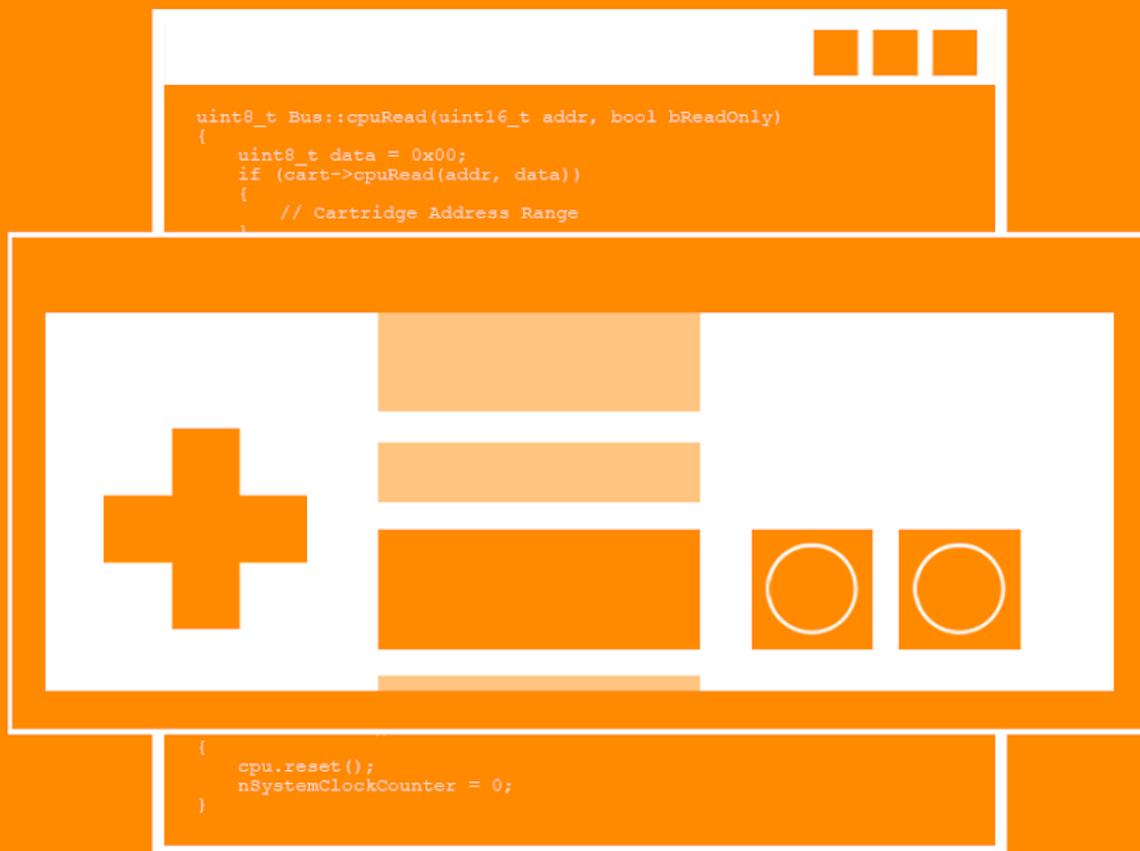


# DECONSTRUYENDO NES

UNA MIRADA AL SISTEMA DESDE LA PROGRAMACIÓN DE ALTO NIVEL

POR LARA PORTILLO Y CARLOS SÁNCHEZ



VJ1225 - SISTEMAS OPERATIVOS  
DISEÑO Y DESARROLLO DE VIDEOJUEGOS



# ÍNDICE

<b>INTRODUCCIÓN AL HARDWARE</b>	<b>4</b>
Componentes	4
Seguridad del sistema y primeros exploits	6
Actualidad: emulación y legalidad	7
<b>DESARROLLO DEL EMULADOR</b>	<b>8</b>
Conceptos básicos: notación hexadecimal y operaciones bit a bit	8
La implementación de la CPU 6502	9
La clase Bus	17
Programación de instrucciones	24
Lógica de NES: buses, RAMs, ROMs y mappers	29
La clase Cartridge	35
El renderizado de la PPU	46
<b>CONCLUSIONES Y RESULTADO FINAL</b>	<b>53</b>
<b>BIBLIOGRAFÍA</b>	<b>56</b>
<b>WEBGRAFÍA</b>	<b>56</b>



## INTRODUCCIÓN AL HARDWARE

Año 1985. *Nintendo Entertainment System* (comúnmente conocida por su abreviación, *NES*) llegaba a las tiendas de Estados Unidos como una consola de 8 bits de marcado corte nipón, tras su aterrizaje en oriente bajo el nombre de *Famicom* (abreviación, esta vez oficial, de *Family Computer*) allá por 1983. Si bien formó parte de la tercera generación de consolas (tras las máquinas clásicas de Atari y, posteriormente, las veteranas *Game & Watch*, *Arcadia 2001* y *Magnavox Odyssey*, entre otras), es considerada popularmente como una de las pioneras máquinas domésticas, gracias a la enorme contribución que realizó socialmente revitalizando de manera significativa la industria occidental, y en concreto, la estadounidense. Ayudó, además, a establecer un modelo de negocios estandarizado en un mercado todavía en claro desarrollo, que se adaptó a su contemporaneidad y que resultó extremadamente coherente, habiendo heredado sus bondades en el modelo actual. Asimismo, su fuerte catálogo y su apuesta por el desarrollo de *software* de terceros sentó un claro precedente, y nos dejó con marcas tan reconocibles como *Castlevania*, *Super Mario Bros.* o *The Legend of Zelda*.

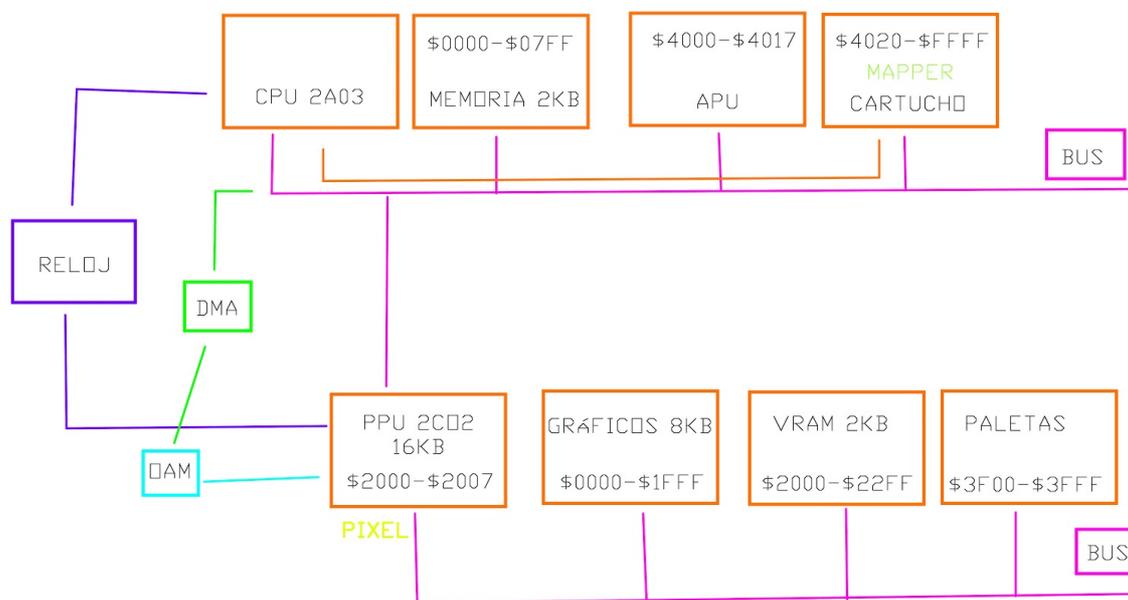
### Componentes

Abrir la máquina de Nintendo a día de hoy arroja luz sobre el arduo proceso de diseño de *hardware* que sus desarrolladores tuvieron que llevar a cabo hace ya casi cuatro décadas. En primer lugar, *NES* monta una *CPU Ricoh 2A03*, variante con audio *in-built* del prolífico *MOS 6502*. La susodicha, al carecer de memoria, debe de estar conectada a un bus de direcciones con un rango direccionable de 64 *Kb* que comparte con una memoria de 2 *Kb* (mapeada entre las direcciones \$0000 y \$07FF), con la *APU [Audio Processing Unit]* (mapeada entre las direcciones \$4000 y \$4017) y un espacio reservado para el dispositivo contenedor de las memorias que envuelven los programas que corre la *CPU* [es decir, el cartucho *per sé*] (mapeado entre las direcciones \$4020 y el final del espacio de direcciones, \$FFFF), el cual también presenta un circuito conocido como *mapper*. Este último es el responsable del *bank switching* [técnica de comunicación bidireccional con la *CPU*, que le otorga valores de diferentes direcciones de memoria al *mapper* en función del estado del cartucho], y su funcionamiento será de obligado estudio más adelante, al facilitar considerablemente el ensanchamiento del espacio dedicado a memorias de otros tipos.

Asimismo, existe otro dispositivo conectado al bus, la *PPU 2C02 [Picture Processing Unit]* (mapeada entre las direcciones \$2000 y \$2007), que se halla conectada a otro bus adicional con un rango direccionable de 16 *Kb*. Este último bus, de funcionamiento semejante, es compartido además por la memoria gráfica de 8 *Kb* encargada de guardar *sprites* o teselas (mapeada entre las direcciones \$0000 y \$1FFF, y alojada, ciertamente, en el cartucho), por una pequeña memoria de vídeo o *V-RAM* de 2 *Kb* (mapeada entre las direcciones \$2000 y \$27FF, y alojada, ciertamente, en el cartucho) y por una memoria adicional dedicada a las paletas (mapeada entre las direcciones \$3F00 y \$3FFF).

Adicionalmente, la *PPU* presenta otro dispositivo conectado directamente a ella, pero que no presenta acceso a su bus, el cual recibe el nombre de *OAM* [*Object Attribute Memory*], y se encarga de almacenar las localizaciones de los *sprites* que se imprimen por pantalla.

Todos y cada uno de estos dispositivos están subordinados a los relojes del sistema [*clocks*], que indican cuándo deben de ejecutarse tanto la *CPU* como la *PPU* (donde cada *tick* dibuja un píxel en pantalla). No obstante, la *CPU* y la *PPU* no corren a la misma velocidad, pues la frecuencia de esta última es tres veces mayor que la recibida por la *CPU*. Con tal de no ver su velocidad de procesamiento mermada, NES incluye, asimismo, un periférico dedicado exclusivamente a la comunicación fugaz entre la *CPU* y la *OAM* llamado *DMA* [*Direct Memory Access*], que suspende temporalmente la *CPU* y transfiere la memoria directamente a la *OAM*, de manera que la *CPU* puede preparar una región de su propio espacio de direcciones para los *sprites* sin la necesidad de empantanarse con la transferencia manual de información a través de un espacio limitado.



Como información adicional, cabe añadir que la *PPU* permitía y permite almacenar hasta 64 *sprites* del tamaño de  $8 \times 8$  píxeles – 8 píxeles en horizontal por 8 en vertical – u  $8 \times 16$ , además de facilitar la realización de un *scrolling* horizontal sobre un plano de fondo y la generación de imágenes de una resolución de 256 píxeles de ancho por 240 de alto, con la posibilidad de mostrar hasta 8 *sprites* distintos en una misma imagen.

Con estas características en mano, resulta sencillo preguntarse para qué podía la consola de Nintendo almacenar 64 *sprites* si solo podría mostrar 8 de ellos en cada imagen.

La respuesta residía en la mayoría de consolas de la época (y de la actualidad): dispositivos como NES enviaban y envían entre 50 y 60 imágenes distintas por segundo al televisor para darnos la sensación, como hace el cine, de movimiento. La consola, por tanto, podía mandar en cada uno de esos llamados *frames* hasta 8 sprites distintos hasta haber mostrado los 64, y posteriormente repetir el proceso.

Debido a la rapidez a la que pasaba todo esto, el usuario podía ver una gran cantidad de *sprites* moviéndose por una pantalla donde solo se apreciaría un ligero parpadeo. Por supuesto, la pericia de los programadores y otros chips contenidos en el cartucho con capacidad de operaciones extras permitieron la implementación en NES de funciones que originalmente no fueron concebidas para la consola; prueba de ello sería el *scrolling* horizontal y vertical simultáneo de propuestas como *Super Mario Bros 3*.

### Seguridad del sistema y primeros exploits

En pro de adaptarse a las necesidades del mercado americano, Nintendo concibió su archiconocido *Nintendo Seal of Quality*, que controlaba el catálogo de juegos de la consola, mientras implementaba un sistema de seguridad, bautizado como *Nintendo CIC (Checking Integrated Circuit)*, que impedía la ejecución de juegos de importación y cartuchos no originales. *CIC* se componía de dos chips: uno que formaba parte del núcleo del sistema, y otro que debía de permanecer en el cartucho. Durante el encendido de la máquina, los chips se comunicaban, y solo al hacerlo exitosamente se cargaba el cartucho; de no ser así, *NES* se reiniciaba.

No obstante, el pasar del tiempo permitió a las compañías aprovecharse de los *exploits* del sistema con tal de inhabilitarlo. Camerica, una de ellas, lo estudió a fondo hasta llegar a la conclusión de que, si se enviaba un pico de voltaje al chip al comienzo de la inicialización, la comunicación tan siquiera podía tener tiempo a realizarse, lo que procedía a la carga del cartucho. Este método, aunque funcional, quedó rápidamente relegado a la obsolescencia gracias al lanzamiento de la segunda versión de la consola, la cual prescindía del chip *CIC* interno en una búsqueda por abaratar costes de producción.

Tuvo que verse sucedido, por ende, por el que acabaría siendo el procedimiento más conocido: la implementación del *Rabbit Chip* de Tengen Inc., empresa renacida de las cenizas de Atari Games (muy damnificada, como parte de Warner Bros., durante la crisis del videojuego del 83) tras su compra en 1985 por parte de Jack Tramiel, fundador de la empresa Commodore. Tramiel era de aquellos que criticaban a Nintendo por sus restrictivas políticas de lanzamientos; los japoneses difícilmente permitían a compañías externas (*third-parties*) publicar más de cinco juegos anuales en sus consolas, debiendo estos, además, de ser exclusivos de NES durante al menos dos años. También buscaban el control absoluto de la producción, recayendo en la empresa nipona decisiones tales como cuántos cartuchos se fabricarían de cada una de las propuestas que se lanzasen en su sistema.

Tras fracasar en la negociación con Nintendo de unas mejores condiciones hacia Tengen Inc., Tramiel se interesó en maximizar su rendimiento económico mediante ingeniería inversa, investigando el funcionamiento del código fuente de los chips *CIC* (para lo que necesitó acceder a la oficina de patentes) en la más sincera búsqueda por concebir el suyo propio, el cual acabaría recibiendo, tras diversos intentos fallidos, el nombre de *Rabbit*.

Por supuesto, años más tarde los cartuchos no oficiales de los que se beneficiaba este chip (que prescindían del citado *Nintendo Seal of Quality*, y que presentaban un característico color negro) tuvieron que ser retirados del mercado, fruto de una extensa batalla legal entre ambas compañías.

### Actualidad: emulación y legalidad

Como consecuencia de la repercusión mediática que tuvieron estas y otras batallas legales, así como de la desinformación bulliciosa producida por el auge de los foros de internet allá por principios de siglo, la emulación tiende a relacionarse erróneamente con la piratería, y por tanto, con la ilegalidad. Bien es cierto que, tal y como dicta la ley, la emulación puede llegar a considerarse ilegal sin la autorización expresa del poseedor de los derechos de autor, entendiéndose como propiedad intelectual aquel bien intangible cuyo valor queda asociado al autor conceptual. No obstante, lo cierto es que la ley únicamente ampara los casos relacionados con la distribución malintencionada de dicho *software* (considerándose la descarga del mismo una parte activa del proceso), y no con su ejecución. Por ende, adquirir una copia de una entrega, conseguir su código y ejecutarlo en un emulador está lejos de ser un acto ilícito.

Esto último es posible gracias a dispositivos como los *ROM dumper*, que permiten extraer el contenido de la memoria de los cartuchos a través de la conexión directa con el circuito de su entrada. Estos *ROM dumper* funcionan de una forma similar al emulador que vamos a intentar crear, por lo que ejemplificar con ellos de manera recurrente podría sernos de utilidad más adelante.

## DESARROLLO DEL EMULADOR

### Conceptos básicos: notación hexadecimal y operaciones bit a bit

Antes de lanzarnos a escribir el código de nuestro emulador, resulta de vital importancia detenerse a estudiar los componentes y conceptos básicos del sistema en pro de realizar un primer acercamiento documentado. Estudiarlos es una labor imposible sin antes detenerse en un punto pivote que nos acompañará a lo largo y ancho de nuestro estudio en forma de operaciones lógicas y aplicaciones de diversos tipos: la notación hexadecimal. Al estar plenamente familiarizados con ella, en este estudio no llevaremos a cabo una explicación detallada de sus bondades frente a la notación binaria (como el hecho de que su valor sea fácilmente percible o comparable frente a la ininteligibilidad de esta última, sin renegar de su optimalidad) ni de su funcionamiento, mas siempre conviene recordarlo aunque sea con un ejemplo rápido.

