La intención de
que la consola sea más barata
que la competencia de la
época llevaron a que Nintento
se decida a usar una Unidad
de Procesamiento Central
(CPU) desactualizada.
Si bien un procesador de
16-bit hubiera sido lo óptimo,
para mantener los precios
bajos se decidieron a usar una
variante del procesador 6502
de 8 bits, desarrolado por
MOS techonology en 1975.
El chip bastaría para ejecutar
los programas, pero sería
incapaz de generar los gráficos
requeridos por lo que la compañía dicidió usar un segundo chip como una unidad totalmente dedicada al procesamiento de imagenes (Picture Processing Unit, PPU), responsable de calcular y mostrar los gráficos.

Figura 3.1: Placa madre de la consola

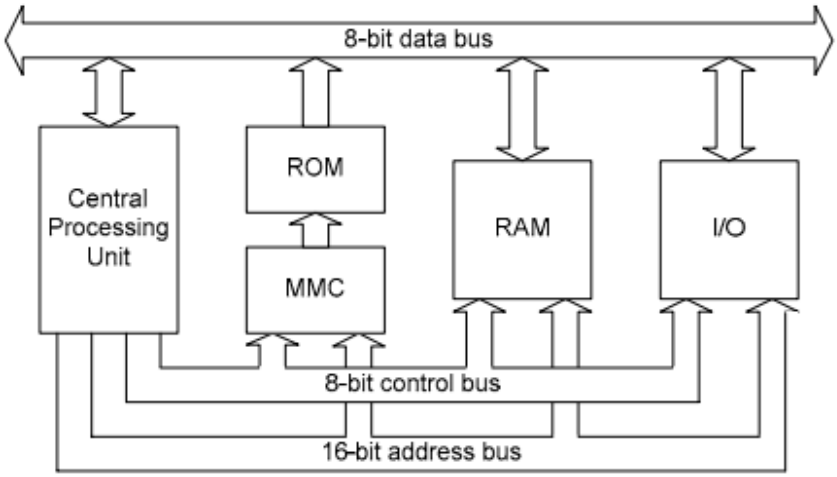


Ambos chips tienen su propia memoria interna, en forma de RAM. Los juegos son usualmente guardados en chips ROM[1] dentro del cartucho, el cual es accedido por la CPU cuando los mismos son insertados en el sistema. La NES usa memory mapped I/O[2] para permitir al procesador comunicarse con sus otros componentes: la PPU y los dispositivos de entrada. La memory mapped I/O es una técnica donde la información puede ser transferida a un dispositivo mediante la escritura en una dirección espécifica en la memoria.

