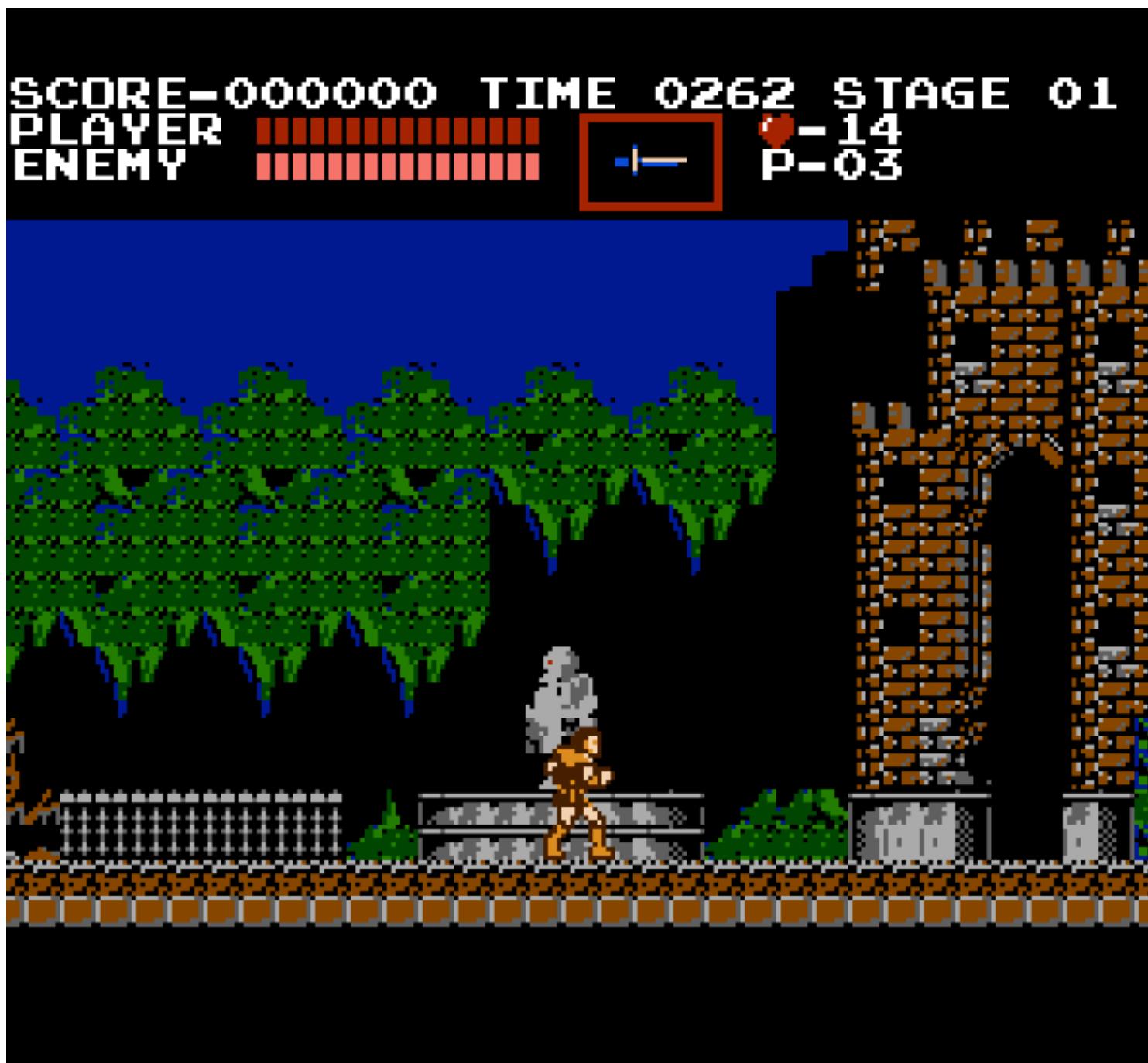


PROJETO MALCONES

PCS3732 - LABORATÓRIO DE PROCESSADORES

Rodrigo Sales Marcolin	12551432
Vinicio Batista da Silva	13695661
Vitor Sasaki Venzel	13682573



Sumário

Introdução.....	2
CPU (Central Processing Unit).....	4
Memória da CPU.....	11
Emulação.....	13
PPU (Picture Processing Unit).....	19
Memória da PPU.....	27
Ciclo de execução da PPU.....	34
Emulação.....	37
Controles.....	47
Emulação.....	48
Mappers e Cartuchos.....	52
Exemplo Mapper 2.....	53
Emulando Mapper 2.....	54
APU (Audio Processing Unit).....	57
Canais de pulso.....	57
Canal de triângulo.....	60
Canal de Ruido.....	60
Canal DMC.....	61
Mixer.....	62
Emulação.....	63
Executando na placa RockPro64.....	72

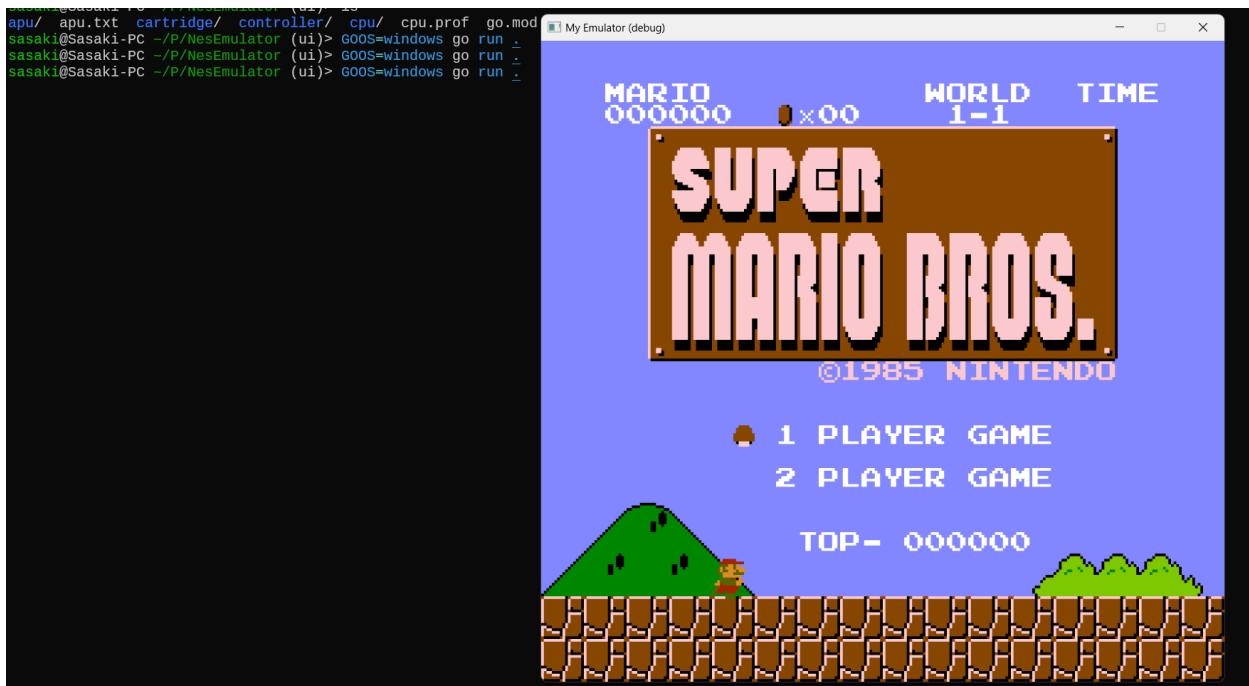
Introdução

Nosso projeto será o desenvolvimento de um emulador de NES (Nintendo Entertainment System) conhecido popularmente como nintendinho, o primeiro console de 8-bits produzido pela Nintendo que foi o primeiro de sistema onde várias grandes séries de jogos tiveram seu início, como Super Mario Bros, The Legend of Zelda, Metroid, Mega Man, Final Fantasy, entre outros.

O objetivo do projeto é conseguir, a partir de dumps da memória de cartuchos desses jogos, também conhecidos como ROMS (NÃO pesquisem roms de Super Mario Bros para o Nintendinho no Google, NÃO vai aparecer um primeiro link onde NÃO vai estar disponível para baixar) simular a execução do processador e dos outros componentes do console (placa de vídeo, processador de áudio, controle etc) a fim de conseguirmos jogar os jogos em nosso sistema.

O Emulador será escrito na linguagem de programação Go, uma linguagem compilada criada pela Google e pelo lendário criador da linguagem B [Ken Thompson](#). É uma linguagem compilada que tem como objetivo ser rápida como C, mas também ter a simplicidade e produtividade das linguagens interpretadas de alto nível. Foi escolhido Go para o projeto, simplesmente pelo fato de ser uma linguagem diferente da linguagem C, mas que cumpria os requisitos que estávamos buscando, sendo uma linguagem compilada, tipada, com vários operadores binários úteis para programar um emulador.

O emulador já está funcional (áudio e vídeo), e seu código está neste repositório do github: [NesEmulator](#)



O console do nintendinho é essencialmente formado pelos seguintes componentes, que devem ser devidamente emulados:

CPU (Central Processing Unit) -> Processador baseado no processador 6502

PPU (Picture Processing Unit) -> "Placa Gráfica" do nintendinho

APU (Audio Processing Unit) -> "Placa de Áudio" do nintendinho

JoyPad -> Controle

Mappers -> Essencialmente os circuitos internos que existiam em alguns cartuchos de jogos

Uma das principais fontes que utilizei para desvendar o funcionamento dos componentes, foi a wiki criada pelos entusiastas de nintendinho: [wiki NesDev](#)

Para exibir as coisas na tela e tocar os sons, será utilizado as bibliotecas Ebiten e Oto disponíveis no ecossistema Go. Vale lembrar que elas serão utilizadas apenas como uma interface para propagar as saídas dos componentes emulados para nossas telas e speakers, não implementando nenhuma lógica.

Nos próximos posts, devo aprofundar em cada um dos componentes, explicando todos os seus comportamentos explicáveis (tem coisa que é melhor só aceitar...) e como os mesmo se tornaram código

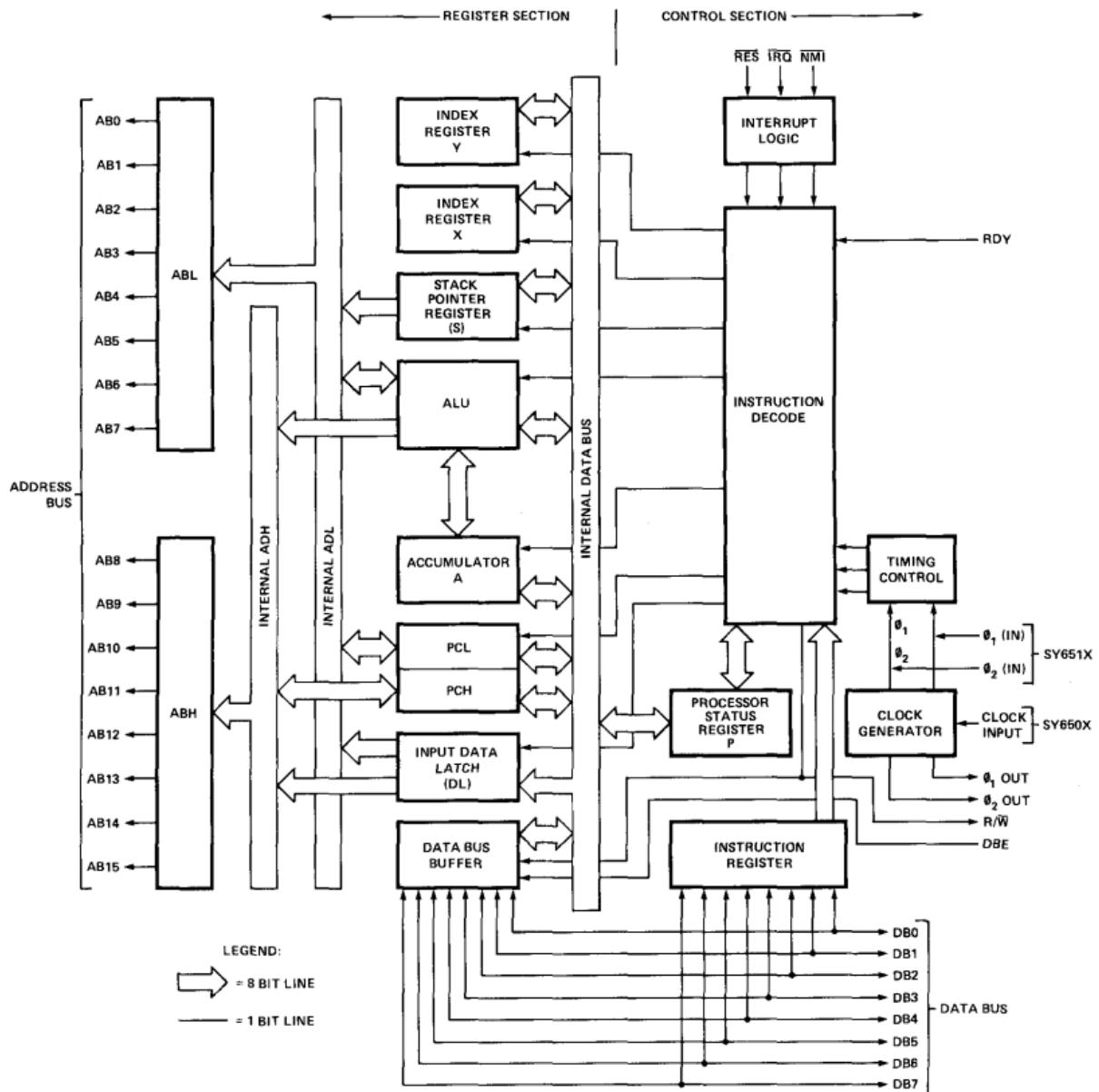
Antes de iniciar as explicações, é importante saber que os consoles do nintendinho não são todos iguais. Devemos lembrar que o videogame é de um tempo antigo, lançado em 1983 em um mundo muito menos globalizado, onde nem a Internet tinha se popularizado ainda. O console possui 2 versões, criadas por conta das diferenças entre as tecnologias de televisores utilizadas pelo mundo na época: NTSC, a versão "original" utilizada nos Estados Unidos e Japão, e PAL, utilizada no resto do mundo. Há algumas diferenças entre as duas versões, e até os jogos eram adaptados de uma versão para outra. A versão que eu emulei é a versão NTSC, e irei comentar mais sobre quando houverem diferenças na implementação.