Así, decimos que el número decimal 65 [base 10] es, en notación binaria [base 2], 01000001 (ya que  $65 = 6 \cdot 10^1 + 5 \cdot 10^0 = 1 \cdot 2^6 + 1 \cdot 2^0$ ), y en notación hexadecimal [base 16], 0041 (ya que  $65 = 4 \cdot 16^1 + 1 \cdot 16^0$ ). De la misma manera, cabe recordar que, cuando hablamos de operaciones *bit a bit* aplicadas a un lenguaje de programación como C++, utilizamos el tipo de datos de 8 *bits* 'char' para almacenar información, cuyos tres primeros *bits* representan valores comprendidos entre 0 y 7. Los dos *bits* más próximos a ellos representan uno de los cuatro estados posibles de la variable, y los dos consiguientes, *switches* en el *hardware* (con tan solo dos posibilidades independientes cada uno: ON/OFF). El *bit* sobrante no es requerido por parte del sistema para la representación de información, aunque sí que lo es dada la incapacidad de la máquina para almacenar únicamente siete *bits*; en la práctica, por consecuencia, no se utiliza.

Pero, si se plantean esta clase de incongruencias, ¿por qué utilizamos operaciones *bit a bit*? Simplemente, porque resulta un procesamiento de los datos (lo que incluye extraer e interpretar las partes relevantes del conjunto de 8 *bits*) eficiente. Al extraer, por ejemplo, los primeros tres dígitos de un 'char' (aquellos que guardan un valor comprendido entre 0 y 7), las operaciones *bit a bit* nos facilitan crear una máscara binaria de los dígitos interesados, de manera que no se precisa la lectura del resto de dígitos. Cabe destacar que algunos de los operadores con los que trabajaremos permitirán invertir el valor de los dígitos (~), extraerlos (&) y establecerlos (|), así como intercambiarlos (>>) o excluirlos (^). Al trabajar con operaciones *bit a bit*, en ocasiones (como la nuestra) puede ser conveniente crear una estructura de datos que represente a la palabra binaria ya dividida en propiedades oportunas.

Esto configura una de las principales razones por las que utilizaremos C++ en este proceso, pues sus *bitfields* se antojan como una solución idónea al problema propuesto. Un ejemplo de estructura de C++ que podría servirnos a la hora de representar tipos de datos 'char' podría ser la siguiente:

```

struct
{
    char unused : 1
    char sw2 : 1; // 1 dígito disponible para el switch 2
    char sw1 : 1; // 1 dígito disponible para el switch 1
    char state: 2; // 2 dígitos disponibles para el estado
    char value : 3; // 3 dígitos disponibles para el almacenaje de valores
}

```

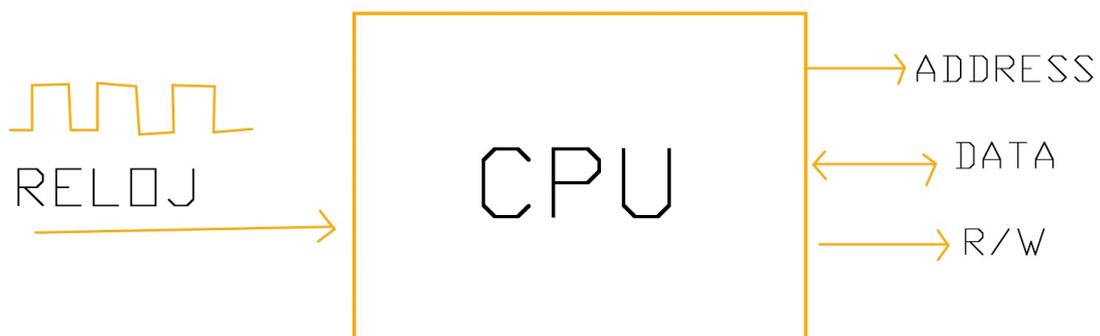
Asimismo, C++ puede resultar tremendamente intuitivo a la hora de descifrar cómo los componentes de NES se comunican entre sí, algo vital si queremos comprender y simular con cierta precisión el funcionamiento de la máquina.

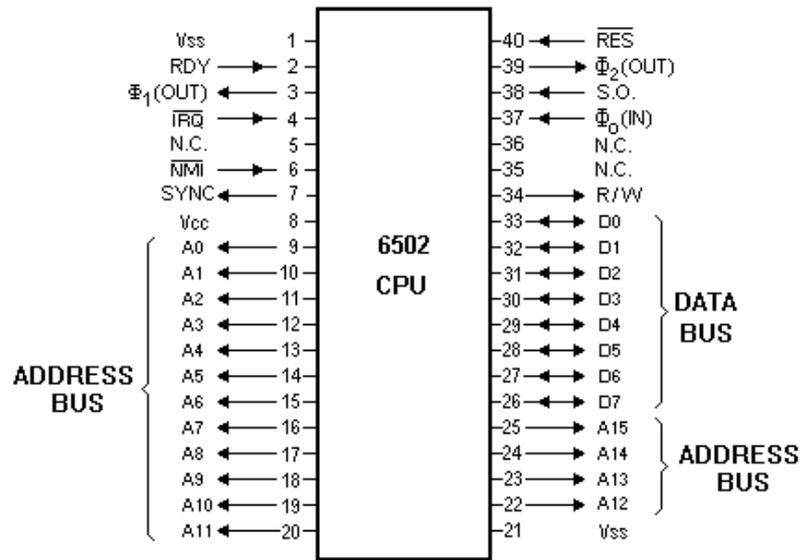
### La implementación de la CPU 6502

A la hora de emular la CPU, lo que debemos estudiar en primer lugar es el nivel de abstracción con el que vamos a diseñar; no nos interesa el funcionamiento eléctrico de la misma (niveles de voltaje, etcétera) sino su comportamiento *per sé*. Partiendo de dicha base, cabe destacar que la CPU como elemento individual es uno de los componentes más sencillos de emular, ya que por sí sola es un elemento que pasa desapercibido al no interactuar con otros componentes. No obstante, para que una CPU sea funcional debe conectarse con el mundo exterior gracias a una *address* (16 bits) que pueda leer y escribir los datos (8 bits), debiendo así analizar las señales que nos permiten cumplir esta función.

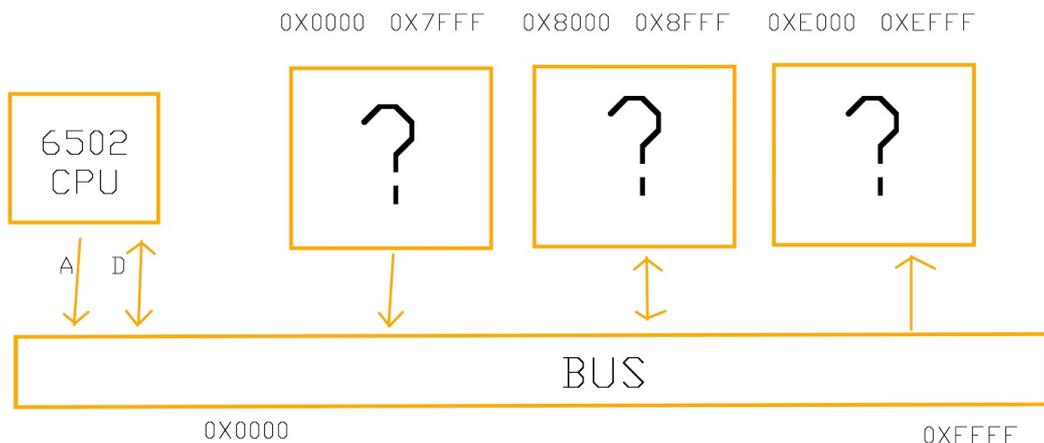
Para indicar si estamos escribiendo o leyendo los datos, debemos añadir una nueva señal - que, en el caso de nuestra emulación, no se implementará como una señal eléctrica -, así como un reloj que fuerce a la CPU para cambiar su estado en cada pulso o ciclo. En cada uno de estos pulsos la CPU puede evaluar si los *inputs* están respondiendo correctamente al cambio de sus *outputs*.

#### 6502 CPU



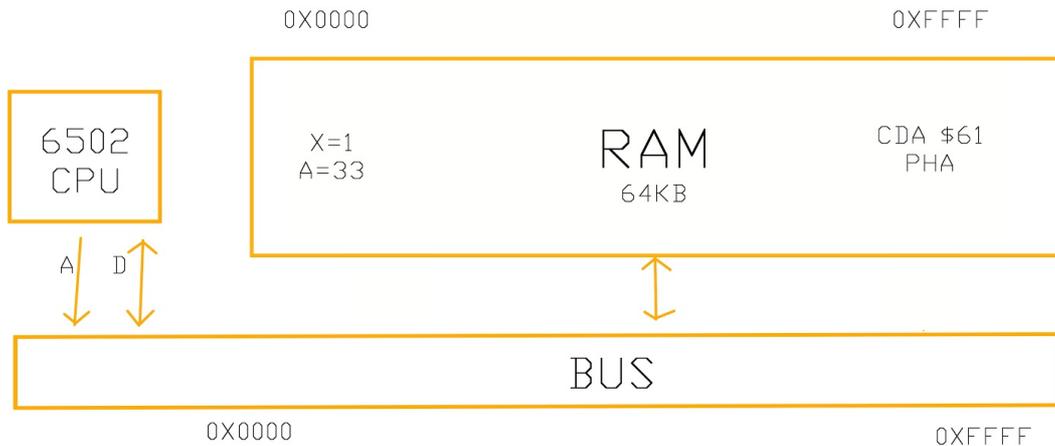


Con estas explicaciones podemos entender que, desprendidos del comportamiento eléctrico, la *CPU* no se puede emular en solitario, ya que por sí sola no puede evaluar intrínsecamente si los *inputs* están ejecutándose correctamente; simplemente responde a ellos. Es por esto que debemos de emular otro componente a su vez, que en nuestro caso será un bus al que se le conectarán la *address* (de 16 bits; irá desde *0x0000* hasta *0xFFFF*) y las líneas de datos. El funcionamiento del mismo será simple: todos los componentes conectados a él deberán responder bien añadiendo datos para que la *CPU* los lea o bien guardando los datos que la *CPU* le envía al bus a través de una señal de *R/W*. Para entenderlo mejor, pondremos tres componentes conectados al bus; no nos interesa para qué sirven, solo si leen, escriben datos o realizan ambas operaciones.

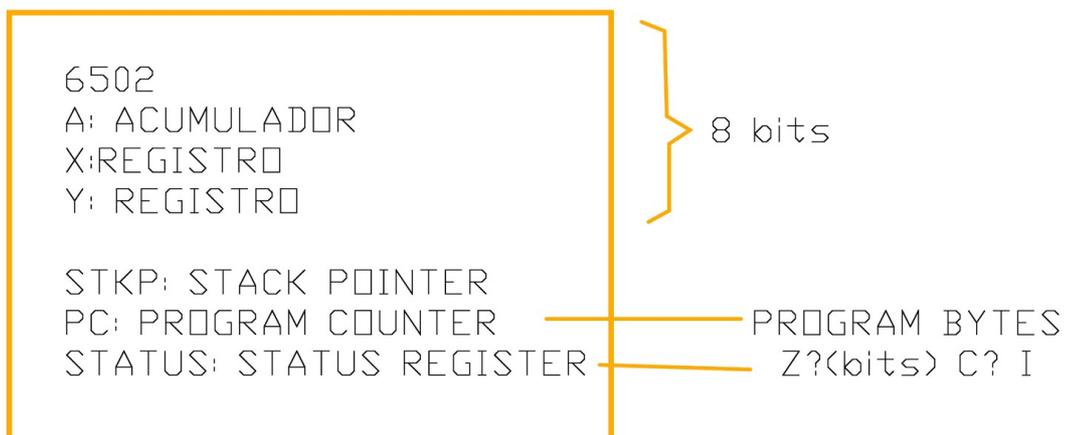


Gracias a la ilustración superior podemos observar cómo si la *CPU*, por ejemplo, lanza una orden a la *address* *0x3000*, el primer componente responderá porque es sensible a ella, enviando datos al bus que acabarán siendo leídos por la *CPU*. En el caso del segundo componente, que puede leer y escribir, no bastará con realizar dicho procedimiento, sino que también se tendrá que mandar la información a escribir.

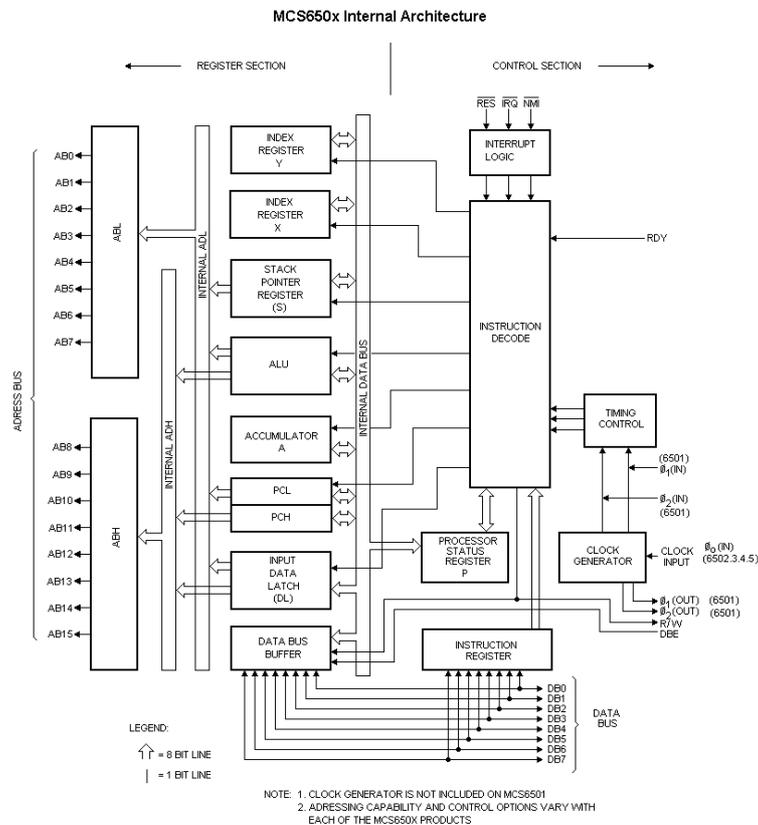
Ya entendido el funcionamiento, trabajaremos con un bus de un único componente, el cual será una *RAM* que podrá leer y escribir datos. Será sensible a toda la *address* de 16 bits, y tendrá un tamaño de 64 Kb. No solo contendrá la información valiosa que utilizaremos para usar los programas, sino también los programas en sí. Por tanto, la mayoría del tiempo la *CPU* estará extrayendo información de la *RAM* para usarla en esos programas, y periódicamente leerá y escribirá información en diferentes localizaciones. Esta es una estructura clásica *Von Neumann*.



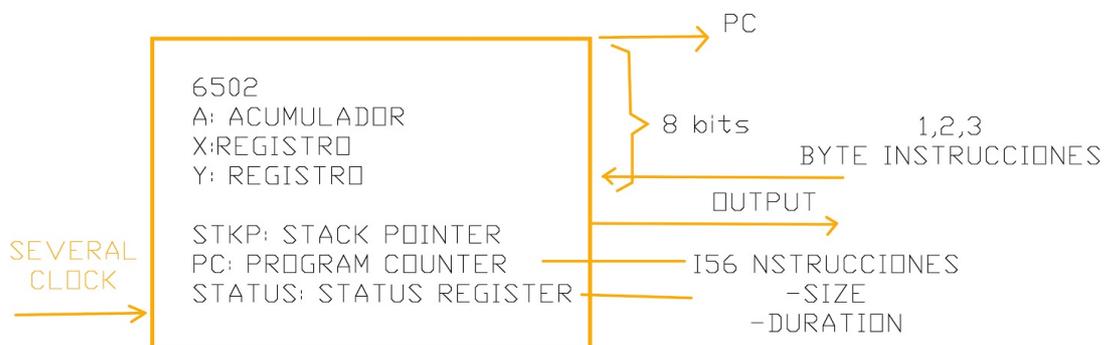
Dentro de la *CPU* hay tres registros principales de 8 bits: *A - Accumulator*, *X - Register*, *Y - Register*. Aunque tengan nombres diferentes, funcionalmente son muy parecidos. También encontramos un *Stack Pointer*, un *Program Counter* y un *Status Register*. Este último, con un tamaño de 8 bits, nos indica el estado de la *CPU* gracias a que contiene varios bits (*Z* si el resultado final fue 0, *C* si ha habido acarreo, *I* instruye a la *CPU* para que interrumpa o anule interruptores). Por su parte, el contador de programas es un contador de 16 bits que guarda la *address* del siguiente programa que la *CPU* debe leer. El *stack pointer*, por último, contiene un número de 8 bits que señala a una *address* en la memoria; el *pull* y el *push* incrementan y decrementan la *address*.



El diagrama de a continuación, conocido como *datasheet*, permite reconocer todos estos componentes de un vistazo, aunque para la emulación no nos interesará en su mayoría al ser un esquema del funcionamiento eléctrico.