[1] Read Only Memory
[2] Input/Output

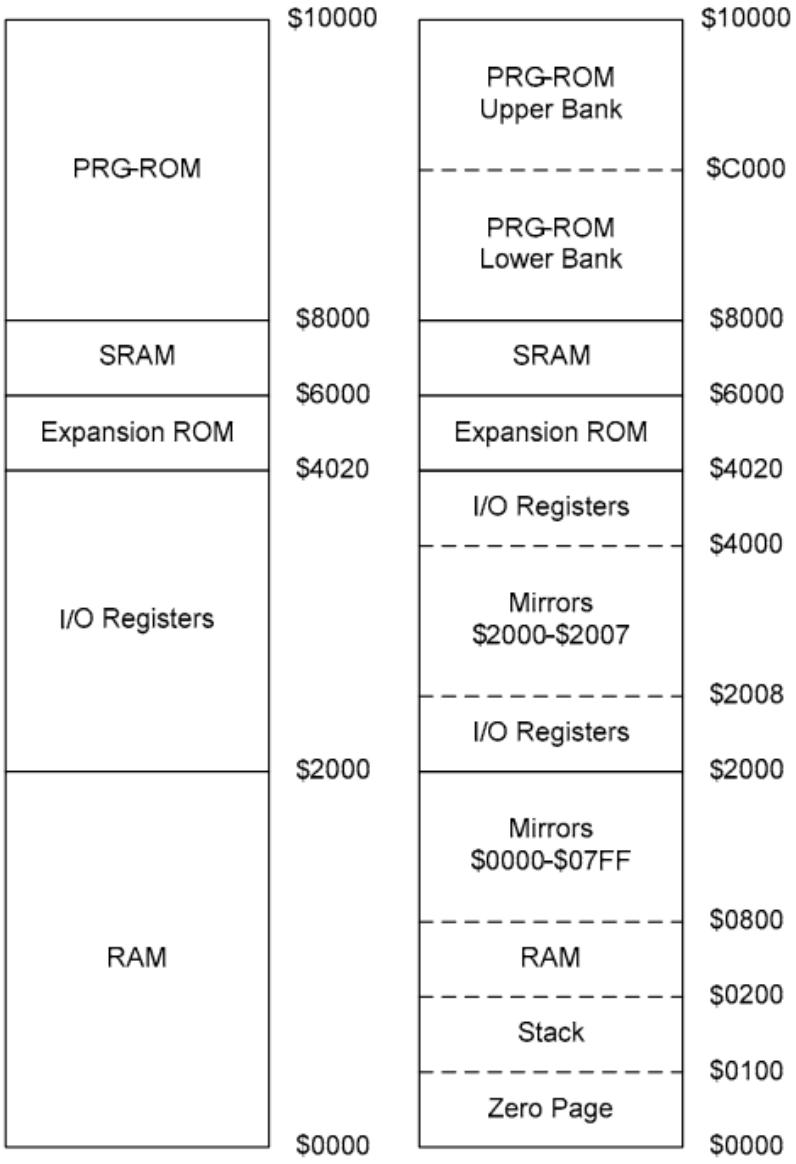
3.1. CPU Memory Map

Figura 3.2: Buses de la consola



La memoria está dividida en tres partes, la ROM dentro de los cartuchos, la RAM de la CPU y los registros I/O. El bus de direcciones es usado para establecer la dirección deseada. El bus de control es usado para informar a los componentes si el pedido es de lectura o escritura. El bus de datos es usado para leer o escribir el byte a la dirección elegida. Notemos que la ROM es de solo lectura y es accedida via el MMC, para permitir el cambio de bancos (explicado más abajo). Los registros I/O son usados para comunicarse con los otros componentes del sistema: la PPU y los dispositivos de entrada.
El chip 2A03 presente en la consola tiene un bus de direcciones de 16-bit y como tal puede soportar 64 KB de memoria en el rango $0000-$FFFF. La figura 3.3 muestra el mapeo de memoria usado por la NES. El lado izquierdo es una versión simplificada mostrando las secciones más importantes, mientras que el lado derecho detalla cada sección.

Figura 3.3: Mapeo de la memoria



Zero Page se refiere a las direcciones en el rango $0000-$00FF, la primera página en memoria. La misma es usada por ciertos modos de acceso para lograr una ejecución más veloz. Las direcciones $0000-$07FF están espejadas tres veces en $0800-$1FFF. Esto significa que por ejemplo, cualquier información escrita en $0000 también se escribirá en $0800, $1000 y $1800. Las direcciones $2000-$2007 están espejadas cada 8 bytes en la región $2008-$3FFF y los restantes registros siguen este espejado. SRAM (o WRAM) es la RAM de guardado, espacio usado para guardar las partidas.

A partir de $8000 se encuentra la PRG-ROM[3] del cartucho. Juegos con solo un banco de 16 KB de PRG-ROM se cargaran tanto en $8000 como en $C000. Juegos con dos bancos de 16 KB de PRG-ROM, cargaran uno en $8000 y otro en $C000. Juegos con más de dos bancos usan mamory mappers para determinar que banco cargar en memoria. El memory mapper vigila las escrituras a una dirección específica (o rango de direcciones) y cuando esa dirección es escrita, realiza un cambio de bancos[4]. Los detalles varían entre diferentes memory mappers.

3.2. Registros

El 6502 tiene menos registros que procesadores similares. Existen tres registros especiales: el program counter, el stack pointer y el status register. También tiene tres registros generales: el acumulador y los registros X e Y, los cuales pueden ser usados para guardar o controlar información temporalmente.

3.2.1. Program Counter (PC)

- Program Counter (PC): Es un registro de 16 bits que guarda la dirección de la siguiente instrucción a ejecutar. El valor puede ser afectado por instrucciones de branch o jump, por llamadas e interrupciones.
- Stack Pointer (SP): La pila esta ubicada en las direcciones $0100-$01FF. El stack pointer es un registro de 8 bits que sirve como un offset del $0100. El stack crece de arriba hacia abajo, por lo que cuando un bits es insertado en el stack, el stack pointer es decrementado. No hay detección de stack overflow y el stack pointer pasara de $00 a $FF.
- Acumulador (A): Es un registro de 8 bits que guarda los resultados de las operaciones aritméticas y lógicas. El acumulador también puede ser asignado a un valor cargado de memoria.
- Registro X: El registro X es un registro de 8 bits tipicamente usado como contador o offset para algunos modos de acceso. El registro X puede ser asignado a un valor en memoria y puede ser usado para obtener o establecer el valor del stack pointer.
- Registro Y: Es un registro de 8 bits, usado de la misma manera que el X, sólo que Y no puede afectar al stack pointer.
- Processor Status (P): El registro de status contiene distintas flags de un bit cada una.