Além disso, por ser muito antigo, todos os jogos eram programados em assembly, que era então montado e gravado em memórias ROM que eram distribuídas em formato de cartucho. Não havia um compilador de linguagem C ou outra linguagem de alto nível para o assembly do processador do nintendinho, e francamente não havia muito sentido em existir, dado os recursos limitados e específicos do console. Vale lembrar que cada byte era valioso, e que cada byte a menos no código do jogo resultaria em cartuchos mais baratos e jogos mais rápidos.

CPU (Central Processing Unit)

A CPU do Nintendinho se chama RP2A03 (RP2A07 na versão PAL), e é uma modificação do famoso processador 6502, mas feito pela Nintendo. Ele conta com algumas diferenças, como a remoção do modo decimal, adição da unidade de geração de áudio, entre outras coisas, a fim de escapar de um processo por copiar a patente dos fabricantes do 6502

A arquitetura do processador é como na imagem:



Como podemos observar, é uma arquitetura de acumulador bastante simples, sem pipeline, cache, tomasulo e todas as coisas que vimos em OAC, reflexo da tecnologia limitada da época. Ele organiza a memória no formato Little Endian, e diferente do ARM, isso não é configurável.

Ele conta com alguns registradores de 8-bits, mas vale lembrar que por ser uma arquitetura de acumulador, esses registradores não são propósito geral, e tem funções bem definidas:

- Program Counter (16 bits):
É o mais diferente dos registradores, já que possui 16 bits ao invés de apenas 8. Isso se dá pelo fato de que se ele tivesse apenas 8 bits, poderíamos ter uma memória de no máximo 256 bytes, o que não iria servir para nada. Além disso, note que ele não é um registrador de 16 bits completo, mas sim dois registradores de 8 bits, por conta disso algumas instruções que fazem com que o registrador de bits mais altos do PC sejam atualizados acabam levando um ciclo a mais, pois são feitas duas somas ao invés de apenas uma. Para confundir ainda mais, por mais que a memória do console não seja paginada, e não tenha nada a ver com paginação, é comum dizer que a memória contém 256 páginas (definidas pelos 8 bits mais altos) de 256 bytes cada (8 bits mais baixos), justamente por conta dessa divisão do PC, e da maior demora quando há um cruzamento de "página". Exemplo: endereço 0x12F6 -> página 0x12, byte 0xF6
- Registradores X e Y (8 bits)
São registradores utilizados em modos de endereçamento de algumas instruções
- Acumulador (8 bits)
É o registrador mais importante, que recebe a maior parte dos resultados das operações aritméticas, load e store que um processador deve ser capaz de fazer
- Stack Pointer (8 bits)
Assim como no ARM, aponta para o topo da pilha, e é utilizado por algumas instruções de push e pop (desta vez não são pseudo!) e também em rotinas de tratamento de interrupções. Entretanto, diferente do ARM, a pilha não é apenas uma abstração, ela realmente existe em hardware, e é simplesmente um espaço reservado da memória que vai do endereço 0x0100 até 0x01FF, também chamado de página 1 (note o byte mais alto igual a 1 0x0100)

- Processor Status (8 bits)

Armazena as flags de status do processador:

7	bit 0

NV1B DIZC	
	+-
	Carry
	+-
	Zero
	+--
	Interrupt Disable
	+---
	Decimal
	+-----
	(No CPU effect; see: the B flag)
	+-----
	(No CPU effect; always pushed as 1)
+	+-----
	Overflow
+	+-----
	Negative

- Carry, Zero, Overflow e Negative são como no ARM, e no geral são setadas quando o resultado de uma operação atende a condição da flag
- Interrupt Disable: Desabilita o tratamento de interrupções, fazendo com que o processador as ignore. Infelizmente nem todas as interrupções são ignoradas, algumas simplesmente são especiais, e teremos de viver com isso
- Decimal: Flag inútil, resíduo da origem do processador como modificação do 6502, não serve para nada
- Flag B: Não tem efeito para o processador, mas é útil para o programador (questionável). Indica se a última interrupção foi causada por um sinal externo, ou por uma instrução.

Instruções

As instruções desse processador são diferentes das instruções padrões que vimos em processadores RISC como o ARM, podendo ter tamanhos variados entre 1 e 3 bytes e levando quantidades de ciclos variáveis para serem executadas. A estrutura das instruções é bem simples, sendo o primeiro byte (presente no endereço que o PC está apontando) chamado de OPCODE, que é decodificado pela unidade de controle, e dependendo de seu valor, pode levar a leitura dos próximos 1 ou 2 bytes.

Tabela com todas os opcodes:

+00	+04	+08	+0C	+10	+14	+18	+1C	+01	+05	+09	+0D	+11	+15	+19	+1D	+02	+06	+0A	+0E	+12	+16	+1A	+1E	+03	+07	+0B	+0F	+13	+17	+1B	+1F
00	BRK	NOP	PHP	NOP	BPL	NOP	CLC	NOP	ORA	ORA	ORA	ORA	ORA	ORA	ORA	STP	ASL	ASL	ASL	STP	ASL	NOP	ASL	SLO	SLO	ANC	SLO	SLO	SLO	SLO	
	d	a	*+d	d,x	a,x	(d,x)	d	#i	a	(d,y)	d,x	a,y	a,x		d		a		d,x	a,x	(d,x)	d	#i	a	(d,y)	d,x	a,y	a,x			
20	JSR	BIT	PLP	BIT	BMI	NOP	SEC	NOP	AND	AND	AND	AND	AND	AND	AND	STP	ROL	ROL	ROL	STP	ROL	NOP	ROL	RLA	RLA	ANC	RLA	RLA	RLA	RLA	
	a	d	a	*+d	d,x	a,x	(d,x)	d	#i	a	(d,y)	d,x	a,y	a,x		d		a		d,x	a,x	(d,x)	d	#i	a	(d,y)	d,x	a,y	a,x		
40	RTI	NOP	PHA	JMP	BVC	NOP	CLI	NOP	EOR	EOR	EOR	EOR	EOR	EOR	EOR	STP	LSR	LSR	LSR	STP	LSR	NOP	LSR	SRE	SRE	ALR	SRE	SRE	SRE	SRE	
	d	a	*+d	d,x	a,x	(d,x)	d	#i	a	(d,y)	d,x	a,y	a,x		d		a		d,x	a,x	(d,x)	d	#i	a	(d,y)	d,x	a,y	a,x			
60	RTS	NOP	PLA	JMP	BVS	NOP	SEI	NOP	ADC	ADC	ADC	ADC	ADC	ADC	ADC	STP	ROR	ROR	ROR	STP	ROR	NOP	ROR	RRA	RRA	ARR	RRA	RRA	RRA	RRA	
	d	(a)	*+d	d,x	a,x	(d,x)	d	#i	a	(d,y)	d,x	a,y	a,x		d		a		d,x	a,x	(d,x)	d	#i	a	(d,y)	d,x	a,y	a,x			
80	NOP	STY	DEY	STY	BCC	STY	TYA	SHY	STA	STA	NOP	STA	STA	STA	STA	NOP	STX	TXA	STX	STP	STX	TXS	SHX	SAX	SAX	XAA	SAX	AHX	SAX	TAS	AHX
	#i	d	a	*+d	d,x	a,x	(d,x)	d	#i	a	(d,y)	d,x	a,y	a,x		#i	d	a		d,y	a,y	(d,x)	d	#i	a	(d,y)	d,y	a,y	a,y		
A0	LDY	LDY	TAY	LDY	BCS	LDY	CLV	LDY	LDA	LDA	LDA	LDA	LDA	LDA	LDA	LDX	LDX	TAX	LDX	STP	LDX	TSX	LDX	LAX	LAX	LAX	LAX	LAX	LAX	LAX	
	#i	d	a	*+d	d,x	a,x	(d,x)	d	#i	a	(d,y)	d,x	a,y	a,x		#i	d	a		d,y	a,y	(d,x)	d	#i	a	(d,y)	d,y	a,y	a,y		
C0	CPY	CPY	INY	CPY	BNE	NOP	CLD	NOP	CMP	CMP	CMP	CMP	CMP	CMP	CMP	NOP	DEC	DEX	DEC	STP	DEC	NOP	DEC	DCP	DCP	AXS	DCP	DCP	DCP	DCP	
	#i	d	a	*+d	d,x	a,x	(d,x)	d	#i	a	(d,y)	d,x	a,y	a,x		#i	d	a		d,x	a,x	(d,x)	d	#i	a	(d,y)	d,x	a,y	a,x		
E0	CPX	CPX	INX	CPX	BEQ	NOP	SED	NOP	SBC	SBC	SBC	SBC	SBC	SBC	SBC	NOP	INC	NOP	INC	STP	INC	NOP	INC	ISC	ISC	SBC	ISC	ISC	ISC	ISC	
	#i	d	a	*+d	d,x	a,x	(d,x)	d	#i	a	(d,y)	d,x	a,y	a,x		#i	d	a		d,x	a,x	(d,x)	d	#i	a	(d,y)	d,x	a,y	a,x		

para obter o opcode, basta somar o valor da linha com o da coluna, por exemplo, para obter o opcode da instrução CLV (no bloco vermelho à esquerda), basta somar 0xA0 + 0x18 = 0xB8

Podemos notar nessa tabela que várias instruções com mnemônicos iguais, como ADC, LDA e STA, aparecem várias vezes com opcodes diferentes. Isso se dá pelo fato que existem vários modos de endereçamento (são as letrinhas embaixo do mnemônico na tabela, os a,y #i (d,x) etc), e o jeito que encontraram para a CPU interpretar cada um, foi dar um opcode diferente para cada par (mnemônico , endereçamento). Esses são os seguintes modos de endereçamento:

- **Implícito** (indicado na tabela por espaço vazio embaixo do mnemônico)
São as instruções que simplesmente não endereçam nada, ou o endereçamento é implícito, como a instrução PHA, que coloca o valor do acumulador na pilha, ou CLC, que limpa a flag de carry, não endereçando a memória.
- **Imediato - #i**
É o mesmo imediato do ARM, indica que o próximo byte depois do opcode é o valor que deve ser utilizado na operação. Um exemplo seria a instrução ora #\$20 (realize OR do conteúdo do Acumulador com o imediato hexa 0x20), que ao passar pelo montador vira os seguintes bytes: 09 20, sendo o primeiro byte o opcode (consulte na tabela), e o segundo é o próprio valor 0x20 que deve ser utilizado na operação.
- **Zero Page - d**
Como dito anteriormente, a memória não é paginada e não tem nada a ver com paginação. Quando a instrução tem esse modo de endereçamento, devemos pegar o byte seguinte ao opcode, e usá-lo para endereçar o byte presente no endereço da página zero (lembrando, nesse contexto página são os 8 bits mais altos do PC, logo pegamos o valor no endereço 0x00[byte]). Um exemplo é a instrução adc \$56 (some acumulador + byte presente no endereço 0x0056 + carry), que ao montar, gera os seguintes bytes 65 56.
- **Absolute - a**
É similar ao Zero Page (d) explicado acima, mas ao invés de pegarmos apenas da

página zero, usamos os próximos 2 bytes seguintes ao opcodes, e usamos isso para formar um endereço completo para onde vamos endereçar. Como exemplo temos a instrução stx \$5612 (armazene o valor do registrador X no endereço 0x5612), que ao montar fica: 8e 12 56. Note que está invertido o endereço, mas isso se dá pelo uso de little-endian.