El funcionamiento de *MOS 6502* a nivel emulable precisa del uso de varios ciclos para ejecutar varias funciones a la vez y conseguir la instrucción. Para cada instrucción tendremos que tener en cuenta el tamaño de esta y su duración, habiendo un total de 65 instrucciones legales - las cuales explicaremos más adelante - que pueden ser mutadas para cambiar su tamaño y duración en función de sus argumentos.



Una instrucción clásica (*LDA*) puede constituirse de 2 *bytes*. Partiendo de las explicaciones previas, el acumulador (*accumulator*) se carga con el valor 65, y los datos se proporcionan inmediatamente. De este modo, podemos leer un *byte* de la instrucción y otro de los datos inmediatos; confirmamos sus 2 *bytes*.

Otro ejemplo de instrucción *LDA* podría ser una cargada desde la memoria. Esta vez con 3 *bytes* (ya que su memoria es de 16 *bits*), con ella cargaremos el acumulador. Un *byte* representará la instrucción y los otros dos representarán la información necesaria para que dicha instrucción se ejecute correctamente. Como último ejemplo, hablaremos de la instrucción ‘*Clear the Carry bit*’ (CLC) en el registrador de estado, la cual no necesita ningún dato para ejecutarse; se puede representar con un solo *byte*.

Aunque funcionalmente las instrucciones sean iguales, se representan de maneras diferentes. Por tanto, para emular las instrucciones debemos emular sus modos funcionales y de *address*, al igual que el número de ciclos que necesitan para ser implementadas. Por suerte, toda esta información podemos obtenerla con el primer *byte*. Mirando en la tabla del *datasheet* que sucede a estas líneas podemos ver la información necesaria para representar las 56 instrucciones en una tabla de 16 × 16, lo cual nos da 256 instrucciones potenciales.

R650X, R651X

R6500 Microprocessors (CPU)

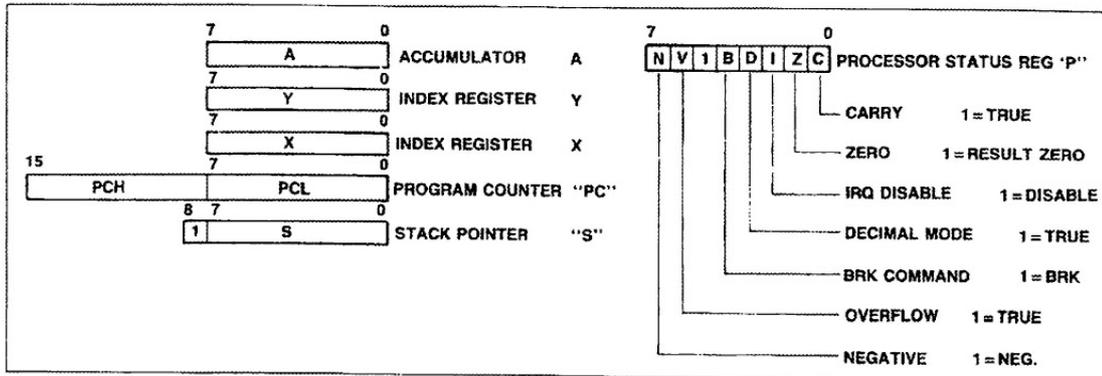
INSTRUCTION SET

The R6500 CPU has 56 instruction types which are enhanced by up to 13 addressing modes for each instruction. The

Accumulator, index registers, Program Counter, Stack Pointer and Processor Status Register are illustrated below.

Alphabetic Listing of Instruction Set

Mnemonic	Function	Mnemonic	Function
ADC	Add Memory to Accumulator with Carry	JMP	Jump to New Location
AND	"AND" Memory with Accumulator	JSR	Jump to New Location Saving Return Address
ASL	Shift Left One Bit (Memory or Accumulator)	LDA	Load Accumulator with Memory
BCC	Branch on Carry Clear	LDX	Load Index X with Memory
BCS	Branch on Carry Set	LDY	Load Index Y with Memory
BEQ	Branch on Result Zero	LSR	Shift One Bit Right (Memory or Accumulator)
BIT	Test Bits in Memory with Accumulator	NOP	No Operation
BMI	Branch on Result Minus	ORA	"OR" Memory with Accumulator
BNE	Branch on Result not Zero	PHA	Push Accumulator on Stack
BPL	Branch on Result Plus	PHP	Push Processor Status on Stack
BRK	Force Break	PLA	Pull Accumulator from Stack
BVC	Branch on Overflow Clear	PLP	Pull Processor Status from Stack
BVS	Branch on Overflow Set	ROL	Rotate One Bit Left (Memory or Accumulator)
CLC	Clear Carry Flag	ROR	Rotate One Bit Right (Memory or Accumulator)
CLD	Clear Decimal Mode	RTI	Return from Interrupt
CLI	Clear Interrupt Disable Bit	RTS	Return from Subroutine
CLV	Clear Overflow Flag	SBC	Subtract Memory from Accumulator with Borrow
CMP	Compare Memory and Accumulator	SEC	Set Carry Flag
CPX	Compare Memory and Index X	SED	Set Decimal Mode
CPY	Compare Memory and Index Y	SEI	Set Interrupt Disable Status
DEC	Decrement Memory by One	STA	Store Accumulator in Memory
DEX	Decrement index X by One	STX	Store Index X in Memory
DEY	Decrement Index Y by One	STY	Store Index Y in Memory
EOR	"Exclusive-OR" Memory with Accumulator	TAX	Transfer Accumulator to Index X
INC	Increment Memory by One	TAY	Transfer Accumulator to Index Y
INX	Increment Index X by One	TSX	Transfer Stack Pointer to Index X
INY	Increment Index Y by One	TXA	Transfer Index X to Accumulator
		TXS	Transfer Index X to Stack Register
		TYA	Transfer Index Y to Accumulator



Programming Model

**INSTRUCTION SET OP CODE MATRIX**

The following matrix shows the Op Codes associated with the R6500 family of CPU devices. The matrix identifies the hexadecimal code, the mnemonic code, the addressing mode, the

number of instruction bytes, and the number of machine cycles associated with each Op Code. Also, refer to the instruction set summary for additional information on these Op Codes.

MSD	LSD															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	BRK Implied 1 7	ORA (IND, X) 2 6				ORA ZP, X 2 3	ASL ZP, X 2 5		PHP Implied 1 3	ORA IMM 2 2	ASL Accum 1 2			ORA ABS, X 3 4	ASL ABS, X 3 6	
1	BPL Relative 2 2**	ORA (IND), Y 2 5*				ORA ZP, X 2 4	ASL ZP, X 2 6		CLC Implied 1 2	ORA ABS, Y 3 4*				ORA ABS, X 3 4*	ASL ABS, X 3 7	
2	JSR Absolute 3 6	AND (IND, X) 2 6			BIT ZP 2 3	AND ZP 2 3	ROL ZP 2 5		PLP Implied 1 4	AND ZP 2 2	ROL Accum 1 2		BIT ABS 3 4	AND ABS 3 4	ROL ABS 3 6	
3	BMI Relative 2 2**	AND (IND), Y 2 5*				AND ZP, X 2 4	ROL ZP, X 2 6		SEC Implied 1 2	AND ABS, Y 3 4*				AND ABS, X 3 4*	ROL ABS, X 3 7	
4	RTI Implied 1 6	EOR (IND, X) 2 6				EOR ZP, X 2 3	LSR ZP 2 5		PHA Implied 1 3	EOR IMM 2 2	LSR Accum 1 2		JMP ABS 3 3	EOR ABS 3 4	LSR ABS 3 6	
5	BVC Relative 2 2**	EOR (IND), Y 2 5*				EOR ZP, X 2 4	LSR ZP, X 2 6		CLI Implied 1 2	EOR ABS, Y 3 4*				EOR ABS, X 3 4*	LSR ABS, X 3 7	
6	RTS Implied 1 6	ADC (IND, X) 2 6				ADC ZP 2 3	ROR ZP 2 5		PLA Implied 1 4	ADC IMM 2 2	ROR Accum 1 2		JMP Indirect 3 5	ADC ABS 3 4	ROR ABS 3 6	
7	BVS Relative 2 2**	ADC (IND), Y 2 5*				ADC ZP, X 2 4	ROR ZP, X 2 6		SEI Implied 1 2	ADC ABS, Y 3 4*				ADC ABS, X 3 4*	ROR ABS, X 3 7	
8		STA (IND, X) 2 6			STY ZP 2 3	STA ZP 2 3	STX ZP 2 3		DEY Implied 1 2		TXA Implied 1 2		STY ABS 3 4	STA ABS 3 4	STX ABS 3 4	
9	BCC Relative 2 2**	STA (IND), Y 2 6			STY ZP, X 2 4	STA ZP, X 2 4	STX ZP, Y 2 4		TYA Implied 1 2	STA ABS, Y 3 5	TXS Implied 1 2			STA ABS, X 3 5		
A	LDY IMM 2 2	LDA (IND, X) 2 6	LDX IMM 2 2		LDY ZP 2 3	LDA ZP 2 3	LDX ZP 2 3		TAY Implied 1 2	LDA IMM 2 2	TAX Implied 1 2		LDY ABS 3 4	LDA ABS 3 4	LDX ABS 3 4	
B	BCS Relative 2 2**	LDA (IND), Y 2 5*			LDY ZP, X 2 4	LDA ZP, X 2 4	LDX ZP, Y 2 4		CLV Implied 1 2	LDA ABS, Y 3 4*	TSX Implied 1 2		LDY ABS, X 3 4*	LDA ABS, X 3 4*	LDX ABS, Y 3 4*	
C	CPY IMM 2 2	CMP (IND, X) 2 6			CPY ZP 2 3	CMP ZP 2 3	DEC ZP 2 5		INY Implied 1 2	CMP IMM 2 2	DEX Implied 1 2		CPY ABS 3 4	CMP ABS 3 4	DEC ABS 3 6	
D	BNE Relative 2 2**	CMP (IND), Y 2 5*				CMP ZP, X 2 4	DEC ZP, X 2 6		CLD Implied 1 2	CMP ABS, Y 3 4*				CMP ABS, X 3 4*	DEC ABS, X 3 7	
E	CPX IMM 2 2	SBC (IND, X) 2 6			CPX ZP 2 3	SBC ZP 2 3	INC ZP 2 5		INX Implied 1 2	SBC IMM 2 2	NOP Implied 1 2		CPX ABS 3 4	SBC ABS 3 4	INC ABS 3 6	
F	BEO Relative 2 2**	SBC (IND), Y 2 5*				SBC ZP, X 2 4	INC ZP, X 2 6		SED Implied 1 2	SBC ABS, Y 3 4*				SBC ABS, X 3 4*	INC ABS, X 3 7	

0	BRK	—OP Code
0	Implied	—Addressing Mode
1 7		—Instruction Bytes; Machine Cycles

\*Add 1 to N if page boundary is crossed.  
 \*\*Add 1 to N if branch occurs to same page;  
 add 2 to N if branch occurs to different page.

R650X, R651X

R6500 Microprocessors (CPU)

INSTRUCTION SET SUMMARY

Mnemonic	OPERATION	IMMEDIATE		ABSOLUTE		ZERO PAGE		ACCUM		IMPLIED		IND. X		IND. Y		Z PAGE		ABS. X		ABS. Y		RELATIVE		INDIRECT		Z PAGE		PROCESSOR STATUS CODES		Mnemonic													
		OP	OP	OP	OP	OP	OP	OP	OP	OP	OP	OP	OP	OP	OP	OP	OP	OP	OP	OP	OP	OP	OP	OP	OP	OP	OP	OP	OP		OP												
ADC	A ← M + C ← A (4-12)	69	2	2	6C	4	3	65	3	2			65	6	2	71	5	75	4	2	7E	4	3	79	4	1					N	V	...	...	Z	C	ADC						
AND	A ← M ← A (1)	29	2	2	2D	4	3	25	3	2			21	5	2	31	5	35	4	2	3D	4	3	39	4	3					N	V	...	...	Z	C	AND						
ASL	C ← C ← C ← 0				DE	6	3	66	5	2	DA	2	1					16	6	2	1E	7	3								N	V	...	...	Z	C	ASL						
BCC	BRANCH ON C ← 0 (2)																															N	V	...	...	Z	C	BCC					
BCS	BRANCH ON C ← 1 (2)																																N	V	...	...	Z	C	BCS				
BEG	BRANCH ON Z ← 1 (2)																																N	V	...	...	Z	C	BEG				
BET	A ← M				2C	4	3	24	3	2																							N	V	...	...	Z	C	BET				
BMI	BRANCH ON M ← 1 (2)																																	N	V	...	...	Z	C	BMI			
BNE	BRANCH ON Z ← 0 (2)																																	N	V	...	...	Z	C	BNE			
BPL	BRANCH ON M ← 0 (2)																																	N	V	...	...	Z	C	BPL			
BRK	BREAK										00	2	1																					N	V	...	...	Z	C	BRK			
BVC	BRANCH ON V ← 0 (2)																																	N	V	...	...	Z	C	BVC			
BVS	BRANCH ON V ← 1 (2)																																		N	V	...	...	Z	C	BVS		
CLC	0 ← C																																		N	V	...	...	Z	C	CLC		
CLE	0 ← E																																		N	V	...	...	Z	C	CLE		
CLV	0 ← V																																		N	V	...	...	Z	C	CLV		
CMV	A ← M				CD	4	3	CA	3	2																									N	V	...	...	Z	C	CMV		
CPY	E ← M				ED	4	3	E4	3	2																									N	V	...	...	Z	C	CPY		
CPY	Y ← M				CD	4	3	CA	3	2																									N	V	...	...	Z	C	CPY		
DEC	M ← M - 1				CE	6	3	CE	5	2																									N	V	...	...	Z	C	DEC		
DEX	X ← X - 1																																		N	V	...	...	Z	C	DEX		
DEY	Y ← Y - 1																																		N	V	...	...	Z	C	DEY		
EDR	A ← M - A				4D	4	3	45	3	2																										N	V	...	...	Z	C	EDR	
INC	M ← M + 1				EE	6	3	E6	5	2																										N	V	...	...	Z	C	INC	
INX	X ← X + 1																																		N	V	...	...	Z	C	INX		
INY	Y ← Y + 1																																		N	V	...	...	Z	C	INY		
JMP	JUMP TO NEW LOG				4C	3	3																												N	V	...	...	Z	C	JMP		
JSR	JUMP SUB				2C	5	3																												N	V	...	...	Z	C	JSR		
LDA	M ← A				A9	2	2	AD	4	3	AA	2	1																						N	V	...	...	Z	C	LDA		
LDX	M ← X				A2	2	2	A4	4	3	AA	2	1																							N	V	...	...	Z	C	LDX	
LDY	M ← Y				A0	2	2	AC	4	3	AA	2	1																							N	V	...	...	Z	C	LDY	
LSR	C ← C ← C ← 0				4E	6	3	4E	5	2	AA	2	1																							N	V	...	...	Z	C	LSR	
NOP	NO OPERATION																																			N	V	...	...	Z	C	NOP	
ORA	A ← M ← A				09	2	2	0D	4	3	0A	2	1																							N	V	...	...	Z	C	ORA	
PHA	A ← M ← S ← 1 ← S																																			N	V	...	...	Z	C	PHA	
PHP	P ← M ← S ← 1 ← S																																			N	V	...	...	Z	C	PHP	
PLA	S ← 1 ← S ← M ← A																																			N	V	...	...	Z	C	PLA	
PLP	S ← 1 ← S ← M ← P																																				N	V	...	...	Z	C	PLP
RDL	← C ← 0				2E	6	3	2E	5	2	AA	2	1																							N	V	...	...	Z	C	RDL	
RDR	← C ← 0				6E	6	3	6E	5	2	AA	2	1																							N	V	...	...	Z	C	RDR	
RTI	RTN SUB																																			N	V	...	...	Z	C	RTI	
RTS	RTN SUB																																			N	V	...	...	Z	C	RTS	
SBC	A ← M ← C ← A				E9	2	2	ED	4	3	E5	3	2																							N	V	...	...	Z	C	SBC	
SEC	1 ← C																																			N	V	...	...	Z	C	SEC	
SED	1 ← D																																			N	V	...	...	Z	C	SED	
SEI	1 ← E																																			N	V	...	...	Z	C	SEI	
STA	A ← M				8D	4	3	85	3	2																										N	V	...	...	Z	C	STA	
STX	X ← M				BE	4	3	86	3	2																										N	V	...	...	Z	C	STX	
STY	Y ← M				BC	4	3																																				

El primer *byte* se usará como índice de la tabla. Los 4 *bits* últimos de ese *byte* representarán la columna, y los 4 *bits* primeros, la fila. Dentro de cada celda veremos dos números y el tipo; el número de la derecha representará el número de ciclos, mientras que el restante ilustrará el número de *bytes* que esa instrucción es capaz de utilizar. El número que representa el número de ciclos puede tener una pequeña indicación arriba a la derecha.