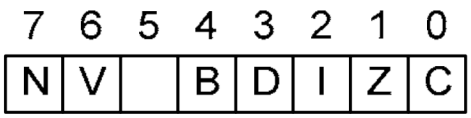
[3]Program ROM

[4]Hace que la memoria mapee hacia un banco distinto al que se estaba usando

- Carry Flag (C): Se activa cuando la última instrucción resulta en un overflow del bit 7 o un underflow del bit 0.
- Zero Flag (Z): Se activa si la última instrucción dio como resultado cero.
- Interrupt Disable (I): Puede ser usado para evitar que el sistema responda a los IRQs. Es activado por la instrucción SEI (Set Interrupt Disable), las IRQs serán ignoradas hasta que se ejecute CLI (Clear Interrupt Disable).
- Decimal Mode (D): Como el chip 2A03 no soporta el modo BCD, este flag es ignorado.
- Break Command (B): Es usado para indicar que una instrucción BRK ha sido ejecutada, causando una IRQ.
- Overflow Flag (V): Es activada si se obtuvo un resultado inválido en la instrucción anterior, tomando el complemento a dos del mismo.
- Negative Flag (N): El bit 7 de un byte representa el signo de ese byte. Esta flag se activa si el bit de signo es 1.

Estos flags están dispuestos en el status register como muestra la figura 3.4

Figura 3.4: Status Register



3.3. Interrupciones

Las interrupciones pueden ser generadas por software y por hardware. La NES tiene tres tipos diferentes que son NMI, IRQ y reset. Las direcciones a la cual saltar cuando una interrupción ocurre son guardadas en un arreglo (o vector table) en el PRG-ROM en $FFFA-$FFFF. Cuando una interrupción ocurre el sistema performa las siguientes acciones:

1. Reconocer el pedido de interrupción que ocurrió.
2. Completar la ejecución de la instrucción actual.
3. Poner el program counter y el status register en el stack
4. Activar el flag interrupt disable para prevenir interrupciones más adelante.
5. Cargar la dirección de la rutina encargada de procesar la interrupción desde la vector table al program counter.

6. Ejecutar la rutina correspondiente.

7. Luego de ejecutar la instrucción RTI (Return From Interrupt), obtener el program counter y el status desde el stack.

8. Continuar la ejecución del programa.

Las **IRQs**, también llamadas maskable interrupt, son generadas por algunos memory mappers. Pueden ser iniciadas por software usando la instruccion BRK (Break). Cuando una IRQ ocurre el sistema salta a la dirección $FFFE-$FFFF.
La **NMI** (Non-Maskable Interrupt) es una interrupción generada por la PPU cuando la V-Blank ocurre al final de cada frame. NMI no es afectadas por el bit de interrupt disable. Sin embargo, NMI puede ser desactivada si el bit 7 del registro de control 1 del PPU ($2000) es limpiado. Cuando una NMI ocurre el sistema salta a la dirección localizada en $FFFA-$FFFB.
La interrupción reset es iniciada al comienzo del sistema y cuando el usuario presiona el botón de reinicio. Cuando esta ocurre el sistema salta a la dirección $FFFC-$FFFD.
El sistema da mayor prioridad a las interrupción reset, seguida de NMI y finalmente IRQ. La consola tiene una latencia de interrupción de 7 ciclos, que implica que tarda 7 ciclos del CPU para comenzar a ejecutar la rutina de interrupción.

3.4. Modos de Acceso

El 6502 tiene distintos modos de acceso, los cuales proveen distintas formas de acceder a memoria. Hay algunos que operan sobre los contenidos de registros. En total existen 13 modos distintos. Los mismos se detallan en [9, p. 39]

3.5. Instrucciones

El procesador tiene 56 instrucciones diferentes aunque algunas se repiten, usando distintos modos de acceso haciendo un total de 151 opcodes válidos de un posible total de 256. Las instrucciones tienen uno, dos o tres bytes de longitud, dependiendo del modo de acceso. El primer byte es el opcode y los restantes los operandos.