- Absolute indirect - (a)

Só a instrução de JMP (opcode 0x6C) usa esse método de endereçamento, ele funciona da mesma forma que o Absolute normal, mas ao invés de buscar apenas 1 byte, busca também o byte no endereço + 1, e usa esses dois bytes como endereço para mover o PC. Essa instrução contém um Bug, onde caso o endereço absoluto termine em 0xFF, ao buscar o segundo o byte, ele não incrementa a página do PC, incrementando apenas os 8 bits mais baixos, fazendo com que o endereço vire 0x00. Exemplo: JMP (\$06FF) vai buscar os bytes nos endereços 0x06FF e 0x06FF + 0x0001 => 0x0600 (BUG NO HARDWARE!!)

- Zero Page Indexed X - d,x ; Zero Page Indexed Y - d,y

É no mesmo formato do endereço Zero Page, mas ao invés de usar o byte diretamente como endereço, ele realiza a soma do byte com o valor no registrador X ou Y e usa isso para indexar na memória. Vale notar que caso a soma seja maior que 0xFF, ele simplesmente dá a volta, e começa a partir do 0x00, sem incrementar a página (se o byte for 0xFE, e o valor no registrador for 0x03 a soma resulta 0xFE + 0x03 = 0x02, sendo o endereço final 0x0002). Exemplo: sbc \$40,x(subtrair do acumulador o valor pego da memória e o carry), que vira f5 40 ou stx \$FF,y, que vira 96 ff

- Absolute Indexed X - a,y ; Absolute Indexed Y - a,y

Funciona no mesmo esquema do Zero Page Indexed, mas agora o registrador é somado a um endereço completo de 16 bits, e pode ocorrer mudança de página (não esquecer que não tem nada a ver com a paginação do JJ!!). Exemplo lda \$04FF,x(load para A, o valor no endereço calculado), que vira bd ff 04. Se o endereço calculado passar de 0xFFFF, ele dá a volta para 0x0000, assim como o Zero Page faz na página.

- Indexed indirect X - (d,x)

É o modo mais estranho e suspeito do processador. Ele soma o valor do registrador X com o byte seguinte ao opcode, e usa esse valor como endereço de página zero para buscar 2 bytes, que serão utilizados como o endereço para a operação. Exemplo: lda (\$00,x)(load acumulador), que vira a1 00, imagine que o valor no registrador X é igual a 0x03, e que temos o seguinte na memória:

0x0000 00

0x0001 00

0x0002 00

0x0003 05

0x0004 65

0x0005 00

0x0006 00

....

0x6505 23

Primeiro é feita a soma do byte 0x00 com o registrador X, que é igual a 0x03 (neste caso, se passar de 0xFF, não há mudança de página, e ele volta para 0x00), em seguida, ele pega os valores nos endereços 0x0003 e $0x0003 + 1 = 0x0004$ (novamente, se ultrapassar 0xFF, não há mudança na página), que nesse caso são iguais a 0x05 e 0x65 respectivamente, e os combina para formar um endereço completo de 16 bits: 0x6505(Little Endian sempre!). Esse endereço então é usado para obter o valor que deve ser carregado no acumulador, nesse caso 0x23

- Indirect indexed Y - (d), y

Esse modo é bem mais simples que o modo anterior, primeiro utilizamos o byte seguinte ao opcode como endereço de página zero para obter dois bytes da memória, que então juntamos em um único endereço completo e somamos o valor do registrador Y nele para obter o endereço final. Exemplo: cmp (\$03),y(compara valor com acumulador e seta flags), que vira d1 03. Supondo que a memória está no mesmo estado que no exemplo anterior, temos o seguinte: pegamos o byte 0x03, e com ele obtemos dois bytes da página zero, um no endereço 0x0003 e outro no endereço $0x0003 + 1 = 0x0004$ (não há mudança de página se ultrapassar 0xFF), que nesse caso são iguais a 0x05 e 0x65 e montamos o endereço 0x6505, que então somamos ao registrador Y (nesse caso poderá haver mudança de página ex: $0x65FF + 0x0002 = 0x6601$), e esse endereço é utilizado para buscar o byte que iremos comparar com o acumulador.

- Relativo *+d

Só é usado nas instruções de Branch, e como o nome sugere, utiliza o próximo byte depois do opcode como um incremento relativo que será adicionado no PC para calcular o endereço para onde o PC deve ir caso a condição do branch ocorra (não existe branch incondicional como no ARM, ou use condicional, ou use JMP). Nesse caso dos branchs, esse incremento é considerado com sinal em complemento de 2, podendo levar o PC para um endereço anterior, o que é muito útil para implementar loops condicionais.

Não vou passar por cada instrução individual explicando seu funcionamento pois levaria uma eternidade, ao invés disso falarei apenas sobre as mais interessantes. Deixo também este site que utilizei como referência para aprender sobre as outras instruções: [Instruções](#)

Instrução brk - Essa é uma instrução interessante, pois ela gera um interrupt IRQ, sendo uma forma primitiva da instrução swi presente no ARM. O seu funcionamento também é parecido, ela empilha o valor do PC e o valor das Flags, e depois coloca no PC o valor igual aos

endereço presente nos 2 bytes lidos nos endereços 0xFFFFE e 0xFFFF, como se estivesse consultando um vetor de interrupções, e pulando para o endereço da subrotina de tratamento que encontrou. Entretanto ela é basicamente inútil, e em todos os jogos que testei, nenhum utiliza essa instrução. Ela tem uma instrução equivalente rti (return from interrupt), que simplesmente desempilha os valores, e retorna a execução de onde ela parou.

Ao investigar mais a fundo, nos deparamos com uma versão primitiva de um vetor de interrupções:

0xFFFFA - byte baixo do endereço da subrotina de interrupção NMI

0xFFFFB - byte alto do endereço da subrotina de interrupção NMI

0xFFFFC - byte baixo do endereço da subrotina de interrupção RESET

0xFFFFD - byte alto do endereço da subrotina de interrupção RESET

0xFFFFE - byte baixo do endereço da subrotina de interrupção IRQ

0xFFFFF - byte alto do endereço da subrotina de interrupção IRQ

A interrupção de RESET ocorre quando é apertado fisicamente o botão de reset no console (note que neste caso, não é um reset completo, já que nem todos os dados na memória RAM podem ser apagados, sendo executada apenas uma subrotina para resetar o estado do jogo. O reset completo só ocorre com um hard reset do console, ou tirando da tomada)

A interrupção IRQ (interrupt request) ocorre quando chamamos a instrução brk, ou por dispositivos externos, como o cartucho ou a unidade de áudio (os controles não geram interrupções!, eles funcionam de uma forma diferente)

A interrupção NMI (non-maskable interrupt) é uma interrupção especial gerada pela placa gráfica, que será detalhada no post sobre ela.

Saindo do tópico de interrupções, podemos notar na tabela que alguns mnemônicos estão em negrito, isso não é por acaso. Essas instruções são chamadas de instruções ilegais, ou instruções não oficiais. Elas existem como instruções residuais do processador 6502 no qual o nintendinho foi baseado. Na teoria, esses opcodes nunca deveriam ser usados para programar jogos, pois o seu comportamento não é documentado e não fazem parte da documentação oficial que a Nintendo liberou para os programadores da época.

Entretanto isso não foi o suficiente para conter os programadores, e quando eles perceberam que existiam alguns opcodes não utilizados, em uma tentativa desesperada de achar algo para otimizar os jogos, começaram gravar esses opcodes estranhos na memória e mandar o processador rodar, tentando ver se a saída tinha algum sentido.

Infelizmente alguns deles tinham sentido, e funcionavam de forma parecida com as instruções equivalentes do 6502, sendo assim incorporados a jogos. Não há um consenso geral sobre o

mnemônico e o funcionamento de algumas dessas instruções, dado que são ilegais, a melhor fonte que encontrei para os entender foi um .txt aleatório: [opcodes ilegais](#).

Desses opcodes, é interessante notar a presença de vários nop com modos de endereçamento diferentes, o que é um tanto cômico, dado que a instrução simplesmente não faz nada, tendo apenas um tamanho maior dependendo do modo. Além disso, temos várias instruções stp, que coloca o processador em modo sleep, que felizmente não são utilizadas em nenhum jogo.

Memória da CPU

Abaixo vai um diagrama mostrando como são mapeados os endereços de CPU do nintendinho:

Address range	Size	Device
\$0000–\$07FF	\$0800	2 KB internal RAM
\$0800–\$0FFF	\$0800	
\$1000–\$17FF	\$0800	Mirrors of \$0000–\$07FF
\$1800–\$1FFF	\$0800	
\$2000–\$2007	\$0008	NES PPU registers
\$2008–\$3FFF	\$1FF8	Mirrors of \$2000–\$2007 (repeats every 8 bytes)
\$4000–\$4017	\$0018	NES APU and I/O registers
\$4018–\$401F	\$0008	APU and I/O functionality that is normally disabled. See CPU Test Mode .
\$4020–\$FFFF	\$BFE0	Unmapped. Available for cartridge use.
• \$6000–\$7FFF	\$2000	Usually cartridge RAM, when present.
• \$8000–\$FFFF	\$8000	Usually cartridge ROM and mapper registers.

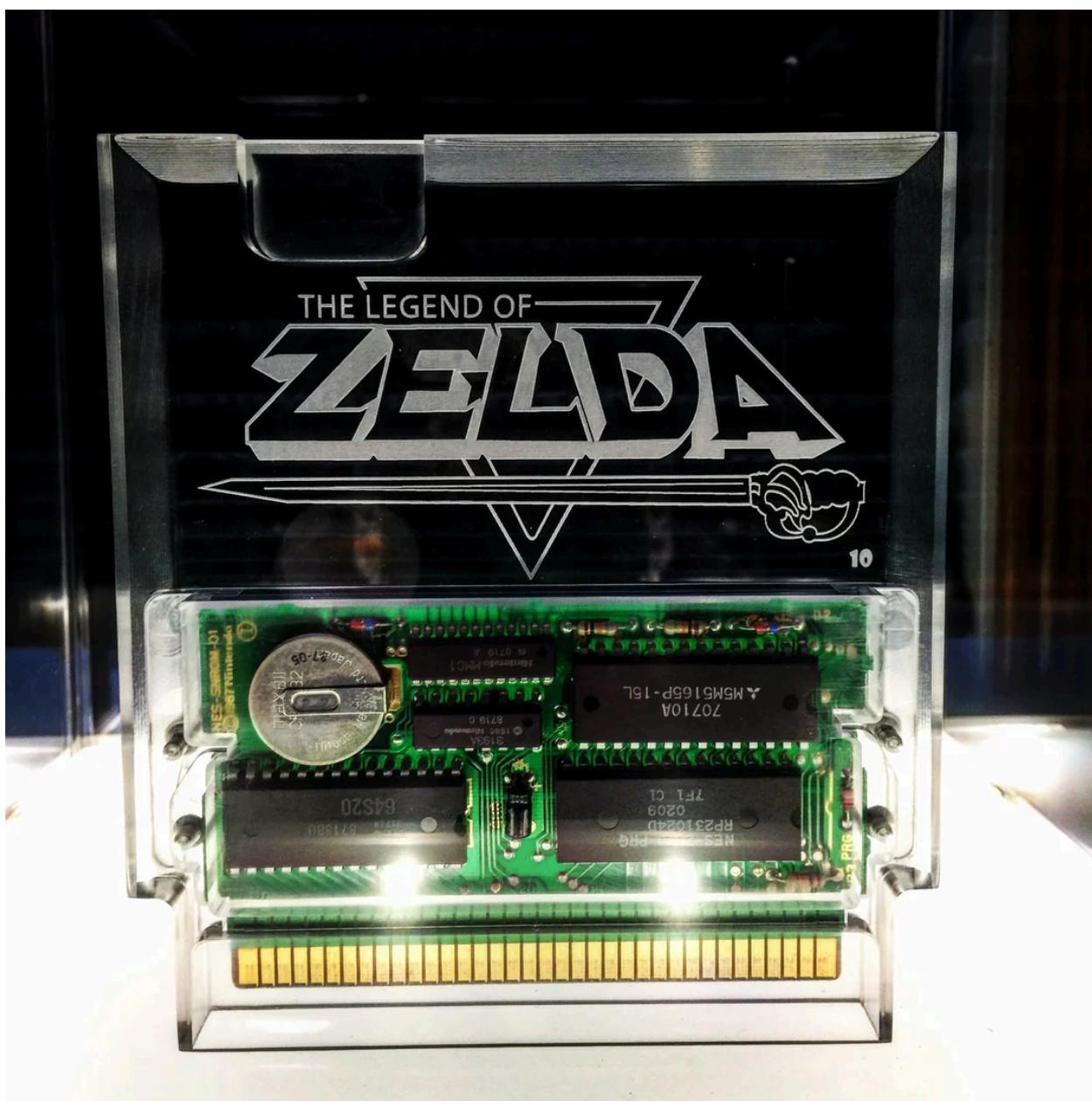
Como dito anteriormente, o Program Counter tem 16 bits, podendo endereçar um espaço de 65536 bytes. Entretanto, como podemos ver na figura, nem todos esses 64K endereços são usados para acessos a memória.