Gracias a esta tabla podemos construir las bases de la emulación. Tal y como podemos apreciar, no todos los espacios de la tabla tienen información: si la instrucción nos llevase a uno de esos espacios en blanco, entonces sería una instrucción 'ilegal'. En tal caso, la instrucción no devolvería el resultado esperado por sus diseñadores, sino que la *CPU* la trataría a través de diversos 'microprogramas' encargados de esta clase de excepciones. En este estudio no trabajaremos con estas instrucciones ilegales.

#### //Pasos para emular la CPU

1. Leer byte @PC
2. OPCODE[byte]--> modo dirección
3. Leer 0,1 o 2 o más bytes
4. Ejecutar
5. Esperar, contar ciclos, completar

### La clase *Bus*

Ahora que tenemos las bases teóricas de la CPU, podemos empezar, por fin, a escribir el código de la emulación. Este tendrá dos clases principales: '*Bus*' y '*olc6502*'. La clase *Bus()* podrá ser leída y escrita por la *CPU* con información de 8 *bits*. Para la emulación en C++ trabajaremos con varios ficheros: *bus.h*, *bus.cpp*, *olc6502.h* y *olc6502.cpp*. Naturalmente, el bus tendrá conectados varios dispositivos, que en este caso, serán la *CPU* y una *RAM* de 64 *Kb*, tal y como refleja el código descrito a continuación.

A la *RAM* le asignaremos un rango dentro del código de *bus.cpp* (tal y como se aprecia en las sentencias subrayadas de naranja). En este apartado no nos servirá de nada - ya que le daremos el rango entero de instrucciones -, pero será vital más adelante.

#### //Bus.h

```
#pragma once
#include <stdint>
#include <array>
#include "olc6502.h"

class Bus
{
public:
    Bus();
    ~Bus();
```

```
public: // Dispositivos
    olc6502 cpu;

    // RAM falsa
    std::array<uint8_t, 64 * 1024> ram;

public: // CPU puede acceder para leer y escribir
    void write(uint16_t addr, uint8_t data);
    uint8_t read(uint16_t addr, bool bReadOnly = false); //Para evitar problemas
                                                    ahora está en false, para que no funcione
};
```

```
//Bus.cpp
#include "Bus.h"

Bus::Bus()
{
    // Conecta la CPU al bus de comunicación
    cpu.ConnectBus(this);

    // Borra los contenidos de la RAM, por si acaso
    for (auto &i : ram) i = 0x00;
}

Bus::~Bus() {}

void Bus::write(uint16_t addr, uint8_t data)
{
    if (addr >= 0x0000 && addr <= 0xFFFF)
        ram[addr] = data;
}

uint8_t Bus::read(uint16_t addr, bool bReadOnly)
{
    if (addr >= 0x0000 && addr <= 0xFFFF)
        return ram[addr];

    return 0x00;
}
```

Para conectarlo a la CPU añadiremos unas líneas de código en *ol6502.h*. De este modo esta apuntará al bus. Así, en su código añadiremos líneas de código para llamar bien a las funciones, y que su contenido se lea y escriba correctamente.

```
//En la parte public:

void ConnectBus(Bus *n) { bus = n; }
```

```

//En la parte private:
// Vinculaciones al bus de comunicación
Bus      *bus = nullptr;
uint8_t  read(uint16_t a);
void      write(uint16_t a, uint8_t d);

//En ol6502.cpp
uint8_t  olc6502::read(uint16_t a)
{
    // En operaciones normales, el "read only" será false.
    // Algunos dispositivos pueden querer cambiar su estado al ser leídos,
    // mientras que el desensamblador querrá leerlos sin modificar su estado.
    return bus->read(a, false);
}

// Escribe un byte al bus en la dirección especificada
void olc6502::write(uint16_t a, uint8_t d)
{
    bus->write(a, d);
}

```

Ahora que tenemos conectada la *CPU* al bus podemos programar los componentes del núcleo de la *CPU*. Empezaremos por una enumeración de los *bits* del registro de estado.

```

enum FLAGS6502
{
    C = (1 << 0),    // Bit de acarreo
    Z = (1 << 1),    // Cero
    I = (1 << 2),    // Deshabilitar interrupciones
    D = (1 << 3),    // Modo decimal (en desuso; NES no lo tenía
                    // implementado)
    B = (1 << 4),    // "Break"
    U = (1 << 5),    // Sin uso
    V = (1 << 6),    // Desbordamiento
    N = (1 << 7),    // Negativo
};

```

Como bien se puede observar, cada *bit* comunica una orden diferente. Los *bits V* y *N*, sin ir más lejos, usan variables firmadas, cuyo uso explicaremos más adelante.

Para definir las indicaciones utilizaremos una variable de 8 *bits* llamada *status* que representará el registrador de estado. Usando el mismo método, crearemos el resto de elementos mencionados de la *CPU*.

```

uint8_t  a   = 0x00;    // Registro A (Accumulator)
uint8_t  x   = 0x00;    // Registro X
uint8_t  y   = 0x00;    // Registro Y
uint8_t  stkp = 0x00;    // Puntero Stack (apunta a la localización en el bus)
uint16_t pc  = 0x0000;    // Program Counter (PC)
uint8_t  status = 0x00;    // Registro de estado

```

Para cada instrucción vamos a emular el *addressing mode* y la operación. Todo esto será programado individualmente con funciones. En total habrá doce *addressing modes*, como vimos en la tabla 16 × 16.

```
uint8_t IMP();    uint8_t IMM();    uint8_t ZP0();    uint8_t ZPX();
uint8_t ZPY();    uint8_t REL();    uint8_t ABS();    uint8_t ABX();
uint8_t ABY();    uint8_t IND();    uint8_t IZX();    uint8_t IZY();
```

Después vendrán los 56 *opcodes* que vimos en el *datasheet*.

```
uint8_t ADC();    uint8_t AND();    uint8_t ASL();    uint8_t BCC();
uint8_t BCS();    uint8_t BEQ();    uint8_t BIT();    uint8_t BMI();
uint8_t BNE();    uint8_t BPL();    uint8_t BRK();    uint8_t BVC();
uint8_t BVS();    uint8_t CLC();    uint8_t CLD();    uint8_t CLI();
uint8_t CLV();    uint8_t CMP();    uint8_t CPX();    uint8_t CPY();
uint8_t DEC();    uint8_t DEX();    uint8_t DEY();    uint8_t EOR();
uint8_t INC();    uint8_t INX();    uint8_t INY();    uint8_t JMP();
uint8_t JSR();    uint8_t LDA();    uint8_t LDX();    uint8_t LDY();
uint8_t LSR();    uint8_t NOP();    uint8_t ORA();    uint8_t PHA();
uint8_t PHP();    uint8_t PLA();    uint8_t PLP();    uint8_t ROL();
uint8_t ROR();    uint8_t RTI();    uint8_t RTS();    uint8_t SBC();
uint8_t SEC();    uint8_t SED();    uint8_t SEI();    uint8_t STA();
uint8_t STX();    uint8_t STY();    uint8_t TAX();    uint8_t TAY();
uint8_t TSX();    uint8_t TXA();    uint8_t TXS();    uint8_t TYA();

uint8_t XXX(); // Función para las instrucciones ilegales
```

También debemos tener en cuenta los ciclos del reloj. Por tanto, necesitamos una función que indique a la CPU que queremos que ocurra un ciclo.

```
void clock();
```

De igual manera, en la 6502 hay tres entradas más: el reseteo, la señal del interruptor estándar (que puede ser ignorado dependiendo de la señal) y un interruptor no mascarable (nunca debe de poder inhabilitarse). Estas tres entradas también deben de contar con su debida representación, y, junto al resto, pueden ocurrir en cualquier momento, no deben estar sincronizadas y deben poder parar el proceso que se esté ejecutando si terminara la instrucción.

```
void irq(); // Solicitud de interrupción (ejecuta una instrucción en una
            // localización específica)
void nmi(); // Solicitud de interrupción no-maskable (no puede ser deshabilitada)
void clock(); // Realiza un ciclo de reloj
```

Con esta última implementación nos acercamos al funcionamiento básico de la CPU, pero para su correcto funcionamiento precisaremos de algunas variables extras. Entre ellas, como tendremos que acceder y usar datos de la fuente correcta, deberemos de crear una variable que guarde dicha información.

```
uint8_t fetch();
uint8_t fetched = 0x00; // Represents the working input value to the ALU
```

Dependiendo del tipo de *addressing mode* querremos leer diferentes localizaciones de la memoria, así que guardaremos también esa localización en una variable.

```
uint16_t addr_abs = 0x0000; // All used memory addresses end up in here
```

Las instrucciones de *rama* en esta *CPU* solo pueden saltar cierta distancia desde la localización donde la instrucción fue llamada, saltando a una localización relativa. El *opcode* usado será guardado en `uint8_t opcode = 0x00;` y el número de ciclos que quedan por ejecutar en `uint8_t cycles = 0.`

Lo único que a estas alturas resta por añadir es la tabla de  $16 \times 16$ . Para ello crearemos un *struct* llamado *instruction* que guardará un mnemónico (*name*), una función que apunta al *address mode*, otra que apunta a la operación que debe ser realizada y un contador del número de ciclos.

```
struct INSTRUCTION
{
    std::string name;
    uint8_t      (olc6502::*operate )(void) = nullptr;
    uint8_t      (olc6502::*addrmode)(void) = nullptr;
    uint8_t      cycles = 0;
};

std::vector<INSTRUCTION> lookup; // Vector que guarda todo
```

Ahora debemos empezar con la implementación, comenzando por la tabla de  $16 \times 16$ .

```
using a = olc6502;
lookup =
{
    { "BRK", &a::BRK, &a::IMM, 7 }, { "ORA", &a::ORA, &a::IZX, 6 }, { "???", &a::XXX, &a::IMP, 2 }, { "???", &a::XXX, &a::IMP, 8 }, { "???", &a::NOP, &a::IMP, 3 }, { "ORA",
    { "BPL", &a::BPL, &a::REL, 2 }, { "ORA", &a::ORA, &a::IZY, 5 }, { "???", &a::XXX, &a::IMP, 2 }, { "???", &a::XXX, &a::IMP, 8 }, { "???", &a::NOP, &a::IMP, 4 }, { "ORA",
    { "JSR", &a::JSR, &a::ABS, 6 }, { "AND", &a::AND, &a::IZX, 6 }, { "???", &a::XXX, &a::IMP, 2 }, { "???", &a::XXX, &a::IMP, 8 }, { "BIT", &a::BIT, &a::ZP0, 3 }, { "AND",
    { "BMI", &a::BMI, &a::REL, 2 }, { "AND", &a::AND, &a::IZY, 5 }, { "???", &a::XXX, &a::IMP, 2 }, { "???", &a::XXX, &a::IMP, 8 }, { "???", &a::NOP, &a::IMP, 4 }, { "AND",
    { "RTI", &a::RTI, &a::IMP, 6 }, { "EOR", &a::EOR, &a::IZX, 6 }, { "???", &a::XXX, &a::IMP, 2 }, { "???", &a::XXX, &a::IMP, 8 }, { "???", &a::NOP, &a::IMP, 3 }, { "EOR",
    { "BVC", &a::BVC, &a::REL, 2 }, { "EOR", &a::EOR, &a::IZY, 5 }, { "???", &a::XXX, &a::IMP, 2 }, { "???", &a::XXX, &a::IMP, 8 }, { "???", &a::NOP, &a::IMP, 4 }, { "EOR",
    { "RTS", &a::RTS, &a::IMP, 6 }, { "ADC", &a::ADC, &a::IZX, 6 }, { "???", &a::XXX, &a::IMP, 2 }, { "???", &a::XXX, &a::IMP, 8 }, { "???", &a::NOP, &a::IMP, 3 }, { "ADC",
    { "BVS", &a::BVS, &a::REL, 2 }, { "ADC", &a::ADC, &a::IZY, 5 }, { "???", &a::XXX, &a::IMP, 2 }, { "???", &a::XXX, &a::IMP, 8 }, { "???", &a::NOP, &a::IMP, 4 }, { "ADC",
    { "???", &a::NOP, &a::IMP, 2 }, { "STA", &a::STA, &a::IZX, 6 }, { "???", &a::XXX, &a::IMP, 2 }, { "???", &a::XXX, &a::IMP, 6 }, { "STY", &a::STY, &a::ZP0, 3 }, { "STA",
    { "BCC", &a::BCC, &a::REL, 2 }, { "STA", &a::STA, &a::IZY, 6 }, { "???", &a::XXX, &a::IMP, 2 }, { "???", &a::XXX, &a::IMP, 6 }, { "STY", &a::STY, &a::ZPX, 4 }, { "STA",
    { "LDY", &a::LDY, &a::IMM, 2 }, { "LDA", &a::LDA, &a::IZX, 6 }, { "LDX", &a::LDX, &a::IMM, 2 }, { "???", &a::XXX, &a::IMP, 6 }, { "LDY", &a::LDY, &a::ZP0, 3 }, { "LDA",
    { "BCS", &a::BCS, &a::REL, 2 }, { "LDA", &a::LDA, &a::IZY, 5 }, { "???", &a::XXX, &a::IMP, 2 }, { "???", &a::XXX, &a::IMP, 5 }, { "LDY", &a::LDY, &a::ZPX, 4 }, { "LDA",
    { "CPY", &a::CPY, &a::IMM, 2 }, { "CMP", &a::CMP, &a::IZX, 6 }, { "???", &a::NOP, &a::IMP, 2 }, { "???", &a::XXX, &a::IMP, 8 }, { "CPY", &a::CPY, &a::ZP0, 3 }, { "CMP",
    { "BNE", &a::BNE, &a::REL, 2 }, { "CMP", &a::CMP, &a::IZY, 5 }, { "???", &a::XXX, &a::IMP, 2 }, { "???", &a::XXX, &a::IMP, 8 }, { "???", &a::NOP, &a::IMP, 4 }, { "CMP",
    { "CPX", &a::CPX, &a::IMM, 2 }, { "SBC", &a::SBC, &a::IZX, 6 }, { "???", &a::NOP, &a::IMP, 2 }, { "???", &a::XXX, &a::IMP, 8 }, { "CPX", &a::CPX, &a::ZP0, 3 }, { "SBC",
    { "BEQ", &a::BEQ, &a::REL, 2 }, { "SBC", &a::SBC, &a::IZY, 5 }, { "???", &a::XXX, &a::IMP, 2 }, { "???", &a::XXX, &a::IMP, 8 }, { "???", &a::NOP, &a::IMP, 4 }, { "SBC",
};
```

Como ya vimos, cada casilla tiene toda la información necesaria, que ya tiene su función correspondiente en el código. Del mismo modo que el resto, debemos crear una clase y una variable (`using a = olc6502;`). La función del reloj ejecutará la variable de los ciclos internos cuando sea igual a 0.

```

// Realiza un ciclo de reloj
void olc6502::clock()
{
    {
        opcode = read(pc);
#ifdef LOGMODE
        uint16_t log_pc = pc;
#endif
        // Asigna a 1 el bit flag de estado sin uso
        SetFlag(U, true);

        // Incrementa el program counter. Se lee el byte opcode
        pc++;

        // Comienza el número de ciclos
        cycles = lookup[opcode].cycles;

        // Realiza la extracción de los datos usando el modo de addressing
requerido      uint8_t additional_cycle1 = (this->*lookup[opcode].addrmode)();

        // Realiza la operación
        uint8_t additional_cycle2 = (this->*lookup[opcode].operate)();

        // ¿El addressmode o el opcode han alterado el número de ciclos que
requiere
        // la instrucción antes de su compleción?
        cycles += (additional_cycle1 & additional_cycle2);

        // Asigna a 1 el bit flag de estado sin uso
        SetFlag(U, true);
#ifdef LOGMODE
        if (logfile == nullptr)      logfile = fopen("olc6502.txt", "wt");
        if (logfile != nullptr)
        {
            fprintf(logfile, "%10d:%02d PC:%04X %s A:%02X X:%02X Y:%02X
%s%s%s%s%s%s%s STKP:%02X\n",
                    clock_count, 0, log_pc, "XXX", a, x, y,
                    GetFlag(N) ? "N" : ".",      GetFlag(V) ? "V" : ".",
GetFlag(U) ? "U" : ".",
                    GetFlag(B) ? "B" : ".",      GetFlag(D) ? "D" : ".",
GetFlag(I) ? "I" : ".",
                    GetFlag(Z) ? "Z" : ".",      GetFlag(C) ? "C" : ".",      stkp);
        }
#endif
    }

    // Incrementa el contador global, algo poco útil en la práctica
pero muy práctico para debugging
    clock_count++;

    // Decrementa el número de ciclos restantes para la instrucción
    cycles--;
}

```

Al coexistir una cantidad tal de *addressing modes* y tipos de instrucciones (12 y 56, respectivamente), nos centraremos únicamente en el funcionamiento detallado de algunas de ellas, aunque nuestro código de referencia tenga todas programadas. Cabe destacar que muchas son muy similares a otras, así que en la mayoría de estos casos explicaremos una como base y el resto tan solo serán mencionadas.

```
uint8_t olc6502::IMP()
{
    fetched = a;
    return 0;
}
```

Este *addressing mode* es implícito, por lo que no hay datos dentro de la instrucción. Implícito también quiere decir que podría estar ejecutándose encima del acumulador, en cuyo caso la variable `fetched` estaría contextualizada con sus contenidos.

```
uint8_t olc6502::IMM() // Immediate Mode Addressing
{
    addr_abs = pc++;
    return 0;
}
```

*Immediate Mode Addressing* significa que los datos son parte de la instrucción. Todos los *address mode* van a definir la variable absoluta, de tal modo que la instrucción sabe de dónde leer la información (en este caso, del siguiente *byte*).

```
uint8_t olc6502::ZP0() // Zero Page Addressing
{
    addr_abs = read(pc);
    pc++;
    addr_abs &= 0x00FF;
    return 0;
}
```

La siguiente es la *Zero Page Addressing*. Las *páginas* son formas conceptuales de guardar memoria. Sabemos que para cada *address memory* vamos a necesitar *16 bits*, y que cada una se puede dividir en 2 grupos de *4 bytes*. Los 4 primeros serán para la página, y los 4 últimos para el desplazamiento de la misma. Esto nos permite concluir que, en total, serían 256 páginas de 256 bytes. *Zero Page Addressing* quiere decir que el byte que estamos interesados en leer está en algún lugar de la página 0.

```
uint8_t olc6502::ZPX()
{
    addr_abs = (read(pc) + x);
    pc++;
    addr_abs &= 0x00FF;
    return 0;
}
```

El *addressing mode* descrito superiormente es parecido al anterior, con la salvedad de que esta vez los contenidos del registro *X* se agregan a la dirección única del *byte* proporcionada. Esto es útil para iterar a través rangos dentro de la primera página.