3.5.1. Implementación

Para sintetizar todas las instrucciones necesarias en el código, se descompuso cada instrucción en micro-operaciones. Por ejemplo para hacer un ADD (suma), primero se traen los operandos de memoria o de registros, luego se efectua la suma, después se guarda el resultado en el registro correspondiente y por último

se modifican las flags del status register adecuadas. Cada uno de estos pasos es una micro-operación.
Se puede aprovechar el hecho de que sólo se necesitan 56 micro-operaciones para poder expresar *todas* las instrucciones del procesador, para lograr un emulador del procesador mucho más corto en terminos de código.
El procedimiento que se siguió fue el siguiente. Se juntaron todas las micro-operaciones en una misma función, que recibe la instrucción a ejecutar. Para cada micro-instrucción se decide si se debe ejecutar o no, usando una máscara de 256 bits (un bit por cada instrucción). Así, si el bit i de la máscara correspondiente a la micro-instrucción j es 1, significa que j es una de las micro-instrucciones que componen la instrucción i. En otras palabras, para cada operación se accede a las máscaras de bits de cada una de las micro-instrucciones y si el bit resulta uno la micro-operación es ejecutada.
Para evitar acceder a la máscara de bits en tiempo de ejecución, se usaron templates (C++) las cuales realizan todo el proceso de las máscaras de bits en tiempo de compilación, lo que lleva a un ejecutable con 256 funciones (una para cada operación) automáticamente sintetizadas.
Para guardar las máscaras de bits en el código, se utilizó una codificación especial, la cual está detallada en util/specs/base256-explained.txt .

3.6. PPU

No se pretende explicar en detalle el completo funcionamiento de la parte gráfica de la consola. La PPU tiene una memoria separada de la CPU con su propio memory map, y a su vez contiene cuatro registros que controlan mediante distintos bits su funcionamiento. Para información detallada puede consultar [9, p. 16] y los documentos disponibles en [11].

3.6.1. Colores

La consola cuenta con 52 colores. Pero no todos pueden usarse a la vez. Existen dos paletas de 16 colores, la paleta de imagen y la de sprites, en cada una no se guardan los colores, sino los índices a los colores del sistema.

3.6.2. Composición de la pantalla

Cada frame que es mostrado en la pantalla está formado por el fondo y los sprites (objetos móviles). El fondo en un principio estaría fijo, pero es posible configurar la PPU para que el fondo se vaya desplazando horizontal por ejemplo, tal como lo hacía Super Mario Bros. Tanto los sprites como el fondo se dividen en 'tiles' de 8x8 pixeles, todos estos tiles son guardados en la pattern table, ubicada en la memoria PPU. Para saber que tile va en que lugar de la pantalla se usan los name tables. Cada name table está asociada a un pattern table. No se

pretende ahondar en el tema, pero combinando la información del tile con la pattern table se obtiene el color del pixel resultante.

3.6.3. Proceso de dibujado

Las imagenes son mostradas en la televisión por un rayo de electrones a alta velocidad que se mueve por la pantalla, de izquierda a derecha, dibujando cada pixel. Una sóla lina de pixeles es llamada scanline. Al final de cada scanline el rayo de electrones debe moverse a la siguiente línea y retornar a la izquierda antes de proceder. El tiempo que tarda en hacer esto es conocido como periodo Horizontal Blank (H-Blank).
Luego de dibujar la pantalla una vez, el rayo de electrones debe retornar a la izquina superior izquierda, para empezar el siguiente frame. El tiempo que toma hacer esto es conocido como el periodo Vertical Blank (V-Blank). Cuando el periodo V-Blank comienza, la PPU prende el bit 7 del registro I/O $2002.
Para ver el proceso completo realizado por el emulador ver el archivo emulador/ppu.cpp, línea 249.

Bibliografía

[1] OS Dev Wiki,
disponible en http://wiki.osdev.org/Main_Page.

[2] Operating System Development Series, BrokenThorn Entertainment,
disponible en http://www.brokenthorn.com/Resources/OSDevIndex.html.

[3] MosquitOS, Open-sourced hobby operating system, Tristan Seifert,
disponible en https://github.com/tristanseifert/MosquitOS.

[4] Skelix OS Tutorials, Mo Xiaoming,
disponible en http://skelix.net/skelixos/index_en.html.

[5] Cedille, A microkernel with a purpose - to keep most of the code out of the kernel, Corwin Mcknight,
disponible en https://github.com/Lionel07/Cedille.

[6] pcore, Student project of OS kernel in C/C++, Korepwx,
disponible en https://github.com/korepwx/pcore.

[7] A small hobby OS, Eduardo Veiga,
disponible en https://github.com/cinemascop89/kernel.

[8] Creating a NES emulator in C++11, Joel Yliluoma (Bisqwit), disponible en https://www.youtube.com/watch?v=y71lli8MS8s.

[9] Nintendo Entertainment System Documentation, Patrick Diskin,
disponible en http://nesdev.com/NESDoc.pdf.

[10] NES Devolpment Wiki,
disponible en http://wiki.nesdev.com/w/index.php/Nesdev.

[11] NES Technical Documents,
disponible en http://www.castledragmire.com/hynes/reference/docs/.