Inicialmente, vemos que ele tem apenas 2KB de memória RAM que são mapeados nos endereços 0x0000-0x07FF do espaço da CPU, ou seja, se fizermos um lda \$0010, estaremos carregando um valor da RAM para o acumulador. Podemos notar também que os próximos endereços são espelhos para a memória RAM (ou seja, os endereços 0x0010 0x0810 0x1010 0x1810 todos mapeiam para o mesmo lugar na memória), o que é contra intuitivo, pois parece apenas desperdiçar endereços, dado que poderíamos colocar mais memória RAM e mapear neles. Entretanto, por mais que isso seja verdade, temos de lembrar que o console foi lançado em 1983, e nessa época chips de memória RAM eram muito caros, e não era viável para a Nintendo colocar mais do que os 2KB de memória que foram colocados.

Em seguida, vemos que temos os Registradores de PPU e seus espelhos mapeados nas posições de 0x2000-0x3FFF. Nesse caso temos os chamados memory mapped registers, são registradores da placa de vídeo que estão mapeados em endereços de memória da CPU, ou seja, caso a CPU execute um Load ou um Store nesses endereços, ela vai estar na verdade escrevendo ou lendo de um registrador da placa de vídeo, efetivamente controlando esse componente.

Nos próximos endereços 0x4000-0x401F temos os Registradores de APU, que são análogos aos da PPU, mas para o processador de áudio, e no mesmo bolo de endereços duas posições de memória especiais, que mapeiam os controles.

Por fim, temos o espaço que é mapeado no cartucho, onde no geral temos nos endereços 0x6000-0x7FFF uma memória RAM alimentada por bateria (lembre que a memória RAM necessita de energia para manter os dados) existente apenas em alguns cartuchos de jogos lançados alguns anos após o lançamento do console, que servem primariamente para implementar salvamento de progresso em jogos longos. Um jogo notável que utiliza essa memória é o primeiro The Legend of Zelda, que por ser um jogo longo, necessitava salvar o progresso



O resto dos espaços 0x8000-0xFFFF são mapeados na memória ROM persistente, contendo essencialmente o código que deve ser executado. Note que o vetor de interrupção está mapeado nesse espaço de memória, então fica a cargo do programador definir ele corretamente.

Emulação

A CPU é o primeiro componente que vamos emular, e é também o mais simples. Para este projeto iremos escrever essencialmente um interpretador, que realiza o fetch da instrução e a executa atualizando os valores nos registradores e na memória.

Vamos começar definindo um struct para armazenar os valores da CPU:

```
type Cpu struct {
    Pc           uint16
    Acc, Xidx, Yidx, Sptr, Psts uint8
    opcodeTable map[uint8]string
}
```

Em seguida podemos definir a tabela de opcodes, que nada mais é que um dicionário que recebe um opcode, e o transforma em um mnemônico que podemos utilizar para agregar instruções iguais, mas com endereçamentos diferentes

```
func Generate() map[uint8]string {
    opcodeTable := map[uint8]string{
        0x61: ADC,
        0x65: ADC,
        0x69: ADC,
        0x6D: ADC,
        0x71: ADC,
        0x75: ADC,
        0x79: ADC,
        0x7D: ADC,

        0x09: ORA,
        0x05: ORA,
        0x15: ORA,
        0x0D: ORA,
        0x1D: ORA,
        0x19: ORA,
        0x01: ORA,
        0x11: ORA,
    }
    // continua para todos os opcode
}
```

Agora vamos definir a memória, que por enquanto podemos considerar apenas como um array de 64K uint8, sem se preocupar com os mapeamentos:

```
type Memory [0xFFFF]uint8
```

Implementamos também alguns métodos para facilitar a leitura e a escrita na memória:

```
func MemRead(addr uint16) uint8 {
    return MainMemory[addr]
}

func MemRead16(addr uint16) uint16 {
    low := uint16(MainMemory[addr])
```

```

        high := uint16(MainMemory[addr+1]) << 8
        return high + low
    }
    func MemWrite(addr uint16, val uint8) uint8 {
        MainMemory[addr] = val
    }
    func MemWrite16(addr uint16, val uint16) uint8 {
        MainMemory.ram[addr] = uint8(val & 0xff)
        MainMemory.ram[addr+1] = uint8((val >> 8) & 0xff)
    }
}

```

Em seguida implementamos na CPU uma função que executa a próxima instrução:

```

func (cpu *Cpu) executeNext() {
    instruction := MainMemory.memRead(cpu.Pc)
    mnemonic := cpu.opcodeTable[instruction]
    addressingMode := (instruction >> 2) & 0b00000111

    switch mnemonic {
        // Accumulator instructions
        case ADC:
            op, size := cpu.getAluOperand(addressingMode)
            var result uint16 = uint16(cpu.Acc) + uint16(op) +
                uint16(cpu.getFlag(Carry))
            cpu.calcAndSetFlags([]string{Carry, Zero, Overflow, Negative},
                result, cpu.Acc, op)
            cpu.Acc = uint8(result)
            cpu.Pc += uint16(size)
        case ORA:
            op, size := cpu.getAluOperand(addressingMode)
            var result uint8 = cpu.Acc | op
            cpu.calcAndSetFlags([]string{Zero, Negative}, uint16(result),
                0, 0)
            cpu.Acc = uint8(result)
            cpu.Pc += uint16(size)
    }
    // continua para o resto dos mnemonicos
}

```

Com isso codificado já conseguimos colocar esta função para rodar dentro de um loop, e se houver instruções, elas vão sendo executadas. Entretanto, com tantos detalhes e comportamentos específicos é extremamente fácil cometer erros. Antes de passar para os próximos componentes, seria interessante realizar testes na CPU, e felizmente, não precisamos bolar os testes nós mesmos. A comunidade entusiasta já desenvolveu um arquivo com um grande número de instruções na maior parte dos contextos possíveis que podemos

executar em nosso processador, e a cada instrução, comparar o estado do nosso processador com um arquivo log que foi gerado pelo hardware original.

Esse arquivo se chama nestest, e para conseguir executar os testes com ele, precisamos codificar uma forma de carregar o seu código para a memória, e um jeito de gerar logs iguais ao log dourado, para que possamos apenas procurar as diferenças entre os nossos e correto.

O Arquivo utilizado pelo nestest tem um formato especial chamado iNES, que na verdade é o formato de arquivo padrão para dumps da memória de cartuchos de jogos, logo não é perda de tempo implementar a sua leitura, já que será usado até o fim do projeto. Não entrarei em detalhes sobre o formato pois foge do escopo, mas aqui está um link que o explica em bastantes detalhes:[Formato INES](#)

Vamos iniciar criando um struct para representar o cartucho, juntamente com seu método que recebe o caminho para um arquivo iNES e faz seu parsing.

```
type Cartridge struct {
    PrgRom      []uint8
    PrgRomSize  uint16
    ChrRom      []uint8
    ChrRomSize  uint16
    Trainer     []uint8
    HasTrainer  bool
    MapperType   uint8
    MirroringType string
}

func ReadFromFile(path string) Cartridge {
    var cartridge Cartridge
    file, err := os.Open(path)
    if err != nil {
        fmt.Println("Error opening file", err)
        panic("Error reading from file")
    }

    defer file.Close()
    reader := bufio.NewReader(file)

    header := make([]byte, 16)
    // continua fazendo parse ....
```

Por enquanto, iremos mexer apenas com PRG ROM, mas já deixei feito o resto das partes do cartucho mesmo sem saber para que servem, dado que eventualmente serão necessárias.

Agora podemos modificar a nossa memória, para mapear os endereços 0x8000-0xFFFF ao nosso cartucho:

```

// type Memory [0xFFFF]uint8

type Memory struct {
    ram [0x0800]uint8
    rom cartridge.Cartridge
}

var MainMemory Memory

func MemRead(addr uint16) uint8 {
    switch {
    case addr <= 0x07FF:
        return MainMemory.ram[addr]
    case addr >= 0x8000:
        return readPrgRom(addr)
    }
    return 0
}
// continuar outros métodos de acesso ...
// .....

func readPrgRom(addr uint16) uint8 {
    addr -= 0x8000
    return MainMemory.rom.PrgRom[addr]
}

```

Agora basta desenvolver uma função que gere um log de mesmo formato que o log utilizado pelos criadores do nestest e colocar os testes para executar. O log tem o seguinte formato:

```
C000 4C F5 C5 JMP $C5F5 A:00 X:00 Y:00 P:24 SP:FD PPU: 0, 21 CYC:7
```

Podemos descartar os valores de PPU (ciclos da placa de vídeo) e CYC (ciclos do processador) por enquanto, e focar em obter os resultados corretos das operações. Não irei explicar a função que gera o Log, pois foge do escopo, mas aqui está o link do repositório no commit em que consegui fazer os testes executarem: [Repositório](#)

Se tudo estiver correto, os testes devem falhar bem no finalzinho do arquivo, na seguinte linha:

```
C68B 8D 15 40 STA $4015 = FF A:02 X:FF Y:15 P:25 SP:FB PPU:233,107 CYC:26520
```

O que faz sentido, dado que o teste está tentando escrever no endereço 0x4015, que é um registrador de APU, que não está mapeado.

Com o interpretador da CPU implementado, podemos agora prosseguir para os próximos componentes necessários para conseguir realmente jogar e se divertir

PPU (Picture Processing Unit)

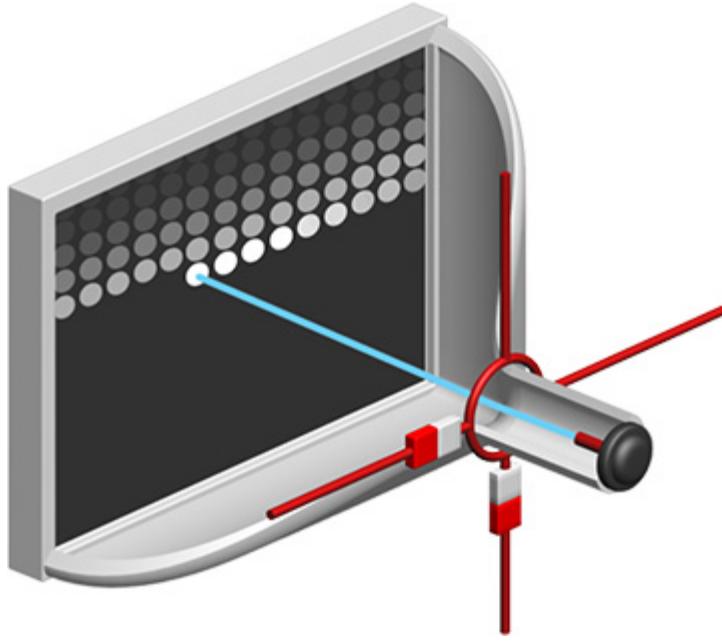
Algumas das imagens desse post foram roubadas de [NES em Rust](#), que por mais que seja um boa fonte, contém bastantes erros e não atingiu uma implementação boa o bastante e não roda nem super mario. Não sei se é culpa da linguagem ou outra coisa... Mas tem boas imagens :)

O próximo componente em que trabalharemos é a chamada Picture Processing Unit ou PPU. Esse componente é basicamente uma placa de vídeo primitiva e simplificada, que é controlada pela CPU do console para exibir o jogo corretamente na tela. No console, o chip utilizado é o 2C02 (2C07 na versão PAL), e sua frequência de clock é 3x maior que a CPU, ou seja, a cada ciclo de CPU, a PPU executa 3 ciclos. Diferente das placas de vídeo atuais, que podem ser programadas para executar várias tarefas diferentes além de enviar vídeo para um tela, esse chip do nintendinho é extremamente especializado: serve apenas para gerar e enviar pixels para TVs de tubo.

As TVs de tubo na época usavam tecnologias de transmissão analógica de vídeo diferentes dependendo da região em que foi produzida, essas tecnologias, que eram NTSC ou PAL definiam o tipo de nintendinho que era vendido na região. Uma das principais diferenças entre NTSC e PAL é a taxa de atualização da tela, sendo 60Hz para NTSC e 50Hz para PAL, fazendo com que os jogos, por mais que funcionem em qualquer de versão do console, aparentam mais lentos ou mais acelerados se jogado na versão errada (ex: jogo PAL parece acelerado em um console NTSC)

Pela PPU ser um hardware bastante especializado, seu funcionamento é bastante similar a forma com que as TVs de tubo geram sua imagem: A TV de tubo funciona direcionando raios de elétrons de variadas intensidades em uma tela fosforescente, fazendo com que haja cor. Elas funcionam varrendo a tela da esquerda para direita, de cima a baixo, disparando os elétrons em cada coordenada, fazendo aparecer um pixel.