### Programación de instrucciones

Ahora que tenemos los *addressing modes* programados, podemos pasar a emular las instrucciones, pues ahora sabemos en qué espacio de la memoria están los datos que queremos. Como necesitamos buscarlo, completaremos la función `fetch`: queremos que funcione para todas menos para las implícitas, ya que estas no usan datos.

```
uint8_t olc6502::fetch()
{
    if (!(lookup[opcode].addrmode == &olc6502::IMP))
        fetched = read(addr_abs);
    return fetched;
}
```

Es hora de programar las instrucciones. La primera será una instrucción lógica *bit a bit*: *AND*. Esta será la base de la que partiremos para el resto de las instrucciones.

```
// Instrucción lógica AND
// Función:  A = A & M

uint8_t olc6502::AND()
{
    fetch();
    a = a & fetched;
    SetFlag(Z, a == 0x00);
    SetFlag(N, a & 0x80);
    return 1;
}
```

Lo primero que se hace es guardar los datos para ejecutar una lógica *bit a bit* entre el acumulador y los datos. Posteriormente se actualizará el registro de estado. Mirando la *datasheet* podemos apreciar que seguramente se precise de un ciclo extra.

La siguiente instrucción que explicaremos es la *BCS*, la cual sigue un esquema muy similar. En este caso es una *rama* si el *bit* del estado de registro está en la instrucción. Si el *bit* es igual a *1*, se configura la dirección absoluta para ser el contador del programa más el compensador (*offset*) relativo. Las instrucciones de *rama* son especiales, ya que cambiarán la variable de ciclos añadiendo en el inicio uno extra, y, en caso de cruzar una *página*, añadirá otro más. De nuevo, toda esta información puede encontrarse en el *datasheet*.

Con esta estructura podemos hacer muchas más instrucciones como la *BEQ* (*rama* = 0), *BMI* (si *rama* negativa), *BNE* (*rama* no es igual), etcétera; simplemente debemos mirar el *datasheet* y cambiar las variables.

```

// Instrucción l3gica Branch if Carry Set (Acarreo)
// Funci3n:  if(C == 1) pc = address

uint8_t olc6502::BCS()
{
    if (GetFlag(C) == 1)
    {
        cycles++;
        addr_abs = pc + addr_rel;

        if ((addr_abs & 0xFF00) != (pc & 0xFF00))
            cycles++;

        pc = addr_abs;
    }
    return 0;
}

```

Otra estructura de instrucci3n es la *CLC* (que limpia el acarreo). Tiene una estructura simple aplicable a otras como *CLD* (que limpia la se1al decimal).

```

// Instrucci3n l3gica Clear Carry Flag (CLC)
// Funci3n:  C = 0

uint8_t olc6502::CLC()
{
    SetFlag(C, false);
    return 0;
}

```

Las funciones m1s problem1ticas - y, a la vez, las m1s usadas - son *ADC* y *SBC*. La funci3n *ADC* (suma) se consigue a trav3s del siguiente c3digo:

```

uint8_t olc6502::ADC()
{
    // Extrae la informaci3n a a1adir al acumulador
    fetch();

    // Se a1ade en un dominio de 16-bit domain para capturar cualquier bit
    // de acarreo que pueda encontrarse en el bit 8
    temp = (uint16_t)a + (uint16_t)fetch + (uint16_t)GetFlag(C);

    // El flag de carry flag existe en el bit 0 del byte superior
    SetFlag(C, temp > 255);

    // El flag Zero se asigna si el resultado es 0
    SetFlag(Z, (temp & 0x00FF) == 0);

    // El flag Overflow se asigna en consecuencia
    SetFlag(V,  (~(uint16_t)a ^ (uint16_t)fetch) & ((uint16_t)a ^
(uint16_t)temp)) & 0x0080);
}

```

```

// El flag negativo se asigna al bit más significativo del resultado
SetFlag(N, temp & 0x80);

// Carga el resultado en el acumulador
a = temp & 0x00FF;

// Se puede requerir un ciclo de reloj adicional
return 1;
}
    
```

La base de esta instrucción es añadir al acumulador datos guardados desde la memoria y el bit acarreado. Aunque la operación parezca simple, el problema viene cuando los programadores quieren usar un número compuesto por 8 bits negativos, ya que pueden salirse del rango asignado (0-255) y provocar *overflow* o desbordamiento. Por ejemplo, el número 10000100 en caso de números positivos sería 132 (no hay desbordamiento), pero si son negativos entonces sería equivalente a -124, ya que el rango que queremos es de -128 a 127.

Para saber si un número es negativo, así, debemos mirar el primer bit. Si es igual a 1 entonces es negativo.

$$\begin{array}{r}
 10000100 = 132 \quad \text{or} \quad -124 \\
 +00010001 = 17 \\
 \hline
 10010101 = 149 \quad \checkmark \qquad \qquad \frac{17}{-107} \quad \checkmark
 \end{array}$$

Como podemos ver en el ejemplo, ambos son correctos y esto nos permite calcular números negativos aunque nuestro hardware solo trabaje con números positivos (no firmados). No obstante, se debe de tener en cuenta que si sumamos números - bien sean negativos o positivos - y salimos del rango asignado - en este caso, 127 - no podremos ejecutar el programa.

La V en el registro de estado representa el *overflow*, y nos indica si al usar números negativos hemos obtenido un resultado no coherente. Esto puede ser resuelto con una simple tabla de verdad que indique cuándo existe este problema:

A	MR	R	V
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Siguiendo tales indicaciones, tendremos que tener cuidado y programar para aquellos casos en los que  $V$  es igual a 0. Usando el acumulador y la memoria podemos obtener una versión mejorada de la tabla que indica correctamente cuándo hay *overflow* y su fórmula. Gracias a todo este análisis obtenemos la función descrita anteriormente.

A	MR	R	V	A^R	~(A^M)
0	0	0	0	0	1
0	0	1	1	1	1
0	1	0	0	0	0
0	1	1	0	1	0
1	0	0	0	1	0
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	0	1

$$V = (A^R) d \sim (A^M) \rightarrow \text{Overflow}$$

La instrucción *SBC* (resta) por suerte sigue una estructura similar a la anterior:

```
uint8_t olc6502::SBC()
{
    fetch();

    // Se opera en un dominio 16-bit para capturar cualquier acarreo

    // Invertimos los últimos 8 bits con xor
    uint16_t value = ((uint16_t) fetched) ^ 0x00FF;

    temp = (uint16_t)a + value + (uint16_t)GetFlag(C);
    SetFlag(C, temp & 0xFF00);
    SetFlag(Z, ((temp & 0x00FF) == 0));
    SetFlag(V, (temp ^ (uint16_t)a) & (temp ^ value) & 0x0080);
    SetFlag(N, temp & 0x0080);
    a = temp & 0x00FF;
    return 1;
}
```

Ahora que tenemos resueltas las instrucciones tanto simples como complejas, podemos programar las instrucciones de apilamiento (*stack*). Si bien en este estudio no nos explayaremos en su funcionamiento al completo, cabe destacar que al hacer uso de la memoria llaman a las funciones *write* y *read* descritas anteriormente, en pro de acceder al bus de manera correcta. Algunas de estas funciones son *PLA* y *PHA*.

Llega así el momento de centrarnos en los interruptores: *reset*, *irq* y *nmi*. Como las sentencias y los métodos que componen su código carecen de dificultad y destacan por su sencillez y claridad, simplemente las mostraremos a continuación.

```
void olc6502::reset()
{
    // Se escoge la dirección a la que asignar el program counter
    addr_abs = 0xFFFFC;
    uint16_t lo = read(addr_abs + 0);
    uint16_t hi = read(addr_abs + 1);

    // Se asigna
    pc = (hi << 8) | lo;

    // Se reinician los registros internos
    a = 0;
    x = 0;
    y = 0;
    stkp = 0xFD;
    status = 0x00 | U;

    // Se reinician las variables internas auxiliares
    addr_rel = 0x0000;
    addr_abs = 0x0000;
    fetched = 0x00;

    // Se reinicia el tiempo
    cycles = 8;
}

void olc6502::irq()
{
    // Si las interrupciones están habilitadas
    if (GetFlag(I) == 0)
    {
        // Se empuja el program counter a la pila
        // El proceso se realiza dos veces al trabajar con 16 bits
        write(0x0100 + stkp, (pc >> 8) & 0x00FF);
        stkp--;
        write(0x0100 + stkp, pc & 0x00FF);
        stkp--;

        // Se empuja el registro de estado a la pila
        SetFlag(B, 0);
        SetFlag(U, 1);
        SetFlag(I, 1);
        write(0x0100 + stkp, status);
        stkp--;

        // Se lee la nueva localización del program counter desde la nueva
        // dirección
        addr_abs = 0xFFFFE;
        uint16_t lo = read(addr_abs + 0);
        uint16_t hi = read(addr_abs + 1);
        pc = (hi << 8) | lo;
        // Tiempo de IRQs
        cycles = 7;
    }
}
```

```

void olc6502::nmi()
{
    write(0x0100 + stkp, (pc >> 8) & 0x00FF);
    stkp--;
    write(0x0100 + stkp, pc & 0x00FF);
    stkp--;

    SetFlag(B, 0);
    SetFlag(U, 1);
    SetFlag(I, 1);
    write(0x0100 + stkp, status);
    stkp--;

    addr_abs = 0xFFFFA;
    uint16_t lo = read(addr_abs + 0);
    uint16_t hi = read(addr_abs + 1);
    pc = (hi << 8) | lo;

    cycles = 8;
}

```

Con todo el código necesario listo, únicamente necesitaremos utilizar función extra para hacerlo funcionar. Esta será *disassemble*, la cual compilará todo el código de manera correcta y emulará el programa deseado.

### Lógica de NES: buses, RAMs, ROMs y mappers

Antes de continuar con la programación y de adentrarnos en el funcionamiento exacto de la *PPU* (recordamos, *Picture Processing Unit*), conviene realizar un nuevo parón teórico para acercarse a la lógica del sistema, y así poder realizar con tino y entendimiento los cálculos, interacciones y métodos siguientes.

Con este punto de partida, comenzaremos desmintiendo algo que hasta ahora habíamos tomado como certero: mientras que hasta ahora hemos estado trabajando con una *RAM* de *64 Kb* (que configuraba todo el rango de direcciones), *Nintendo Entertainment System* contaba, realmente, con *2 Kb* de memoria *RAM*, la cual era accesible desde *0x0000* hasta *0x1FFF*, un rango de *8 Kb*. ¿Cómo podía, entonces, estar una memoria de *2 Kb* asignada a un rango de *8 Kb*? A través del *mirroring*, una técnica que implementaba la máquina, y que utilizaremos considerablemente en los próximos capítulos de este estudio.

El principio subyacente del *mirroring* es que los rangos de direcciones se repiten, por lo que, aunque el rango direccionable de la *RAM* sea de *0* a *8 Kb*, solo los primeros *2 Kb* se asignan al *hardware*, y cada segmento de *2 Kb* restante se asigna a dicha sección. Por tanto, si se escribe en la primera sección, conceptualmente se escribirá también en todas las demás, al limitarse estas a replicar (o, más bien, a redirigir) el resultado de la primera.

En la práctica, el funcionamiento del *mirroring* es considerablemente sencillo. Supongamos que queremos escribir en la dirección `0x1234` (en decimal, 4660), lo cual escapa del rango máximo de 2 Kb.

A través de una sencilla operación *bit a bit* (concretamente, a través de un *AND*) con el valor que realmente refleja el máximo rango direccionable (en este caso, `0x7FF`; en decimal, 2047), podemos obtener una nueva dirección, `0x0234` (en decimal, 564), que encaja coherentemente en el rango máximo.

Por otra parte, desde las primeras páginas de este estudio hemos señalado la existencia de un dispositivo adicional conectado a la *CPU*, el cual no es otro que la *PPU*, con su propio bus de memoria (y con un rango direccionable mucho menor, de paso sea dicho). Los primeros 8 Kb de ese *addressable range* (desde `0x0000` hasta `0x1FFF`) se asignan a la *pattern memory* o memoria de patrones, encargada de almacenar el aspecto de los *sprites*. Los 2 Kb siguientes se dedican a una *RAM* adicional de 2 Kb (desde `0x2000` hasta `0x2FFF`) que aloja los *name tables*; *arrays* bidimensionales que almacenan los IDs de los patrones que se deben de mostrar en el fondo o *background*. Por último, existe una *RAM* más (desde `0x3F00` hasta `0x3FFF`), que almacena la información relativa a las *palettes* o paletas, dictando qué colores deberán de mostrarse en pantalla al combinar los *sprites* y el *background*. Los datos relativos al programa (es decir, el programa en sí, pero también la información requerida para el renderizado del mismo), tal y como comentamos al inicio, se alojan junto a las tablas de los patrones en el propio cartucho, que ocupa prácticamente la segunda mitad del rango direccionable del bus de la *CPU*. No obstante, por cuestiones relativas a la simplicidad, asumiremos que las susodichas tablas se almacenan en la *NES*.

Existe un dispositivo adicional cuya existencia merece la pena recordar a estas alturas, y no es otro que el *mapper*, que ocupa su lugar en el mismo rango que la *ROM* del programa (solo de lectura) en el bus de la *CPU*, permitiendo la escritura en dicho rango. Y dada la capacidad del bus para discernir entre lecturas y escrituras, podemos dividir los intentos de la *CPU* consecuentemente, gestionando tanto sus lecturas como sus configuraciones en el circuito de *mapping*.

Recordadas estas bases, y partiendo del código detallado previamente, nos volveremos a introducir en la programación con la creación de dos nuevas clases, una que representará la *PPU* (que llamaremos `olc2C02`, dada la designación del *chip*) y otra que hará lo propio con el cartucho (que llamaremos *Cartridge*). Asimismo, antes de ensuciarnos las manos convendrá tener en cuenta que a partir de este momento contaremos con diversos dispositivos que potencialmente podrán realizar lecturas y escrituras, por lo que convendrá diferenciar las lecturas/escrituras de la *CPU* y de la *PPU*. Consecuentemente, modificaremos el nombre de nuestros métodos *write* y *read* del bus a `cpuWrite` y `cpuRead`, respectivamente. Haremos lo propio con la *RAM* de dicho bus, que pasará a llamarse `cpuRam`; también aprovecharemos para cambiar su tamaño a 2 Kb. Realizados estos cambios, conectaremos la *PPU* al bus declarando `olc2C02 ppu;` y añadiendo la variable temporal `uint8_t data = 0x00;` a la función `cpuRead`, y reajustaremos su rango a un valor más adecuado (`addr >= 0x0000 && addr <= 0x1FFF`).

Comenzaremos el proceso de *mirroring*, modificando así la condición del método `cpuRead` con la inclusión de una *AND* (asignaremos `cpuRam[addr & 0x07FF]` a `data`), y realizaremos los cambios oportunos en `cpuWrite`. Asimismo, estrenaremos nuestra clase de la *PPU* con los siguientes métodos:

```
// Conectan la PPU al bus de la CPU

uint8_t cpuRead(uint16_t addr, bool ronly = false);
void    cpuWrite(uint16_t addr, uint8_t data);

// Conectan la PPU a su propio bus

uint8_t ppuRead(uint16_t addr, bool ronly = false);
void    ppuWrite(uint16_t addr, uint8_t data);
```

El principal bus del sistema es el *target* de todas las escrituras de la *CPU*. En nuestro método `cpuWrite`, comprobamos que la escritura apuntase a la *RAM*, pero no hicimos mención a otros posibles rangos direccionales como el de la *PPU*. Realizaremos ahora las modificaciones oportunas, también aplicables a `cpuRead`:

```
void Bus::cpuWrite(uint16_t addr, uint8_t data)
{
    if (addr >= 0x0000 && addr <= 0x1FFF) // RAM
    {
        cpuRam[addr & 0x07FF] = data;
    }
    else if (addr >= 0x2000 && addr <= 0x3FFF) // PPU
    {
        ppu.cpuWrite(addr & 0x0007, data); // Utilizamos mirroring
    }
}

uint8_t Bus::cpuRead(uint16_t addr, bool bReadOnly)
{
    uint8_t data = 0x00;
    if (addr >= 0x0000 && addr <= 0x1FFF) // RAM
    {
        data = cpuRam[addr & 0x07FF];
    }
    else if (addr >= 0x2000 && addr <= 0x3FFF) // PPU
    {
        data = ppu.cpuRead(addr & 0x0007, bReadOnly);
    }

    return data;
}
```

De manera similar a la *PPU*, estrenaremos nuestra clase *Cartridge* con cuatro nuevos métodos, los cuales son necesarios al estar conceptualmente conectado el cartucho a ambos buses.

```
// Conectan el cartucho al bus de la CPU

bool cpuRead(uint16_t addr, uint8_t &data);
bool cpuWrite(uint16_t addr, uint8_t data);

// Conectan el cartucho al bus de la PPU

bool ppuRead(uint16_t addr, uint8_t &data);
bool ppuWrite(uint16_t addr, uint8_t data);
```

Tras este paso, conviene volver a las definiciones de las funciones de la *PPU* con el objetivo de describir su funcionamiento básico de manera provisional. Crearemos, para ello, una variable temporal *data*, a la que en el método `cpuRead` se le asignará una de las ocho posibles direcciones devueltas por la *CPU*. Al conocer los posibles resultados de antemano, estos podrán ser tratados a través de un `switch`, que preferiremos utilizar antes que simples variables al presentar un funcionamiento más cercano al de la transacción real; los datos leídos por la *PPU* en una ubicación no siempre se corresponden con el último valor guardado en ella.

Realizaremos el mismo razonamiento para el método `cpuWrite`, y como a lo largo de esta sección no estudiaremos a fondo el comportamiento de la *PPU*, utilizaremos algunos marcadores simples para rellenar sus debidas funciones, a las que volveremos más tarde.

```
uint8_t olc2C02::cpuRead(uint16_t addr, bool rdonly)
{
    uint8_t data = 0x00;

    switch (addr)
    {
        case 0x0000: // Control
            break;
        case 0x0001: // Mask
            mask.reg = data;
            break;
        case 0x0002: // Estado
            break;
        case 0x0003: // Dirección de OAM
            break;
        case 0x0004: // Datos de OAM
            break;
        case 0x0005: // Scroll
            break;
        case 0x0006: // Dirección de PPU
            break;
        case 0x0007: // Datos de PPU
            break;
    }

    return data;
}
```

```

void olc2C02::cpuWrite(uint16_t addr, uint8_t data)
{
    switch (addr)
    {
        case 0x0000: // Control
            break;
        case 0x0001: // Mask
            mask.reg = data;
            break;
        case 0x0002: // Estado
            break;
        case 0x0003: // Dirección de OAM
            break;
        case 0x0004: // Datos de OAM
            break;
        case 0x0005: // Scroll
            break;
        case 0x0006: // Dirección de PPU
            break;
        case 0x0007: // Datos de PPU
            break;
    }
}

uint8_t olc2C02::ppuRead(uint16_t addr, bool ronly)
{
    uint8_t data = 0x00;
    addr &= 0x3FFF;

    return data;
}

void olc2C02::ppuWrite(uint16_t addr, uint8_t data)
{
    addr &= 0x3FFF; // Masking en caso de que la PPU trate de escribir
                   // en una dirección de su bus fuera del rango direccional
}

```

En un intento por asemejar nuestra clase principal Bus al núcleo de la emulación, siendo la clase que representará en gran medida el funcionamiento de NES, aprovecharemos esta serie de pequeños ajustes para realizar algunas importantes modificaciones en ella, incluyendo las tres formas de interacción existentes entre la máquina y el exterior: el cartucho (representado a través de una función a la que se le pasa un objeto de tipo *Cartridge*), el botón de reinicio y el reloj.

```

#include "Cartridge.h"

class Bus
{
    [...]

public: // Interfaz del sistema
    void insertCartridge(const std::shared_ptr<Cartridge>& cartridge);
    void reset();
    void clock();
}

```

```
private:
    uint32_t nSystemClockCounter = 0; // Guarda el nº de llamadas a clock()
};
```

```
void Bus::reset()
{
    cpu.reset();
    nSystemClockCounter = 0;
}

void Bus::clock()
{
    nSystemClockCounter++;
}
```

La *PPU* también cuenta con acceso al cartucho, por lo que almacenaremos en ella un puntero privado a un objeto de tipo *Cartridge*. Crearemos, de la misma manera, una pequeña interfaz que permita realizar la conexión entre el cartucho y la *PPU*.

```
class olc2C02
{
    [...]

private:
    std::shared_ptr<Cartridge> cart;

public:
    // Interfaz
    void ConnectCartridge(const std::shared_ptr<Cartridge>& cartridge);
    void clock();
};
```

Como el cartucho no es, en resumidas cuentas, más que otro dispositivo conectado al bus principal, realizaremos tal implementación para que lo descrito hasta ahora tenga sentido.

```
class Bus
{
public:
    olc6502 cpu;
    olc2C02 ppu;
    std::array<uint8_t, 2048> cpuRam;
    std::shared_ptr<Cartridge> cart;

    [...]
};
```

```

void Bus::insertCartridge(const std::shared_ptr<Cartridge>& cartridge)
{
    this->cart = cartridge;
    ppu.ConnectCartridge(cartridge); // La definimos a continuación
}

void olc2C02::ConnectCartridge(const std::shared_ptr<Cartridge>& cartridge)
{
    this->cart = cartridge; // Asigna la variable local del cartucho al
    puntero
}

```

Y así como la clase del bus precisa de dispositivos conectados a ella, la *PPU* también necesita de estos; concretamente, de memorias.

```

class olc2C02
{
private:
    uint8_t      tblName[2][1024]; // VRAM; almacena la información de las
    tablas
    uint8_t      tblPalette[32]; // RAM; almacena la información de paletas

    [...]
};

```

### La clase *Cartridge*

Una vez hechos estos reajustes, podemos volver al campo teórico para poner sobre la mesa nuevas ideas que nos ayudarán con las posteriores inclusiones del código (especialmente, con aquellas relativas a la clase *Cartridge*). Y es que, en primer lugar, cabe señalar que la mayoría de los pines del cartucho son simplemente los pines de datos y de dirección tanto para la *CPU* como para la *PPU*, lo que nos permite asumir que existen algunas conexiones directas tanto a la memoria del programa (*program memory*, o *PRG*) como a la memoria de patrones (*pattern memory*, o *CHR*). No obstante, el cartucho es un elemento considerablemente más complejo que un par de conexiones; en él pueden coexistir diversas memorias con sus diferentes chips de memoria, o bien una gran memoria (usualmente dividida en *banks*) a cuyas secciones se haga referencia a través de un *mapper* configurado por la *CPU* que recoja las direcciones entrantes (*incomes*) de ambos buses y las transforme en una posición de memoria válida del cartucho. A modo de curiosidad, este factor fue el que posibilitó la inexistencia de tiempos de carga en las consolas antiguas, pues su sistema no debía de cargar nada: solo tenía que mapear la *address line* correcta al lugar indicado.

Llega el turno, entonces, de añadir las correspondientes memorias a la clase *Cartridge*, las cuales serán representadas a través de vectores (pues no sabremos cuán grandes necesitarán ser hasta que procedamos a la lectura del archivo). También añadiremos en el constructor de la clase un *string* que represente la dirección (*path*) del fichero a ejecutar, y fuera de este tres variables privadas que reflejen el *mapper*

utilizado, cuántos *banks* de la primera memoria existen, y cuántos de ellos hay en la segunda memoria.

```
class Cartridge
{
public:
    Cartridge(const std::string& sFileName);
    ~Cartridge();

private:

    uint8_t nMapperID = 0;
    uint8_t nPRGBanks = 0;
    uint8_t nCHRBanks = 0;

    std::vector<uint8_t> vPRGMemory;
    std::vector<uint8_t> vCHRMemory;

    std::shared_ptr<Mapper> pMapper;

    [...]

};
```

```
Cartridge::Cartridge(const std::string& sFileName)
{
    // Creación de una estructura Header
    struct sHeader
    {
        char name[4];
        uint8_t prg_rom_chunks;
        uint8_t chr_rom_chunks;
        uint8_t mapper1;
        uint8_t mapper2;
        uint8_t prg_ram_size;
        uint8_t tv_system1;
        uint8_t tv_system2;
        char unused[5];
    } header;

    std::ifstream ifs;
    ifs.open(sFileName, std::ifstream::binary); // Se abre el fichero en
    binario
    if (ifs.is_open()) // ... e inmediatamente se lee a través del Header
    {
        ifs.read((char*)&header, sizeof(sHeader));

        if (header.mapper1 & 0x04) // 512 bytes para training; se ignora
            ifs.seekg(512, std::ios_base::cur);

        // A través del Header extraemos el mapper usado por la ROM
        nMapperID = ((header.mapper2 >> 4) << 4) | (header.mapper1 >> 4);

        // Averiguamos el tipo de archivo de la ROM (tres tipos posibles)
        uint8_t nFileType = 1;
```

```

        if (nFileType == 0) // Lo ampliaremos más adelante
        {

        }

        if (nFileType == 1)
        {
            // Leemos (y asignamos) cuántos banks hay en la ROM
            nPRGBanks = header.prg_rom_chunks;
            // Readaptamos el tamaño del vector a dicho tamaño
            vPRGMemory.resize(nPRGBanks * 16384);
            // Leemos (y asignamos al vector) los datos del fichero
            ifs.read((char*)vPRGMemory.data(), vPRGMemory.size());

            // Realizamos el mismo proceso con CHR
            nCHRBanks = header.chr_rom_chunks;
            vCHRMemory.resize(nCHRBanks * 8192);
            ifs.read((char*)vCHRMemory.data(), vCHRMemory.size());

        }

        if (nFileType == 2) // Lo ampliaremos más adelante
        {

        }

        ifs.close(); // Cerramos el fichero
    }
}

bool Cartridge::cpuRead(uint16_t addr, uint8_t &data)
{
    return false;
}

bool Cartridge::cpuWrite(uint16_t addr, uint8_t data)
{
    return false;
}

bool Cartridge::ppuRead(uint16_t addr, uint8_t & data)
{
    return false;
}

bool Cartridge::ppuWrite(uint16_t addr, uint8_t data)
{
    return false;
}

```

Los últimos métodos de este código - que ya habían sido implementados previamente - pueden ser especialmente llamativos si tenemos en cuenta que devuelven un booleano en lugar de, por ejemplo, el valor extraído de la lectura; este se encarga de comunicar al sistema si el cartucho está gestionando esa lectura o escritura.

Ya que hablamos de esta clase de funciones, es buen momento para volver al código del bus principal, y concretamente, a su método `cpuWrite`, el cual actualmente asume

que nada interfiere con la dirección de memoria recibida como parámetro. No obstante, conviene estudiar antes de llevar a cabo la asignación si el cartucho puede gestionar la escritura, en cuyo caso se devolverá `true` y la *RAM* no se actualizará (o no se escribirá en *PPU*).

La introducción de un condicional como este permitirá al cartucho ser consciente de lo que está ocurriendo en el bus principal. Cabe esperar que en la amplia mayoría de los casos tales procedimientos le sean irrelevantes, pues, al fin y al cabo, ¿por qué querría el cartucho interferir en la lectura y escritura de la *CPU* en su *RAM*? No obstante, esta implementación nos permitirá dar lugar más adelante a una extensión de nuestro emulador, a través de la cual con un único cambio en el circuito de *mapping* del cartucho podremos cambiar el comportamiento del emulador. Asimismo, su inclusión permite la previsión de errores, ya que si por alguna razón la *CPU* provocase una interacción relativa al cartucho, tal sentencia se limitaría a devolver `true`, y no brindaría un resultado inesperado ante el resto de comprobaciones. Tal y como hemos realizado con anterioridad, aplicaremos esta lógica también a `cpuRead`, y consecuentemente, de manera muy similar, a nuestra clase *PPU*.

```
void Bus::cpuWrite(uint16_t addr, uint8_t data)
{
    if (cart->cpuWrite(addr, data))
    {}
    else if (addr >= 0x0000 && addr <= 0x1FFF)
    {
        cpuRam[addr & 0x07FF] = data;
    }
    else if (addr >= 0x2000 && addr <= 0x3FFF)
    {
        ppu.cpuWrite(addr & 0x0007, data);
    }
}

uint8_t Bus::cpuRead(uint16_t addr, bool bReadOnly)
{
    uint8_t data = 0x00;
    if (cart->cpuRead(addr, data))
    {}
    else if (addr >= 0x0000 && addr <= 0x1FFF)
    {
        // System RAM Address Range, mirrored every 2048
        data = cpuRam[addr & 0x07FF];
    }
    else if (addr >= 0x2000 && addr <= 0x3FFF)
    {
        // PPU Address range, mirrored every 8
        data = ppu.cpuRead(addr & 0x0007, bReadOnly);
    }
    return data;
}

uint8_t olc2C02::ppuRead(uint16_t addr, bool rdonly)
{
    uint8_t data = 0x00;
    addr &= 0x3FFF;
```

```

        if (cart->ppuRead(addr, data))
        {}

        return data;
    }
void olc2C02::ppuWrite(uint16_t addr, uint8_t data)
{
    addr &= 0x3FFF;

    if (cart->ppuWrite(addr, data))
    {}
}

```

Todos estos avances en nuestro código, a estas alturas, parecen suficientes para implementar finalmente esa nueva clase que durante tantas páginas anteriores hemos adelantado: la clase *Mapper*. En su constructor, asignaremos el número de *PRG banks* y de *CHR banks*, los cuales asignaremos localmente. Posteriormente, añadiremos cuatro funciones de tipo *virtual* - dada la naturaleza abstracta de la clase - que recibirán direcciones de entrada del bus de la *CPU* o *PPU* y las transformarán en nuevas direcciones a través de las cuales podremos indexar las *ROMs* tal y como las hemos leído. Si la dirección es mapeada satisfactoriamente, se devolverá *true*.

```

class Mapper
{
public:
    Mapper(uint8_t prgBanks, uint8_t chrBanks);
    ~Mapper();

public:
    // Transforman la dirección del bus de CPU en PRG ROM offset
    virtual bool cpuMapRead(uint16_t addr, uint32_t &mapped_addr) = 0;
    virtual bool cpuMapWrite(uint16_t addr, uint32_t &mapped_addr) = 0;

    // Transforman la dirección del bus de PPU en CHR ROM offset
    virtual bool ppuMapRead(uint16_t addr, uint32_t &mapped_addr) = 0;
    virtual bool ppuMapWrite(uint16_t addr, uint32_t &mapped_addr) = 0;

protected:
    uint8_t nPRGBanks = 0;
    uint8_t nCHRBanks = 0;
};

```

```

#include "Mapper.h"

Mapper::Mapper(uint8_t prgBanks, uint8_t chrBanks)
{
    nPRGBanks = prgBanks;
    nCHRBanks = chrBanks;
}

```

```
Mapper::~Mapper() // No hay nada que añadir al ser una clase abstracta
{
}
```

Con la creación de esta clase, tenemos una plantilla perfecta sobre la que comenzar a erigir los diferentes tipos de *mappers*, los cuales podrán regirse por sus propias reglas y variables privadas y específicas (en una utilización explícita de programación orientada a objetos, la cual nos permite encapsular funcionalidades). El primero al que daremos forma, que llamaremos *Mapper\_000*, quedará definido por los siguientes métodos heredados:

```
class Mapper_000 : public Mapper
{
public:
    Mapper_000(uint8_t prgBanks, uint8_t chrBanks);
    ~Mapper_000();

public:
    bool cpuMapRead(uint16_t addr, uint32_t &mapped_addr) override;
    bool cpuMapWrite(uint16_t addr, uint32_t &mapped_addr) override;
    bool ppuMapRead(uint16_t addr, uint32_t &mapped_addr) override;
    bool ppuMapWrite(uint16_t addr, uint32_t &mapped_addr) override;
};
```

```
#include "Mapper_000.h"

Mapper_000::Mapper_000(uint8_t prgBanks, uint8_t chrBanks) : Mapper(prgBanks,
chrBanks)
{
}

Mapper_000::~Mapper_000()
{
}

bool Mapper_000::cpuMapRead(uint16_t addr, uint32_t &mapped_addr)
{
    if (addr >= 0x8000 && addr <= 0xFFFF)
    {
        // nPRGBanks almacena el número de bloques de 16 Kb cargados como
        PRG // ROM; si hay más de uno, entonces estaremos trabajando con una ROM
        // de 32 Kb (los cuales se mapearían exactamente con la mitad del
        // rango. Por ello, se realiza masking de la dirección.
        mapped_addr = addr & (nPRGBanks > 1 ? 0x7FFF : 0x3FFF);
        // Si la ROM es de 16 Kb, simplemente se realiza mirroring.
        return true;
    }
    return false;
}

bool Mapper_000::cpuMapWrite(uint16_t addr, uint32_t &mapped_addr)
{
    if (addr >= 0x8000 && addr <= 0xFFFF)
    {
```

```

        mapped_addr = addr & (nPRGBanks > 1 ? 0x7FFF : 0x3FFF);
        return true;
    }
    return false;
}
bool Mapper_000::ppuMapRead(uint16_t addr, uint32_t &mapped_addr)
{
    if (addr >= 0x0000 && addr <= 0x1FFF)
    {
        // No hay bank switching.
        mapped_addr = addr;
        return true;
    }

    return false;
}

bool Mapper_000::ppuMapWrite(uint16_t addr, uint32_t &mapped_addr)
{
    // En este punto del estudio no tiene sentido; solo devolveremos falso
    return false;
}