Note que nesta imagem estamos vendo por trás da TV, mas a imagem para quem assiste é desenhada da esquerda para direita

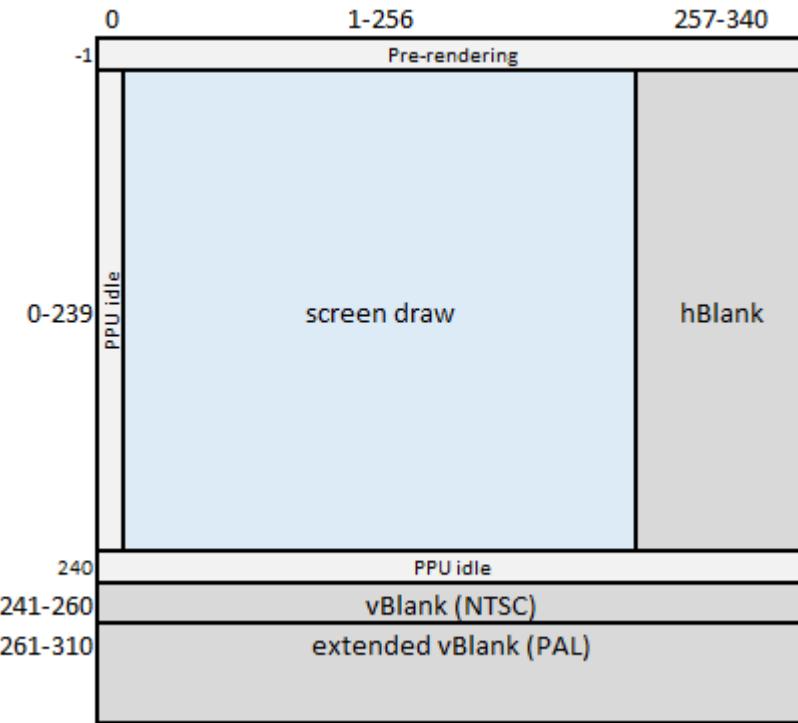


A PPU funciona da mesma forma efetivamente: varre a imagem da esquerda para direita de cima para baixo, e para cada ciclo de clock, busca informações da memória e realiza o processamento para definir qual a cor do pixel que deve ser exibido neste coordenada, e envia este pixel para a TV. Em linguagem mais técnica, cada linha horizontal da TV é chamada de Scanline, sendo no caso do NES, 240 scanlines visíveis, cada uma com 256 dots, ou pixels, dando a resolução final de 256 x 240.

Um frame de um jogo é construído a partir de dois componentes: o background e os sprites. O background é constituído de um grid de blocos 8x8 pixels chamados tiles, que são armazenados na CHR ROM do cartucho e carregados na memória de vídeo da PPU para serem renderizados. Já os sprites são coisas móveis, como os personagens e inimigos de um jogo, que também são formados de tiles, mas tem prioridade sobre o background, aparecendo em frente a ele.

Em mais detalhes, a PPU basicamente executa um loop infinito, onde inicia na primeira scanline no primeiro pixel, e a cada ciclo de clock, avança para o próximo pixel na scanline, e caso seja o último pixel, avança para o primeiro pixel da próxima scanline e caso seja a última scanline, volta para a primeira scanline primeiro pixel, recomeçando o processo. Ao executar esse processo, a PPU passa por 261 scanlines com 340 dots em cada, sendo que desses, apenas os primeiros 256 dots das 240 primeiras scanlines efetivamente geram um pixel.

Quando a PPU está nas 21 scanlines que sobram, dizemos que ela está em VBlanking (vertical blanking), e quando está nos 84 dots extras, ela está em HBlanking (Horizontal Blanking). Tanto a primeira scanline quanto o primeiro dot de cada scanline também não renderiza nada, sendo apenas utilizados internamente na PPU para recarga de buffers e registradores internos.



Parece estranho a existência desses espaços vazios no loop de execução da PPU mas eles existem por uma razão: Se a CPU realizar acessos a memória enquanto a PPU está realizando a renderização de um pixel, há uma grande chance dos dados na memória da PPU serem corrompidos, e a imagem na tela ficar quebrada. Por conta disso, é disponibilizado esses períodos vazios onde a PPU apenas espera, permitindo que a CPU escreva nos seus registradores mapeados em memória, e configure o componente.

Além disso, para não haver necessidade do programador ficar contando os ciclos exatos para descobrir quando foi iniciado um VBlank onde ele pode escrever na PPU, toda vez que a PPU atinge a segunda scanline do VBlank, ela envia uma interrupção chamada NMI (Non-maskable interrupt) para a CPU, que então pode ser tratada com as rotinas de atualização de gráficos. A interrupção tem esse nome, pois a flag de Interrupt Disable simplesmente não funciona para ela. O único jeito de desabilitar a NMI é escrevendo em um registrador da PPU que desabilita a geração da interrupção.

Registradores da PPU

- PPUCTRL: Registrador mapeado para o endereço 0x2000 da CPU, que só pode receber escritas (stores)

```

7 bit 0
-----
VPHB SINN
|||| ||||| Base nametable address
|||| || (0 = $2000; 1 = $2400; 2 = $2800; 3 = $2C00)
|||| |+--- VRAM address increment per CPU read/write of PPUDATA
|||| | (0: add 1, going across; 1: add 32, going down)
|||| +---- Sprite pattern table address for 8x8 sprites
|||| | (0: $0000; 1: $1000; ignored in 8x16 mode)
|||| +----- Background pattern table address (0: $0000; 1: $1000)
|||| +----- Sprite size (0: 8x8 pixels; 1: 8x16 pixels - see PPU OAM#Byte 1)
|+----- PPU master/slave select
| (0: read backdrop from EXT pins; 1: output color on EXT pins)
+----- Vblank NMI enable (0: off, 1: on)

```

Serve basicamente para fazer configurações básicas do funcionamento da PPU, definindo a nametable base, tamanho dos sprites, se deve disparar NMI ou não, entre outras configurações que entenderemos melhor em breve.

- PPUMASK: Registrador mapeado para o endereço 0x2001 da CPU, que só pode receber escritas

```

7 bit 0
-----
BGRs bMmG
|||| ||||| Greyscale (0: normal color, 1: greyscale)
|||| ||+-- 1: Show background in leftmost 8 pixels of screen, 0: Hide
|||| |+-- 1: Show sprites in leftmost 8 pixels of screen, 0: Hide
|||| +---- 1: Enable background rendering
|||| +----- 1: Enable sprite rendering
|||| +----- Emphasize red (green on PAL/Dendy)
|+----- Emphasize green (red on PAL/Dendy)
+----- Emphasize blue

```

Serve para configurar a renderização em si, controlando as cores, e habilitando ou desabilitando a renderização de background e sprites.

- PPUSTATUS: Registrador mapeado para o endereço 0x2002 da CPU, que só pode ser lido

```

7 bit 0
-----
VSOx xxxx
||||| |||||
|||+-+-+++- (PPU open bus or 2C05 PPU identifier)
||+----- Sprite overflow flag
|+----- Sprite 0 hit flag
+----- Vblank flag, cleared on read. Unreliable; see below.

```

Alguns jogos fazem a leitura desse registrador para solucionar de forma criativa alguns problemas. Os 5 bits mais baixos efetivamente são inúteis. A flag de Sprite overflow indica se existem mais de 8 sprites sendo renderizados na mesma scanline, o que é útil, dado que a PPU só consegue renderizar no máximo 8 sprites por scanline, assim é possível o programar realizar algo para alternar entre os sprites que deverão ser renderizados, para conseguir mais de 8 por linha. A Flag de Sprite 0 hit indica se um pixel opaco do sprite número 0 está acima de um pixel opaco do background, efetivamente indicando se houve colisão. A flag de Vblank é outra forma de descobrir se a PPU entrou nessas scanlines, mas não é confiável por conta de bugs no hardware.

- OAMADDR: Registrador mapeado para o endereço 0x2003 da CPU, que só pode receber escritas
Nesse registrador o programador escreve um endereço de 8 bits, que será utilizado no registrador OAMDATA a seguir, para ler ou escrever um valor na memória de sprites. Não é recomendado utilizar tanto esse quanto o registrador OAMDATA, pois qualquer escrita em um tempo errado pode corromper a memória de sprites.
- OAMDATA: Registrador mapeado para o endereço 0x2004 da CPU, que pode ser tanto lido quanto escrito
Lê ou escreve um byte no endereço presente em OAMADDR da memória de sprites. Caso seja uma escrita, o registrador de endereço é automaticamente incrementando, caso seja uma leitura, ele se mantém igual. Novamente, não é recomendado realizar muitas escritas nesse registrador.
- PPUSCROLL: Registrador mapeado para o endereço 0x2005 da CPU, que pode apenas receber escritas

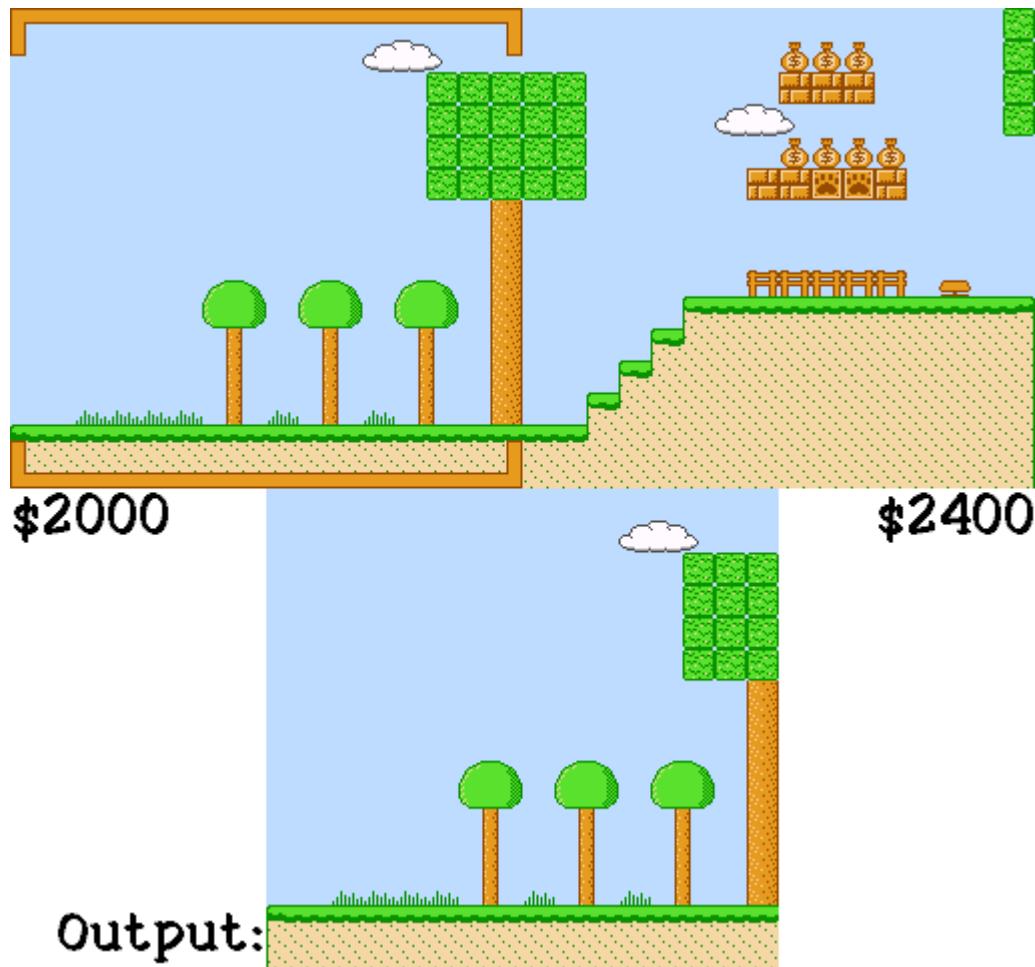
```

1st write
7 bit 0
-----
XXXX XXXX
|||| ||||
++++-+--- X scroll bits 7-0 (bit 8 in PPUCTRL bit 0)

2nd write
7 bit 0
-----
YYYY YYYY
|||| ||||
++++-+--- Y scroll bits 7-0 (bit 8 in PPUCTRL bit 1)

```

Esse registrador é utilizado para controlar o scroll da tela:



O valor de 8bits escrito basicamente indica qual o pixel da tabela principal devemos iniciar na extremidade esquerda (X scroll) ou superior (Y scroll), efetivamente movendo o background para direita ou para baixo, quanto maior for esse valor. Mais detalhes de como isso será feito será detalhado mais à frente. Além disso, esse é um registrador

que é escrito em duas fases, quase como uma serial, onde a primeira escrita vai ser gravada em X scroll, e a escrita subsequente, em Y Scroll.