```

Ahora que nos acercamos al *quid* de la emulación, empieza a ser coherente la idea de intentar gestionar una lectura completa de *CPU*. Para ello, ya que el cartucho está capacitado para interceptar todas las lecturas y escrituras que se realicen en ambos buses, declararemos el *Mapper\_000* en él, agregando también un puntero a la clase abstracta *Mapper* y procediendo a cargar el *mapper* apropiado (a través de su *ID*) tras la carga de los vectores que representan las memorias *CHR*.

```

#include "Mapper_000.h"

class Cartridge
{
    [...]
private:
    std::shared_ptr<Mapper> pMapper;
    [...]
};

```

```

Cartridge::Cartridge(const std::string& sFileName)
{
    [...]
    if (ifs.is_open())
    {
        [...]
        if (nFileType == 2)
        {
            switch (nMapperID)
            {
                case 0: pMapper = std::make_shared<Mapper_000>(nPRGBanks,
nCHRBanks); break;

```

```

    }

    ifs.close();
}
}

```

Este procedimiento, en cierta manera, nos obliga a completar las cuatro funciones de lectura y escritura del cartucho que declaramos algunas páginas atrás, y que igualmente contarán con el derecho de vetar cualquier operación que suceda en alguno de los dos buses.

```

bool Cartridge::cpuRead(uint16_t addr, uint8_t &data)
{
    uint32_t mapped_addr = 0; // Variable temporal para la dirección transformada
    if (pMapper->cpuMapRead(addr, mapped_addr))
        // Si la rutina del mapper para el bus precisa de información del cartucho...
        {
            data = vPRGMemory[mapped_addr]; // Se asigna el offset a la variable
            return true; // El cartucho ha gestionado la dirección
        }
    else
        return false; // El cartucho no está interesado en la gestión
}

bool Cartridge::cpuWrite(uint16_t addr, uint8_t data)
{
    uint32_t mapped_addr = 0;
    if (pMapper->cpuMapWrite(addr, mapped_addr))
        {
            vPRGMemory[mapped_addr] = data;
            return true;
        }
    else
        return false;
}

bool Cartridge::ppuRead(uint16_t addr, uint8_t & data)
{
    uint32_t mapped_addr = 0;
    if (pMapper->ppuMapRead(addr, mapped_addr))
        {
            data = vCHRMemory[mapped_addr];
            return true;
        }
    else
        return false;
}

bool Cartridge::ppuWrite(uint16_t addr, uint8_t data)
{
    uint32_t mapped_addr = 0;
    if (pMapper->ppuMapRead(addr, mapped_addr))
        {
            vCHRMemory[mapped_addr] = data; // Se gestiona la memoria CHR, no
PRG

```

```

        return true;
    }
    else
        return false;
}

```

Con tal de observar de primera mano cómo esta estructura es capaz de instanciarse y de comunicarse entre sí, conviene realizar pruebas de funcionamiento y comenzar con la implementación de inputs, lo que nos ofrecerá un feedback visual directo.

Como parte de ello, concluimos esta sección realizando pequeños ajustes en la clase *olc2C02*, que nos servirán, de paso, para introducir a la *PPU* como protagonista del próximo capítulo. El primero de estos se basará en la inclusión de un *array* que almacenará todos y cada uno de los colores que *NES* es capaz de mostrar por pantalla; proseguiremos con nuevas funciones y variables.

```

class olc2C02
{
    [...]

private:
    olc::Pixel palScreen[0x40]; // Paleta de colores en uso
    olc::Sprite sprScreen = olc::Sprite(256, 240); // Salida por pantalla
    olc::Sprite sprNameTable[2] = {olc::Sprite(256, 240), olc::Sprite(256,
240)};
    // Descripción gráfica de las name tables
    olc::Sprite sprPatternTable[2] =
{olc::Sprite(128,128),olc::Sprite(128,128)};
    // Descripción gráfica de las pattern tables

public:
    // Declaramos funciones que nos devuelvan una referencia a estos sprites
    olc::Sprite& GetScreen();
    olc::Sprite& GetNameTable(uint8_t i);
    olc::Sprite& GetPatternTable(uint8_t i);
    bool frame_complete = false; // Representa si un frame está completo o no

private:
    int16_t scanline = 0; // Fila en pantalla
    int16_t cycle = 0; // Columna en pantalla

public:
    [...]
};

```

A continuación, en el cuerpo de la clase detallaremos las funciones declaradas, de comportamiento trivial, así como el método `clock`, hasta ahora vacío, que jamás se detendrá y que determinará la velocidad de dibujo de los píxeles en pantalla. Este precisará de una lista de valores RGB que representen las entradas de las paletas.

```
olc::Sprite& olc2C02::GetScreen()
{
    return sprScreen;
}

olc::Sprite & olc2C02::GetNameTable(uint8_t i)
{
    return sprNameTable[i];
}

olc::Sprite & olc2C02::GetPatternTable(uint8_t i)
{
    return sprPatternTable[i];
}

void olc2C02::clock()
{
    sprScreen.SetPixel(cycle - 1, scanline, palScreen[(rand() % 2) ? 0x3F :
0x30]); // Usando la paleta, dibujamos un píxel en cada ubicación de la pantalla
        // De momento únicamente mostrará una especie de ruido provisional
    cycle++;
    if (cycle >= 341) // Reseteamos el cycle
    {
        cycle = 0;
        scanline++;
        if (scanline >= 261) // Reseteamos scanline y completamos frame
        {
            scanline = -1;
            frame_complete = true;
        }
    }
}

olc2C02::olc2C02() // Lista de valores RGB
{
    palScreen[0x00] = olc::Pixel(84, 84, 84);
    palScreen[0x01] = olc::Pixel(0, 30, 116);
    palScreen[0x02] = olc::Pixel(8, 16, 144);
    palScreen[0x03] = olc::Pixel(48, 0, 136);
    palScreen[0x04] = olc::Pixel(68, 0, 100);
    palScreen[0x05] = olc::Pixel(92, 0, 48);
    palScreen[0x06] = olc::Pixel(84, 4, 0);
    palScreen[0x07] = olc::Pixel(60, 24, 0);
    palScreen[0x08] = olc::Pixel(32, 42, 0);
    palScreen[0x09] = olc::Pixel(8, 58, 0);
    palScreen[0x0A] = olc::Pixel(0, 64, 0);
    palScreen[0x0B] = olc::Pixel(0, 60, 0);
    palScreen[0x0C] = olc::Pixel(0, 50, 60);
    palScreen[0x0D] = olc::Pixel(0, 0, 0);
    palScreen[0x0E] = olc::Pixel(0, 0, 0);
    palScreen[0x0F] = olc::Pixel(0, 0, 0);

    palScreen[0x10] = olc::Pixel(152, 150, 152);
    palScreen[0x11] = olc::Pixel(8, 76, 196);
    palScreen[0x12] = olc::Pixel(48, 50, 236);
    palScreen[0x13] = olc::Pixel(92, 30, 228);
    palScreen[0x14] = olc::Pixel(136, 20, 176);
    palScreen[0x15] = olc::Pixel(160, 20, 100);
}
```

```

palScreen[0x16] = olc::Pixel(152, 34, 32);
palScreen[0x17] = olc::Pixel(120, 60, 0);
palScreen[0x18] = olc::Pixel(84, 90, 0);
palScreen[0x19] = olc::Pixel(40, 114, 0);
palScreen[0x1A] = olc::Pixel(8, 124, 0);
palScreen[0x1B] = olc::Pixel(0, 118, 40);
palScreen[0x1C] = olc::Pixel(0, 102, 120);
palScreen[0x1D] = olc::Pixel(0, 0, 0);
palScreen[0x1E] = olc::Pixel(0, 0, 0);
palScreen[0x1F] = olc::Pixel(0, 0, 0);
palScreen[0x20] = olc::Pixel(236, 238, 236);
palScreen[0x21] = olc::Pixel(76, 154, 236);
palScreen[0x22] = olc::Pixel(120, 124, 236);
palScreen[0x23] = olc::Pixel(176, 98, 236);
palScreen[0x24] = olc::Pixel(228, 84, 236);
palScreen[0x25] = olc::Pixel(236, 88, 180);
palScreen[0x26] = olc::Pixel(236, 106, 100);
palScreen[0x27] = olc::Pixel(212, 136, 32);
palScreen[0x28] = olc::Pixel(160, 170, 0);
palScreen[0x29] = olc::Pixel(116, 196, 0);
palScreen[0x2A] = olc::Pixel(76, 208, 32);
palScreen[0x2B] = olc::Pixel(56, 204, 108);
palScreen[0x2C] = olc::Pixel(56, 180, 204);
palScreen[0x2D] = olc::Pixel(60, 60, 60);
palScreen[0x2E] = olc::Pixel(0, 0, 0);
palScreen[0x2F] = olc::Pixel(0, 0, 0);

palScreen[0x30] = olc::Pixel(236, 238, 236);
palScreen[0x31] = olc::Pixel(168, 204, 236);
palScreen[0x32] = olc::Pixel(188, 188, 236);
palScreen[0x33] = olc::Pixel(212, 178, 236);
palScreen[0x34] = olc::Pixel(236, 174, 236);
palScreen[0x35] = olc::Pixel(236, 174, 212);
palScreen[0x36] = olc::Pixel(236, 180, 176);
palScreen[0x37] = olc::Pixel(228, 196, 144);
palScreen[0x38] = olc::Pixel(204, 210, 120);
palScreen[0x39] = olc::Pixel(180, 222, 120);
palScreen[0x3A] = olc::Pixel(168, 226, 144);
palScreen[0x3B] = olc::Pixel(152, 226, 180);
palScreen[0x3C] = olc::Pixel(160, 214, 228);
palScreen[0x3D] = olc::Pixel(160, 162, 160);
palScreen[0x3E] = olc::Pixel(0, 0, 0);
palScreen[0x3F] = olc::Pixel(0, 0, 0);
}

```

Por cada *tick* producido por el reloj del sistema, la *PPU* necesitará hacer algo (en otras palabras, habrá sido llamada o *clocked*), algo que le sucede con el triple de frecuencia que a la *CPU*. Así, podremos utilizar la variable del contador del reloj del sistema para llevar también el registro del reloj de la *CPU*, que será relativo al de la *PPU*.

```

void Bus::clock()
{
    ppu.clock();
}

```

```

if (nSystemClockCounter % 3 == 0)
{
    cpu.clock();
}

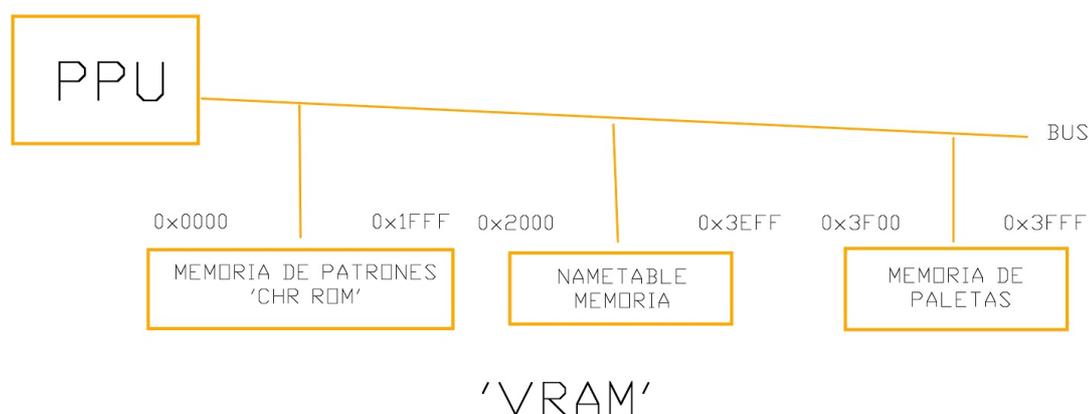
nSystemClockCounter++;
}

```

## El renderizado de la PPU

Como último apartado del análisis de la emulación, veremos más al fondo la PPU y su renderizado de gráficos, en pro de mejorar una emulación imperfecta que, en nuestro estudio, no será soportada por una amplia diversidad de juegos, si bien resultará más que suficiente para la gran mayoría de ellos. Como ya sabemos, la PPU se conecta a través del bus con tres dispositivos clave: la memoria de patrones, la RAM que almacena los *name tables* y la RAM encargada de las paletas de color. La primera contiene los *sprites* (guardados como imágenes de bits), la segunda los diseños de los fondos del juego y la última los colores de estos.

En esta última sección nos centraremos en el renderizado de los fondos, es decir, en el funcionamiento de la segunda RAM. No obstante, cabe destacar que la memoria de patrones no solo sirve para el alojamiento de *sprites*, sino también guarda la información que la PPU necesita para dibujar en la pantalla.

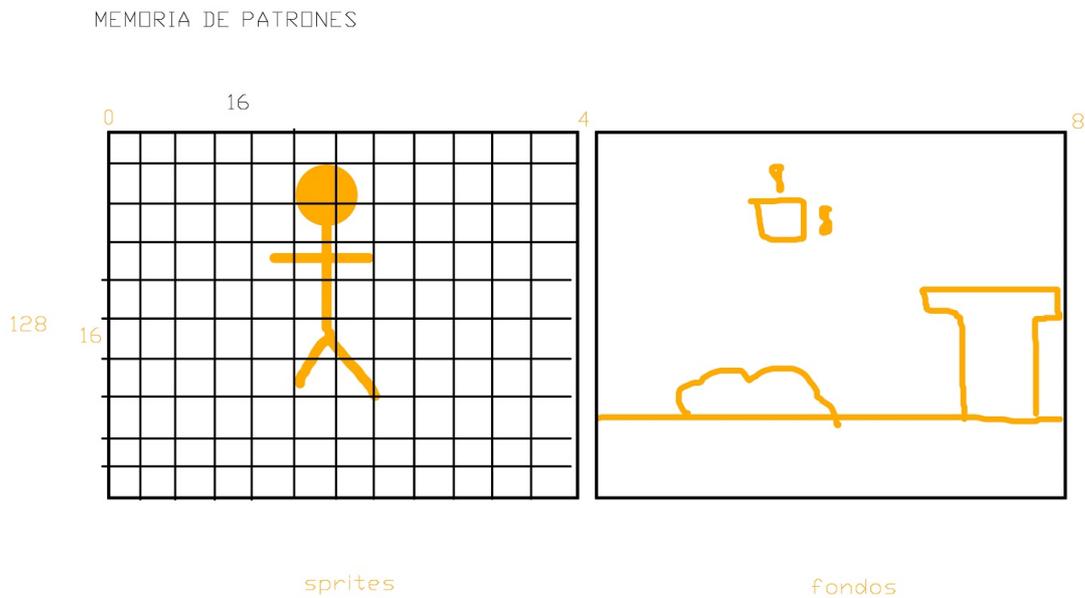


La memoria de patrones de 8 Kb se puede dividir en dos secciones de 4 Kb cada una. A su vez, estas se dividen en una malla de  $16 \times 16$  *sprites* o *tiles*, los cuales no dejan de ser cuadrados de  $8 \times 8$  píxeles. Una imagen que contuviera todos ellos tendría un tamaño, por tanto, de  $128 \times 128$  píxeles.

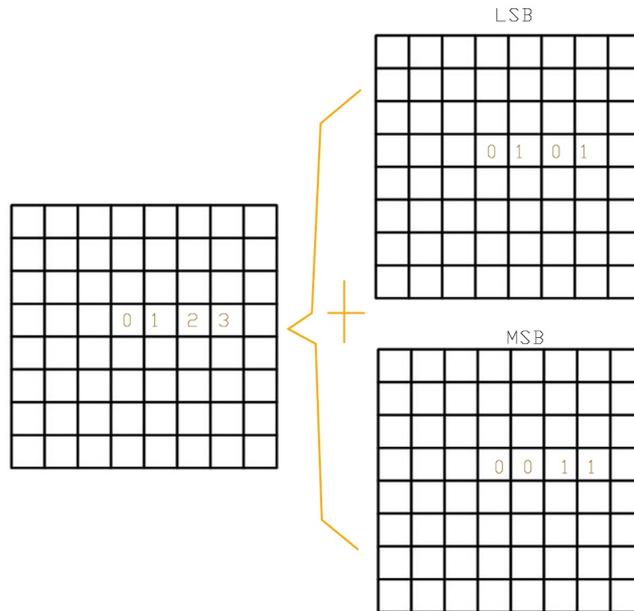
La *PPU* tiene la capacidad para elegir entre la parte derecha de la memoria y la izquierda. Asumiendo que la memoria de patrones se halla plenamente en el cartucho de juego, tendremos que acceder a ella a través del *mapper*, el cual tiene la capacidad de seleccionar los espacios de memoria necesarios para conseguir los *sprites* (muchas de las animaciones de los *sprites*, de hecho, se hacen de este modo).