- PPUADDR: Registrador mapeado para o endereço 0x2006 da CPU, podendo apenas receber escritas

É análogo ao registrador OAMADDR, mas agora o endereço é utilizado para endereçar a memória de vídeo da PPU, podendo ser utilizado para gravar nas nametables, atributos ou até mudar as paletas de cor. Assim como PPUSCROLL, esse registrador também é escrito em duas fases, sendo a primeira escrita os 8 bits mais significativos do endereço, e a segunda escrita os 8 bits menos significativos.

- PPUDATA: Registrador mapeado para o endereço 0x2007 da CPU, podendo tanto ser lido quanto escrito

Análogo ao registrador OAMDATA, utilizado para escrever ou ler valores da memória de vídeo. Novamente, caso ocorra uma escrita, o valor do endereço é incrementado em 1, ou em 32, dependendo do valor escrito em PPUCTRL. Ao ler um valor que não seja a paleta, a primeira leitura sempre retorna o valor da leitura anterior, para realmente obter o valor que precisamos, é necessário ler esse registrador 2 vezes.

- OAMDMA: Registrador mapeado para o endereço 0x4014 da CPU, podendo apenas receber escritas.

Ele é utilizado como forma mais confiável de realizar escritas na memória de sprites. Seu funcionamento é simples, o valor de 8 bits escrito é utilizado como seletor de página, ou seja, os 8 bits mais significativos do endereço da CPU. Ele então bloqueia a execução da CPU, e copia o valor da página inteira da memória da CPU, para a memória de sprites da PPU, efetivamente realizando o Direct Memory Access (DMA) da CPU diretamente na PPU.

Entretanto, nem todos esses são registradores de verdade, alguns são apenas caminhos que mapeiam para registradores internos, e alguns até mapeiam na mesma posição dos registradores internos:

- v: registrador interno utilizado pela PPU como endereço atual da VRAM durante a renderização e também para escritas e leituras da CPU. Quando dizemos que o endereço é incrementado, na verdade é esse registrador que está sendo incrementado. Ele possui 15 bits, e só recebe uma escrita quando a segunda escrita no registrador PPUADDR, e quando isso ocorre, recebe o valor do registrador t.

- t: registrador interno temporário, utilizado como valor de reset para o registrador v. Funciona como uma máquina de escrever, quando chegamos ao final da linha, deveremos voltar o cabeçote para o início. O mesmo ocorre aqui, ao finalizar um frame ou scanline, devemos voltar o valor de v para o valor de t. Esse registrador tem 15 bits e recebe escritas de PPUADDR, PPUSCROLL e PPUCTRL

Ambos os registradores, t e v podem ser interpretados como na imagem abaixo:

```

yyy NN YYYYYY XXXXX
||| || |||| +++++-- coarse X scroll
||| || +++++----- coarse Y scroll
||| +-+----- nametable select
+++-+----- fine Y scroll

```

Nesses registradores, fine Y scroll é utilizado para obter, dentro de um tile, qual a coordenada y do pixel, o nametable select seleciona uma das 4 nametables, e os coarse X e coarse Y scroll são utilizados para selecionar qual tile, sendo de 0 a 31 para X e 0 a 29 para Y.

- x: registrador interno de 3 bits que representa o scroll X fino, basicamente os 4 bits menos significativos, que indicam qual pixel dentro de um tile deve começar o scroll.
- w: registrador interno de 1 bit, que indica se é a primeira ou segunda escrita tanto em PPUSCROLL quanto PPUADDR, sendo compartilhado entre os dois.

Aqui vão algumas prints da wiki de como e quando eles são atualizados:

\$2000 (PPUCTRL) write

```

t: ...GH.. .... <- d: .....GH
<used elsewhere> <- d: ABCDEF..

```

\$2002 (PPUSTATUS) read

```

w:           <- 0

```

\$2005 (PPUSCROLL) first write (w is 0)

```

t: ..... .ABCDE <- d: ABCDE...
x:           FGH <- d: .....FGH
w:           <- 1

```

\$2005 (PPUSCROLL) second write (*w* is 1)

```
t: FGH..AB CDE..... <- d: ABCDEFGH  
w:                      <- 0
```

\$2006 (PPUADDR) first write (*w* is 0)

```
t: .CDEFGH ..... <- d: ..CDEFGH  
    <unused>      <- d: AB.....  
t: Z..... ..... <- 0 (bit Z is cleared)  
w:                      <- 1
```

\$2006 (PPUADDR) second write (*w* is 1)

```
t: ..... ABCDEFGH <- d: ABCDEFGH  
v: <...all bits...> <- t: <...all bits...>  
w:                      <- 0
```

e aqui como acontece a cópia de *t* para *v* quando temos de "repositionar o cabeçote"

At dot 257 of each scanline

If rendering is enabled, the PPU copies all bits related to horizontal position from *t* to *v*:

```
v: ....A....BCDEF <- t: ....A....BCDEF
```

During dots 280 to 304 of the pre-render scanline (end of vblank)

If rendering is enabled, at the end of vblank, shortly after the horizontal bits are copied from *t* to *v*, the full initialization of *v* from *t*:

```
v: GHIA.BC DEF..... <- t: GHIA.BC DEF.....
```

Memória da PPU

A PPU é um componente mais complexo, e por mais que seu funcionamento seja controlado pela CPU, ela ainda possui sua própria memória, totalmente independente da memória da CPU. Esse é o mapa de memória de vídeo da PPU:

Address range	Size	Description
\$0000-\$0FFF	\$1000	Pattern table 0
\$1000-\$1FFF	\$1000	Pattern table 1
\$2000-\$23BF	\$03c0	Nametable 0
\$23C0-\$23FF	\$0040	Attribute table 0
\$2400-\$27BF	\$03c0	Nametable 1
\$27C0-\$27FF	\$0040	Attribute table 1
\$2800-\$2BBF	\$03c0	Nametable 2
\$2BC0-\$2BFF	\$0040	Attribute table 2
\$2C00-\$2FBF	\$03c0	Nametable 3
\$2FC0-\$2FFF	\$0040	Attribute table 3
\$3000-\$3EFF	\$0F00	Unused
\$3F00-\$3F1F	\$0020	Palette RAM indexes
\$3F20-\$3FFF	\$00E0	Mirrors of \$3F00-\$3F1F

Os endereços de 0x0000-0x1FFF contém as duas Pattern Tables. Esses endereços geralmente são mapeados na memória ROM do cartucho, a chamada CHR ROM. As Pattern Tables nada mais são do que as tabelas contendo todos os tiles que são utilizados no jogo, que são carregadas nas nametables para serem renderizadas. A imagem a seguir mostra as duas Pattern Tables do primeiro Mario:



Cada um desses tiles 8x8 é armazenado de uma forma compacta em 16 bytes consecutivos,

onde os 8 primeiros bytes simbolizam o bit baixo seletor da paleta, e os 8 próximos o bit alto. Aqui vai uma figura que deve esclarecer melhor como funciona:

Memory address

Memory address	0x0000	0x0008
0x0000	1 1 1 0 0 0 0 0	0 0 0 0 0 0 0 0
0x0001	1 1 0 0 0 0 0 0	0 0 1 0 0 0 0 0
0x0002	1 0 0 0 0 0 0 0	0 1 1 0 0 0 0 0
0x0003	1 1 1 1 1 1 0 0	0 0 0 0 0 0 0 0
0x0004	1 0 0 0 0 0 0 0	1 1 1 1 0 0 0 0
0x0005	1 1 0 0 0 0 0 0	1 1 1 1 1 1 0 0
0x0006	0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 0
0x0007	0 0 1 0 0 0 0 0	1 1 1 1 1 1 1 0

01	01	01	00	00	00	00	00
01	01	10	00	00	00	00	00
01	10	10	00	00	00	00	00
01	01	01	01	01	01	00	00
11	10	10	10	00	00	00	00
11	11	10	10	10	10	00	00
10	10	10	10	10	10	10	00
10	10	11	10	10	10	10	00

Cada um desses valores de 2 bits é utilizado para selecionar uma cor da paleta, que é definida na attribute table que será vista em breve. Note que um valor de 00 representa transparente, ou seja, caso seja um sprite, renderize o pixel do background no lugar desse pixel.

Nos endereços de 0x3F00-0x3F1F temos a memória RAM que representa as paletas de cor. Cada console tem internamente uma tabela de paleta de sistema, que define as cores possíveis de serem geradas (Essa memória é interna, não podemos acessar, nem mexer nela!). Para definirmos uma cor, simplesmente escrevemos um byte nesses endereços de memória, e esse byte vai equivaler a uma cor da paleta de sistema. A imagem a seguir é um

exemplo de uma paleta de sistema, sendo que os nintendinhos podem ter paletas diferentes, dependendo da origem:

\$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

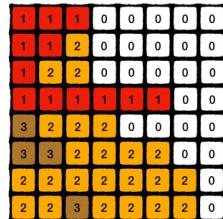
A memória de paleta é composta por 32 bytes, sendo os 16 primeiros utilizados como paletas de background, e os 16 últimos como paletas de sprite. Dentre esses 16 bytes são definidas 4 paletas, sendo assim temos no total 4 paletas de background e 4 paletas de sprite distintas, que são selecionadas pela attribute table ou attribute byte, que veremos a seguir. Além disso, cada uma das paletas tem uma cor neutra em seu primeiro byte, que simplesmente é ligada na cor de backdrop universal em caso de backgrounds, ou na cor do background caso seja um sprite, provendo o efeito de cor transparente. Aqui vai mais uma imagem para deixar mais claro como são organizadas as paletas:

Palette Table

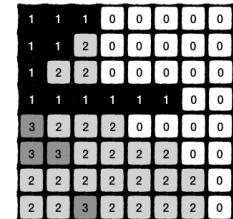
Address	Purpose	Value
3F00	Universal background color	【 20 】
0x3F01 .. 0x3F03	Background palette 0	16, 28, 18
0x3F05 .. 0x3F07	Background palette 1	1E, 10, 00
0x3F09 .. x3F0B	Background palette 2	
0x3F0D .. 0x3F0F	Background palette 3	

NES System Colors

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



BG Palette 0

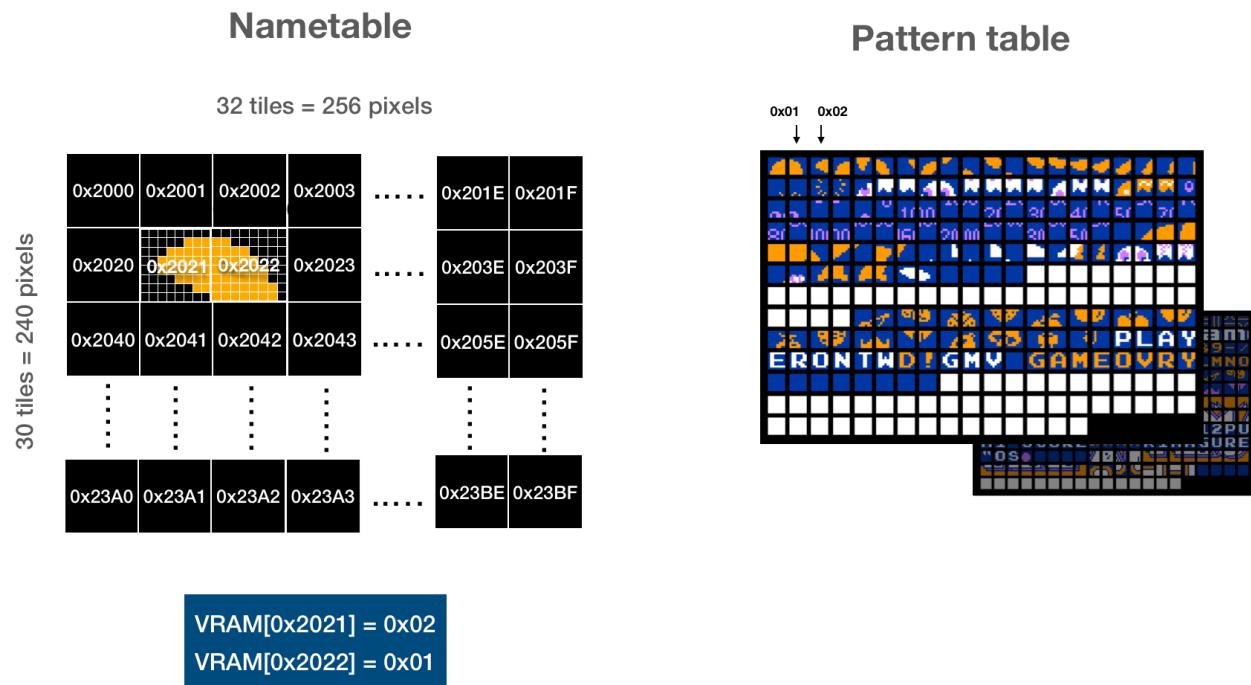


BG Palette 1

Em resumo, cada tile tem para cada pixel, um valor de 2 bits, que é utilizado para selecionar a cor dentro de uma das paletas. A paleta que deve ser usada é definida pela attribute table para os background, ou attribute byte para um sprite.

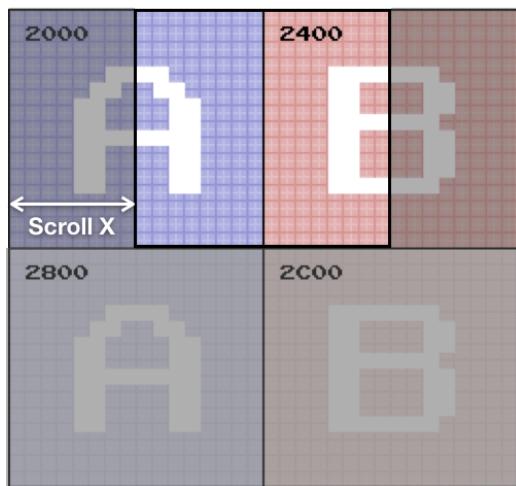
Nos endereços de 0x2000-0x3EFF temos a memória de vídeo VRAM propriamente dita. Ela é composta por 4 nametables, cada uma com sua attribute table. Cada nametable possui 960 bytes, e é nela que gravamos as posições dos tiles que serão renderizados na tela. Temos 960 bytes, pois temos 32 tiles de largura por 30 tiles de altura, ou seja, precisamos de no mínimo $32 \times 30 = 960$ bytes para conseguir armazenar as informações de um frame inteiro. Cada

endereço da nametable corresponde a um tile, o endereço 0x2000 corresponde ao tile cima-esquerdo da tela, o 0x2001 o tile a direita deste, e assim por diante. Cada byte endereça um tile correspondente da pattern table selecionada (a pattern table é selecionada individualmente para sprite e background pelo valor escrito em PPUCTRL). Aqui vai uma imagem novamente para ajudar a entender:

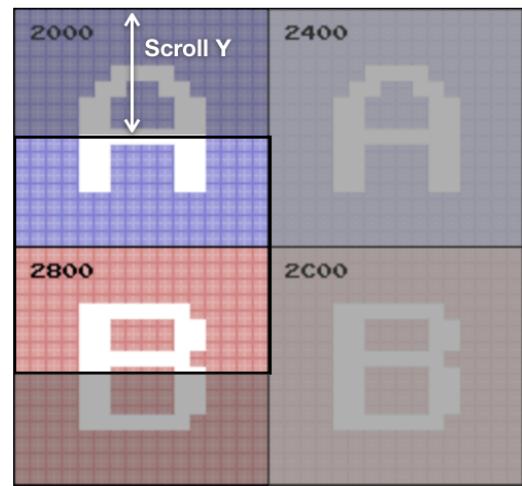


Além disso, podemos notar que temos 4 nametables, mas só temos uma tela na nossa televisão, qual seria o motivo disso? A resposta é simples: para ser possível implementar a rolagem da tela. Na verdade, quase nunca às 4 nametables são utilizadas individualmente, na grande maioria dos jogos, os cartuchos tem um circuito especial que determina um tipo de mirroring. Desta forma, teremos na maioria das vezes, apenas 2 nametables, e 2 espelhos dessas nametables:

Vertical Mirroring



Horizontal Mirroring



Horizontal Scroll

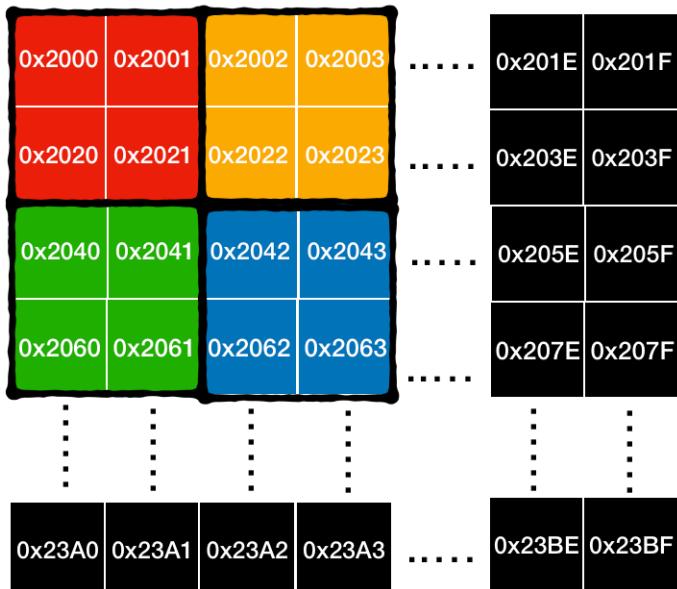
Dessa forma é possível simplificar a lógica da PPU, de forma que para scrollar, ela vá apenas incrementando indefinidamente os endereços, deslizando sobre até a última nametable, de onde ele dá a volta, e retorna para a primeira. Na imagem acima, temos o exemplo dos dois tipos de mirroring mais utilizados: vertical e horizontal. No vertical espelhamos os endereços de 0x2800-0xBFFF para 0x2000-0x23FF e os endereços de 0xC00-2FFF para 0x2400-0x27FF. A informação do tipo de mirroring vem especificada no formato iNES, ou especificada em tempo de execução pelo programador, em alguns cartuchos especiais.

Entretanto, a nametable é só uma parte do quebra-cabeça, já que define apenas qual Tile será usado, mas ainda faltam as informações sobre sua cor. É aí que entram os 64 bytes utilizados para definir a attribute table. Ela é responsável por associar, de forma extremamente comprimida, cada tile a uma paleta de cores. Cada byte da attribute table é responsável por atribuir cor a uma "metatile" de 32x32 (composta por 16 tiles), sendo que cada dois bits desse byte, selecionam a paleta de cada um dos quadrantes dessa "metatile".

Vertical Scroll

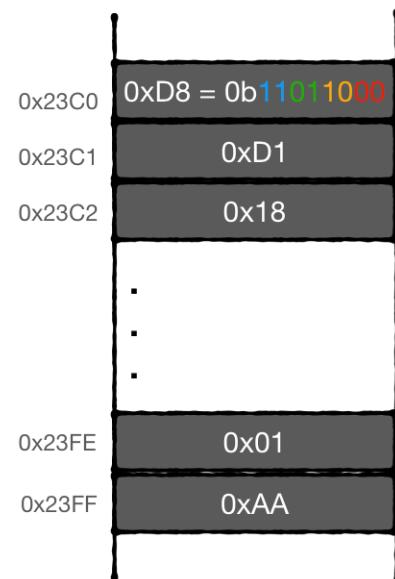
Meta tiles - Blocks

0b11011000



- 0b00 - bg palette 0
- 0b01 - bg palette 1
- 0b10 - bg palette 2
- 0b11 - bg palette 3

Attribute table



Além dessa memória, temos também a memória RAM Object Attribute Memory, chamada OAM. Ela é uma memória de 256 bytes, e sua função é armazenar dados sobre sprites. Cada sprite ocupa 4 bytes, e eles representam, respectivamente: posição Y do primeiro pixel do tile, endereço do Tile na CHR ROM, atributos e posição X do primeiro pixel do tile. Irei detalhar apenas o byte de atributos, pois é o mais importante, e é diferente de como é feito para o background.

```
76543210
|||||||
|||||++- Palette (4 to 7) of sprite
|||+---- Unimplemented (read 0)
||+---- Priority (0: in front of background; 1: behind background)
|+---- Flip sprite horizontally
+---- Flip sprite vertically
```

Nesse byte já temos de cara o valor da paleta, sem precisar fazer conversões complicadas como na attribute table, além da prioridade do sprite, e se devemos espelhar verticalmente ou horizontalmente os sprites. Para renderizar sprites, também há uma flag na PPUCTRL que

controla o tamanho dos sprites, podendo ser 8x8 ou 8x16. Quando o modo 8x16 está ativado, são utilizados 2 tiles, o tile endereçado no byte 2 do OAM, e o tile seguinte, que é colocado abaixo do primeiro tile. Esse modo é bastante utilizado por vários jogos, pois alguns personagens são grandes e precisam de vários sprites para serem representados, com esse modo é possível economizar no número de sprites.

Sintetizando: A renderização do background é feita a partir das nametables e attribute tables, que são definidas pelo jogo, escrevendo na memória com os registradores PPUADDR e PPUDATA. Cada nametable possui um valor que é associado a um tile na CHR ROM, e cada attribute table associa um grupo de tiles da nametable a uma das 4 paletas de cor, sendo a cor definida pelos 2 bits em cada pixel do tile. Já o dos sprites é feito a partir da memória OAM, em que é verificado para cada sprite se ele deve aparecer em um determinado pixel, e então renderizado a partir das informações presentes na OAM.

Ciclo de execução da PPU

Nesta seção vou detalhar melhor o ciclo de execução da PPU, e como ela utiliza as informações acima para criar cada frame

Lembrando que cada ciclo de clock da PPU, avançamos um dot, e ao chegar no último dot, avançamos uma scanline. Irei marcar cada evento com scanline: x dot: y

scanline: 240 dot: 340:

Ao ligar o console, a PPU inicia nesse estado. Nada acontece aqui, pois se trata da idle scanline, seguida das scanlines de Vblank, onde o intuito é nada acontecer mesmo.

scanline 241 dot: 0:

Aqui é o início oficial do VBlank, então devemos disparar a interrupção NMI caso esteja habilitada, e também setar a flag no PPUSTATUS

scanline: 261 dot 0:

Aqui chegamos na última scanline antes de começarmos a renderizar de fato. Essa é a chamada pre-render scanline, onde são preparados alguns buffers internos da PPU. Durante os dots de 280 a 304, recarregamos alguns bits da posição Y do registrador v com bits do registrador t. Além disso, limpamos a flag de VBlank e a flag de sprite zero hit, caso estivesse ativa.

scanline: 0 dot: 0:

Chegamos na primeira scanline visível. Aqui, o primeiro dot é apenas um ciclo de idle, utilizado para preparar novamente os buffers internos.

scanline: 0-239 dot: 1 - 256:

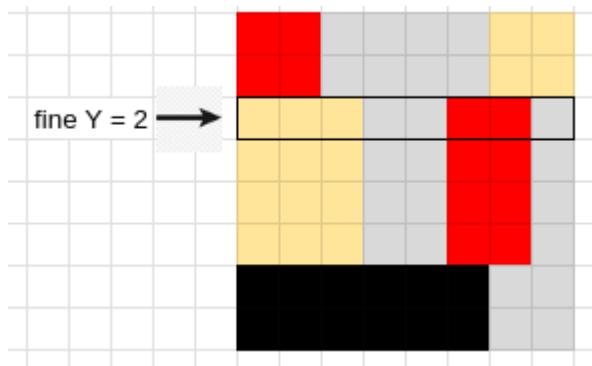
Nesses dots, irá ocorrer a renderização dos pixels. Para fim de entendimento, irei simplificar o buffer de renderização como uma lista de pixels. Essa lista tem 16 pixels, e a cada ciclo, um pixel é escolhido para ser renderizado e a lista é shiftada para esquerda. A cada 8 ciclos, são inseridos mais 8 pixels ao final dessa lista, tornando a lista completa novamente. Note que esse buffer já deve vir pré-carregado pela scanline anterior.

Para preencher esse buffer com os 8 pixels, a ppu faz as seguintes operações:

- Obtém os endereços do tile e do atributo desse tile a partir das seguintes fórmulas utilizando o registrador v:

```
tile address      = 0x2000 | (v & 0xFFFF)
attribute address = 0x23C0 | (v & 0x0C00) | ((v >> 4) & 0x38) | ((v >> 2) & 0x07)
```

- Utiliza esses endereços para ler da memória VRAM os valores da nametable e da attribute table
- Com o valor da nametable, endereça a pattern table, obtém o tile, e desse tile extrai uma linha horizontal de 8 bits de acordo com o valor de fine Y