El tamaño de un *sprite* de  $8 \times 8$  píxeles es muy reducido, por lo que normalmente un *sprite* suele hacerse en varios *tiles* de esta malla. Es por esto que normalmente la mitad izquierda está dedicada al almacenamiento de *sprites* de personajes y la mitad derecha a los fondos.

La *NES* usa  $2$  *bits* por píxel, lo que ofrece un rango de cuatro colores por píxel accesibles desde cuatro números correspondientes  $[0, 1, 2, 3]$ . Para guardar esta información de manera correcta, dividimos cada número en dos planos de *bit*, en función de su importancia. La suma de ambos planos brinda el número correspondiente.



En el caso del valor  $0$ , se alojaría un  $0$  en cada plano. El valor  $1$ , por su parte, tendría un  $0$  en el plano más importante y un  $1$  en el menos importante; el  $2$ ,  $1$  en el más importante; el  $3$ , un  $1$  en cada plano. Esto es algo especialmente a tener en cuenta por un factor que explicaremos en detalle más adelante: los valores  $0$  se pueden considerar transparentes.



Este sistema de mapeo, no obstante, no acaba de ser suficiente para describir el color del píxel, para lo cual se requiere su combinación con una paleta de colores, cuya estructura puede encontrarse a continuación:

PALETAS

- \$3F00
- \$3F01
- \$3F05
- \$3F09
- \$3F0D
- \$3F11
- \$3F15
- \$3F19
- \$3F1D

La primera entrada guarda el color del fondo, que será un valor de 8 bits que nos indica uno de los colores que la NES es capaz de dibujar. Si queremos obtener un color concreto, tendríamos que guardar en la dirección \$3F00 el valor indicado en la paleta (que también se muestra a continuación); el azul oscuro, por ejemplo, se correspondería con el 01, y el verde con el 2a.

savtool's NES palette															
00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f
20	21	22	23	24	25	26	27	28	29	2a	2b	2c	2d	2e	2f
30	31	32	33	34	35	36	37	38	39	3a	3b	3c	3d	3e	3f

Mientras que la primera entrada guarda un byte, el resto guardan cuatro bytes de información. En el caso de la dirección, \$3F01 el último byte no se utiliza, mientras

que los tres primeros guardan colores. La razón de ello es simple: su uso debe de estar acompañado de una numeración en cada entrada, empezando desde el 0 hasta el 7. Si queremos el *ID* de la paleta igual a 1 y el píxel 3, conscientes de la disposición de cuatro byte, multiplicaremos el *ID* de la paleta por 4 y sumaremos el valor del píxel ( $1 \times 4 + 3 = 7$ ). Teniendo en cuenta que nuestra paleta comienza en  $\$3F00$  podemos comenzar a contar desde ahí.

## PALETAS

\$3F00				
\$3F01				
\$3F05	.	.		
\$3F09				
\$3F0D				
\$3F11				
\$3F15				
\$3F19				
\$3F1D				

Como vemos en el esquema, nuestro *byte* sería el tercer *byte* de la dirección  $\$3F05$ .

Si el valor del píxel hubiese sido 0, el total habría sido 4, lo que nos llevaría al cuarto *byte* de la segunda dirección (sin uso). Al reflejar el 0 la transparencia, debe de reflejar el valor atribuido al fondo en lugar de caer en desuso; esta problemática es aplicable a todos los cuartos *bytes* de cada dirección. Una última división de estas entradas que debemos explicar es que las cuatro primeras son utilizadas para las paletas del fondo y las cuatro últimas para los *sprites*.

Ahora que entendemos su funcionamiento teórico, podemos empezar a entrar en detalle con respecto al código. Aunque ya presentábamos un código extenso correctamente implementado (incluyendo la conectividad con la *CPU* y su bus y la paleta de colores de la *NES*), en el anterior apartado nuestro código era un esqueleto de la *PPU*: no podía hacer nada más que crear ruido artificial.

La primera función que construiremos, partiendo de dicha base, será para la representación de *sprites*. Tendremos que crear la malla de  $16 \times 16$ , a la que le pasaremos el *sprite* a dibujar y su paleta de colores.

```

olc::Sprite& olc2C02::GetPatternTable(uint8_t i, uint8_t palette) {
for (uint16_t nTileY = 0; nTileY < 16; nTileY++)
{
    for (uint16_t nTileX = 0; nTileX < 16; nTileX++)
    {
        // Convierte una coordenada de un tile en 2D en un offset en 1D
        // dentro del patrón de la memoria.

        uint16_t nOffset = nTileY * 256 + nTileX * 16;

        //Ahora recorre en bucle filas de 8 bits.
        for (uint16_t row = 0; row < 8; row++)
        {

```

```

// Para cada fila, necesitamos leer ambos planos de bits del carácter
// con el fin de extraer los bits menos importantes y los más importantes
// del valor de píxel de 2 bits. En CHR ROM, cada carácter se almacena
// como 64 bits de lsb, seguido de 64 bits de msb. Esto significa que
// dos filas correspondientes son siempre distancias de 6 bytes en la memoria.

        uint8_t tile_lsb = ppuRead(i * 0x1000 + nOffset + row +
0x0000);
        uint8_t tile_msb = ppuRead(i * 0x1000 + nOffset + row +
0x0008);

// Ahora tenemos una sola fila de los dos planos de bits para el carácter
// que necesitamos iterar a través de las palabras de 8 bits, combinándolos
// para darnos el índice de píxeles final.
        for (uint16_t col = 0; col < 8; col++)
        {
// Podemos obtener el valor del índice simplemente sumando los bits, pero
// solo nos interesa el lsb de las palabras de la fila porque ...
            uint8_t pixel = (tile_lsb & 0x01) + (tile_msb &
0x01);

// ... desplazamos las palabras de la fila 1 bit a laderecha para cada
// columna del carácter.
                tile_lsb >>= 1; tile_msb >>= 1;
// Ahora que conocemos la ubicación y el valor de píxel de NES para una ubicación
// específica en la tabla de patrones, podemos traducir eso a un color de
// pantalla
// y una ubicación (x, y) en el sprite

                sprPatternTable[i].SetPixel
                (
                    nTileX * 8 + (7 - col),
// Debido a que usamos el lsb de la palabra de la fila primero, estamos leyendo
// la fila de derecha a izquierda, así que necesitamos dibujar la fila "al revés"
                    nTileY * 8 + row,
                    GetColourFromPaletteRam(palette, pixel)
                );
            }
        }
    }
}

// Finalmente devuelve el sprite actualizado que representa la tabla de patrones
return sprPatternTable[i];
}

```

Como hemos visto, en el proceso tenemos que llamar a una función llamada `GetColourFromPaletteRam (palette, pixel)` que nos permite dibujar correctamente el píxel, pasándole como argumentos la paleta y el píxel correspondientes. Es ahí donde debemos de aplicar la fórmula del ID de la paleta (multiplicándolo por cuatro).

```

olc::Pixel& olc2C02::GetColourFromPaletteRam(uint8_t palette, uint8_t pixel)
{
    // Esta es una función que toma una paleta específica y un índice de píxeles

```

```

//y devuelve el color de pantalla apropiado.
// 0x3F00" - Offset en el rango direccionable de la PPU donde se almacenan las
paletas
// "palette << 2" - Cada paleta es 4 bytes
// "pixel" - Cada índice de píxel es 0, 1, 2 o 3
// "& 0x3F" - Nos detiene leer más allá de los límites de la matriz
return palScreen[ppuRead(0x3F00 + (palette << 2) + pixel) & 0x3F];

// NOTA: No accedemos a tblPalette aquí, en cambio sabemos que ppuRead()
// mapeará la dirección en una RAM más pequeña en el BUS de la PPU.
}

```

Como se menciona en los comentarios del código, debemos usar la función `ppuRead()` para completar el dibujado de píxeles. Sin embargo, tanto `ppuRead()` como `ppuWrite()` se encuentran sin detallar en este punto del estudio. Nuestro siguiente paso, así, deberá de basarse en completar su codificación. Al presentar un código tan extenso (y poco novedoso), resumiremos su funcionamiento.

Comencemos por la función de lectura. Lo primero que haremos será añadir las tres memorias que necesitamos para leer el cartucho y hacer las relocalizaciones de las direcciones. La memoria de patrones tiene las direcciones de `0` a `1FFF`, y la memoria de tablas desde `2000` a `3FFF`. Por último, la memoria de paletas va desde `3F00` a `3FFF`.

Las tres memorias se añaden como *arrays*, y en el caso de la memoria de patrones, se leen los *5 bytes* necesarios para llegar al lugar de la memoria preciso. Para la lectura, simplemente indicaremos la dirección, y si es el lugar derecho o izquierdo del *array* de datos. Las otras dos memorias presentan un comportamiento muy similar.

La variable `selectedpalette` servirá para escoger la paleta a utilizar. El valor de esta, que podrá variar en tiempo de ejecución, aumentará apretando la tecla *P*. Si activamos ahora la emulación veremos que abajo a la derecha se han añadido dos cuadrados que representan nuestra malla de memoria. Al presionar la *P* se indica sobre cuál de las ocho paletas se está realizando la ejecución. No obstante, el emulador aún presenta carencias básicas que lo hacen poco funcional: todas las paletas se ven grises, y la pantalla de *NES* no carga nada. El problema es simple: el programa está intentando cargar un valor (`2002`) desde el bus de la *CPU*, y no está consiguiendo el dato que busca.

Los registros del `2000` a `2007` son muy importantes, ya que son los que controlan la *PPU*. La *CPU* se comunica con la *PPU* a través de esos ocho registros:

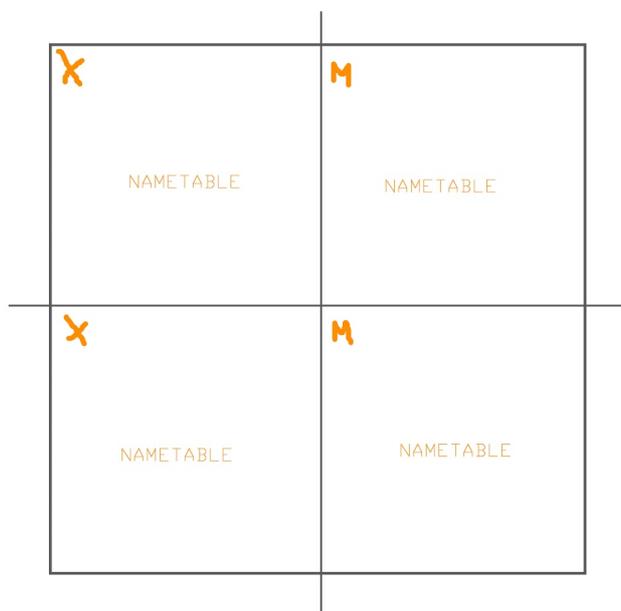
- ❖ `0x2000` → Configura la *PPU* para renderizar de diferentes modos.
- ❖ `0x2001` → Decide qué fondos y *sprites* se dibujan, y qué ocurre en los márgenes de la pantalla.
- ❖ `0x2002` → Registro de estado: indica cuándo es seguro renderizar.
- ❖ `0x2003` y `0x2004` → No son importantes para esta emulación.

- ❖ **0x2005** → Permite la correcta visualización de la pantalla.
- ❖ **0x2006** y **0x2007** → Permite a la CPU leer y escribir en la memoria de la PPU (dirección y datos).

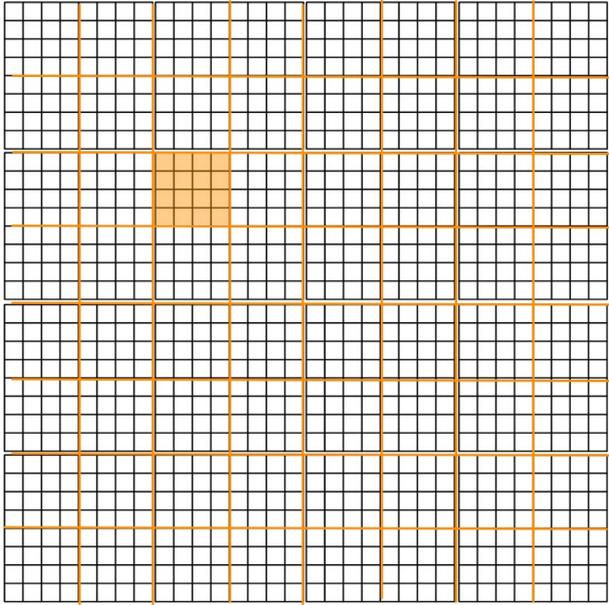
El rango de direcciones de la PPU es de 14 bits, pero la CPU solo puede transferir 8 bits, por lo que se realizan dos escrituras para conseguir la dirección correcta. Por su parte, el registro de estado contiene tres bits: 'vertical blank', sprite cero y sprite overflow; todos están unidos a través de un registro de 8 bits. Por último, el registro 'mark' funciona con una serie de interruptores que representan la PPU que explicamos en la primera parte.

Partiendo de estas bases es hora de encarar, finalmente, la clave de esta sección: el renderizado de fondos. Generalmente suelen ser más estáticos que los objetos, y representan el espacio donde se mueven estos. Los fondos se guardan en la memoria de 'name tables', que tiene un peso de un kilobyte (32 × 32); cada entrada es un byte, y un ID de la memoria de patrones (malla de 16 × 16), lo que configura, en total, una resolución de 256 × 256 píxeles. Sin embargo, solo se pueden usar 240 de los 256 píxeles verticales, resultando en una resolución de 256 × 240.

Para poder hacer que la pantalla se desplace (como ocurre en propuestas como Super Mario Bros.), NES permite guardar a la vez dos NameTables. Usando duplicación de direcciones, podemos conseguir hasta cuatro NameTables, con un peso de 2 Kb.



Los 64 bytes que no se utilizan reciben el nombre de 'memoria de atributos'. Cada uno de ellos es responsable de una función de la tabla, dividiendo así la tabla en secciones de 8 × 8 (cada sección contiene 4 × 4 bytes). De este modo, solo necesitamos 2 bytes por paleta para colorear los 16 bytes, los que a su vez dividiremos en grupos de cuatro con tal de poder indicar cada sección tal y como aparece en el esquema inferior.



00	10
11	01

La memoria de *NameTable*, con dos *bytes* reservados, se accede tanto en la función de escritura como en la de lectura. Para la correcta ejecución del código, resta añadir algunas modificaciones a la función `clock()`, para que así lea correctamente los ciclos de dibujo.

Con todo este análisis y codificación, nuestro emulador de NES está terminado en su funcionamiento más básico.

## CONCLUSIONES Y RESULTADO FINAL

Tras su correspondiente disección, puede llegarse a la conclusión de que la complejidad en el *hardware* de *Nintendo Entertainment System*, hija de su época, se debió a la intención de sus creadores porque la máquina ofreciese unos impactantes resultados a nivel gráfico, aun haciendo gala de unos pocos y económicos recursos.

Entendidas las bases de su arquitectura, la réplica de su funcionamiento a día de hoy - al menos parcial, como la realizada en nuestro caso - no resulta una labor especialmente ardua (gracias, también, a la ingente cantidad de documentación que puede encontrarse en la web). No obstante, tal proceso, conforme avanzamos hacia un futuro en el que cuestiones como la gestión de memoria ocupan una relevancia mínima, se antoja cada vez más necesario, al servir como estudio de la maquinaria pasada y actual, y como apreciación del trabajo realizado por los diseñadores de *hardware* hace ya casi cuarenta años.

El emulador resultante de haber realizado todas y cada una de las operaciones detalladas puede encontrarse (y ejecutarse, correspondientemente) en el siguiente enlace:



URL: <https://cutt.ly/YgV9nx3>



**DECONSTRUYENDO NES**

*Trabajo teórico por Lara Portillo y Carlos Sánchez*

## BIBLIOGRAFÍA

- ❖ [\*Nintendo Entertainment System Documentation\*, por Patrick Diskin](#)
- ❖ [\*Rockwell R650X and R651X Microprocessors \(CPU\)\*, por Rockwell Automation](#)
- ❖ [\*The Hardware Book\*, por Joakim Ögren](#)
- ❖ [\*History of Nintendo: Volume One: Ultimate Guide to Nintendo Games & Hardware\*, por Brian C Byrne](#)

## WEBGRAFÍA

- ❖ [NesDev](#)
- ❖ [NesDev Wiki](#)
- ❖ [Famicom Party](#)
- ❖ [OneLoneCoder](#)
- ❖ [Handheld Museum](#)
- ❖ [NesCart Database](#)
- ❖ [The History About How We Play](#)
- ❖ [Hiscoga](#)
- ❖ [AssemblerGames](#)
- ❖ [Retro Reversing](#)
- ❖ [HyperHype.es](#)