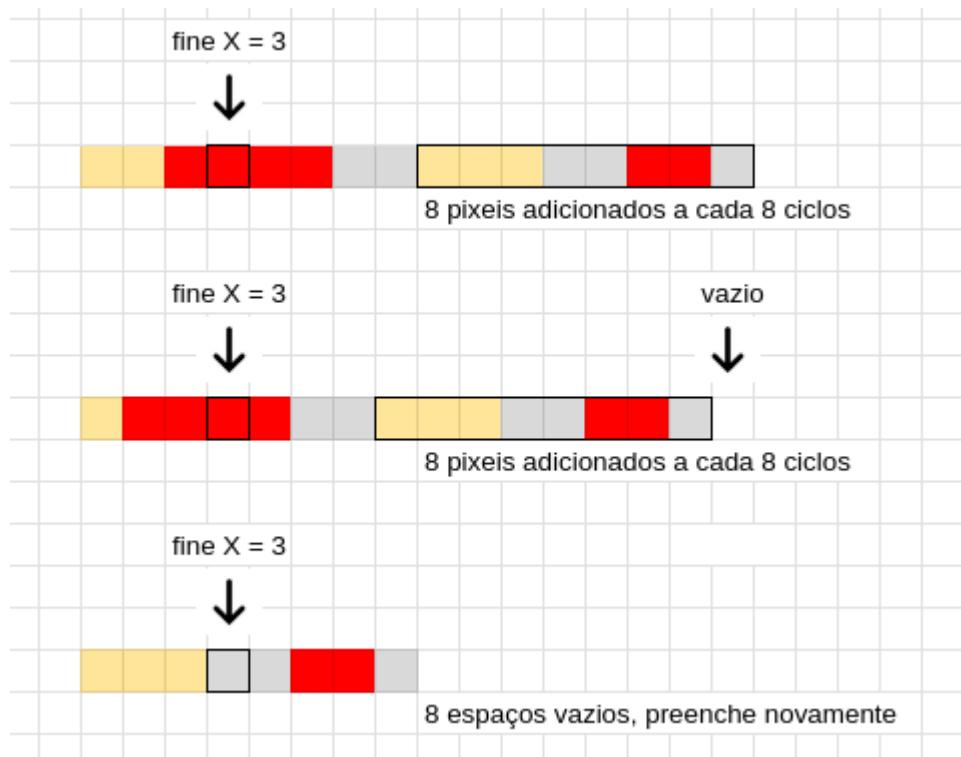
- Em seguida, utiliza o atributo para definir a paleta e quais serão as cores desses pixels. Com os pixels do background definidos, devemos verificar agora os sprites, para ver se algum pixel será sobreposto
- Agora, caso a scanline seja > 0 (não são renderizados sprites na primeira scanline, o que resulta em todos os sprites estarem sempre 1 pixel abaixo do que deveriam estar, bug/limitação de hardware novamente), são obtidos todos os sprites presentes nessa região, bastando realizar comparações entre a posição X do sprite e o dot atual, e a posição Y do sprite com a scanline atual.
- Para cada sprite, utilizamos o seu segundo byte para buscar o tile correspondente, aplicamos todos os espelhamentos presentes no byte de atributo, e novamente utilizamos o valor de fine Y para buscar uma linha horizontal de pixels.
- Agora, para cada pixel do sprite, caso seja transparente, simplesmente ignoramos. Caso o pixel do sprite seja opaco e o background seja transparente, substituimos o pixel

do background pelo do sprite. Caso ambos sejam opacos, verificamos o bit de prioridade no byte de atributo. Se o sprite tiver prioridade sobre background, substituímos o pixel, caso não tenha, ignoramos.

- Também devemos verificar a existência de overflow ou da colisão com sprite 0, setando as respectivas flags.

Ao fim de tudo isso, incrementamos o endereço do registrador v, fazendo a devida troca de nametables caso necessário, para permitir o scrolling.

Para escolher qual pixel da lista de 16 pixels será enviado para o televisor, utilizamos o valor do register fine X da seguinte forma:



scanline: 0-239 dot: 256 e 257:

Nestes dots, acabamos de finalizar a renderização de uma linha de pixels, então incrementamos o fine Y, para descer para a próxima linha do pixel, e caso o fine Y passe de 7, aumentamos o coarse Y e voltamos o fine para 0. Em seguida, devemos voltar o valor da posição X no registrador v, para iniciarmos no pixel da esquerda, então copiamos os valores que estavam no registrador t. É por esse motivo que escritas no meio da renderização causam problemas, pois o valor carregado no registrador t e v são constantemente utilizados, e se não forem corretos, causarão problemas.

scanline 0-239 dot: 327-335:

Nesses dots, devemos pré carregar o buffer com os dois primeiros tiles, isso é feito da mesma forma que nos ciclos visíveis: carregamos 8 pixels no final da lista, shiftamos a lista 8 vezes, e carregamos mais 8 pixels.

scanline 240 dot: qualquer:

Voltamos ao início, e o loop é feito até que o console seja retirado da tomada.

Emulação

Para emular a PPU, vamos começar fazendo uma boa fundação. A primeira coisa a ser feita, é registrar a quantidade de ciclos de Clock que cada instrução da CPU leva. Isso é bastante importante, pois no hardware, tanto a CPU quanto a PPU executam em paralelo, e fazem suposições sobre o estado um do outro, ou seja, é uma arquitetura tão velha que o hardware é determinístico, e conseguimos saber sem sombra de dúvidas qual o estado de todos os componentes depois ou antes da execução de uma instrução. Dessa forma, precisamos garantir que os dois componentes estejam pelo menos próximos um do outro no quesito temporização.

Para fazer isso, basta recorrer aos documentos sobre as instruções de CPU novamente e fazer o tedioso processo de mapear cada instrução e modo de endereçamento ao seu número de ciclos. Com isso feito, podemos confirmar nossa implementação novamente utilizando a coluna CYC do arquivo nestest.

Assim podemos adaptar nosso loop principal, onde executamos uma instrução de CPU, e depois executamos a PPU pelo triplo do número de ciclos que essa instrução levou. Essa estratégia não é a mais precisa, dado que os componentes não estão executando em paralelo, mas em meus testes, não tive nenhum problema, provavelmente por conta dos atrasos serem curtos o suficiente para não estragar a execução.

Com isso feito, podemos partir para a PPU em si. O primeiro passo é o básico, criamos um novo struct para representar a PPU e seus registradores:

```
type Ppu struct {
    // register mapped to cpu memory
    ppuCtrl     uint8
    ppuMask     uint8
    ppuStatus   uint8
    ppuOamAddr  uint8
    ppuOamData  uint8

    ppuSpriteEnabled bool
    ppuBackgroundEnabled bool
    // internal buffers
    readBuffer uint8
    // simulation clock cycles and scanlines
    cycles      uint
    scanlines   uint
    // nmi interrupt signal
    NmiInterrupt bool
}
```

```

// color palette
systemPalette [64][3]uint8
// frame control
CurrentFrame      Frame
    // sprites
spriteLine        [8][8]uint8
spritePosition    [8]uint8
spritePalette    [8][4][3]uint8
spriteNumber     [8]uint8
spritePriority   [8]uint8
spriteCount      uint
// ppu internal registers
loopyV  uint16
loopyT  uint16
fineX   uint8
write   uint8
// output buffers
currentPixel  uint
outputBackgroundRgb [][][3]uint8
outputBackgroundVal []uint8
}

```

Em seguida, criamos um struct para representar a memória da PPU, e já fazemos também os mapeamentos para o cartucho e as configurações de mirroring da nametable:

```

type Memory struct {
    vram      [0x0800]uint8
    rom       cartridge.Cartridge
    paletteRam [0x0100]uint8
    oam       [0x0100]uint8
}

var PpuMemory Memory

func LoadFromCartridge(cartridge cartridge.Cartridge) {
    PpuMemory.rom = cartridge
}

func PpuMemRead(addr uint16) uint8 {
    addr = addr % 0x4000
    if addr <= 0x1FFF {
        return readChrRom(addr)
    } else if addr >= 0x2000 && addr <= 0x3EFF {

```

```

        return PpuMemory.vram[mirrorVramAddress(addr)]
    } else if addr >= 0x3F00 {
        addr = (addr - 0x3F00) % 0x20
        return PpuMemory.paletteRam[addr]
    }
    return 0
}
// .... implementa write, Leitura e escrita em oam, Leitura de tile
func mirrorVramAddress(addr uint16) uint16 {
    // mirrors the unused memory to the vram
    if addr >= 0x3000 {
        addr -= 0x1000
    }
    // performs mirroing according to cartridge info
    switch PpuMemory.rom.MirroringType {

        case cartridge.HorizontalMirroring:
            // HORIZONTAL Mirroring
            //      [ A ] [ A ]
            //      [ B ] [ B ]
            if addr >= 0x2000 && addr <= 0x23FF {
                addr -= 0x2000
            } else if addr >= 0x2400 && addr <= 0x27FF {
                addr -= 0x2400
            } else if addr >= 0x2800 && addr <= 0x2BFF {
                addr -= 0x2400
            } else {
                addr -= 0x2800
            }

        case cartridge.VerticalMirroring:
            // Vertical Mirroring
            //      [ A ] [ B ]
            //      [ A ] [ B ]
            if addr >= 0x2000 && addr <= 0x23FF || addr >= 0x2400 && addr
<= 0x27FF {
                addr -= 0x2000
            } else {
                addr -= 0x2800
            }

        case cartridge.FourScreenMirroring:
            // Four Screen Mirroring

```

```

        //      [ A ] [ B ]
        //      [ C ] [ D ]
        addr -= 0x2000
    }
    return addr
}

```

Com a memória em mãos, podemos implementar também uma função para carregar uma paleta de sistema a partir de um arquivo .pal presente na wiki. A implementação é bastante trivial e foge do escopo, podendo ser consultada no repositório: [Repositório](#)

O último struct que decidi definir foi o struct para representar um frame, que nada mais é que um struct com uma grande lista de inteiros, onde cada 3 inteiros consecutivos representam um pixel no formato RGB:

```

type Frame struct {
    PixelData [XSIZE * YSIZE * 3]uint8
}

func NewFrame() *Frame {
    var frame Frame
    return &frame
}

func (frame *Frame) GetPixelData() []uint8 {
    return frame.PixelData[:]
}

func (frame *Frame) setPixel(x uint, y uint, rgb [3]uint8) {
    address := x*3 + y*3*XSIZE
    copy(frame.PixelData[address:], rgb[:])
}

```

Agora podemos iniciar a implementação da PPU em si. Vamos iniciar codificando todos os registradores mapeados em memória. Para cada um, foi criada um método de struct, que será chamado toda vez que a CPU escrever em seus respectivos endereços:

```

// ----- PPUCTRL 0x2000 REGISTER -----

func (ppu *Ppu) WriteToPpuControl(val uint8) {
    // comportamento quando ativamos NMI no meio de um vblank
    if (val>>7)&0b1 == 1 && (ppu.ppuCtrl>>7)&0b1 == 0 &&
(ppu.ppuStatus>>7)&0b1 == 1 {
        ppu.NmiInterrupt = true
}

```

```

    }
    ppu.ppuCtrl = val
    // t: ...GH.. .... <- d: .....GH
    ppu.loopyT = SetBitToVal(ppu.loopyT, 10, val&0b1)
    ppu.loopyT = SetBitToVal(ppu.loopyT, 11, (val>>1)&0b1)
}

```

Na memória da CPU:

```

func MemWrite(addr uint16, val uint8) {
    switch {
        // cpu RAM
        case addr <= 0x07FF:
            if debug {
                modified.ram[addr] = 1
            }
            MainMemory.ram[addr] = val
        // ppu registers mapped to cpu memory
        case addr >= 0x2000 && addr <= 0x3FFF:
            addr = ((addr - 0x2000) % 0x0008) + 0x2000
            switch addr {
                case PPUCTRL:
                    ppu.GetPpu().WriteToPpuControl(val)
                case PPUMASK:
                    ppu.GetPpu().WriteToPpuMask(val)
                case PPUSCROLL:
                    ppu.GetPpu().WriteToPpuScroll(val)
                case OAMADDR:
                    ppu.GetPpu().WriteToOamAddrRegister(val)
                case OAMDATA:
                    ppu.GetPpu().WriteToOamDataRegister(val)
                case PPUADDR:
                    ppu.GetPpu().WriteToAddrRegister(val)
                case PPUDATA:
                    ppu.GetPpu().WriteToPpuDataRegister(val)
            }
        // OAMDMA, using interrupt
        case addr == OAMDMA:
            MainMemory.OamDmaInterrupt = true
            MainMemory.OamDmaPage = val
        // prg rom
        case addr >= 0x8000:
            fmt.Println("Warning: cant write to ROM")
    }
}

```

```

        return
    }
}

```

A implementação dos outros registradores é bastante repetitiva e trivial, podendo também serem consultadas no repositório.

Por fim, basta criar a função que executa um ciclo de PPU, e seguir a risca o roteiro de execução da PPU que foi explicado acima:

```

func Clock() {

    ppu.runCycle()

    visibleScanlines := ppu.scanlines <= 239
    visibleCycles := ppu.cycles >= 1 && ppu.cycles <= 256
    preRenderScanline := ppu.scanlines == 261
    preRenderCopyY := ppu.cycles >= 280 && ppu.cycles <= 304
    vblank := ppu.scanlines == 241 && ppu.cycles == 1
    vblankEnd := preRenderScanline && ppu.cycles == 1
    spriteEvaluate := visibleScanlines && ppu.cycles == 257

    // rendering visible scanlines
    if ppu.ppuBackgroundEnabled {
        if visibleScanlines {
            if visibleCycles {
                // draw pixel from output buffer
                ppu.renderPixel()
                // if cycle is 8, 16, 24 ... reload shift register
                and update LoopyV
                if ppu.cycles%8 == 0 {
                    ppu.reloadBackgroundBuffer()
                    ppu.incrementLoopyVX()
                }
            }
            // increment y -> go down a pixel
            if ppu.cycles == 256 {
                ppu.currentPixel = 0
                ppu.incrementLoopyVY()
            }
            // get scrollx back to start
            if ppu.cycles == 257 {
                ppu.copyScrollxToLoopyV()
            }
        }
    }
}

```

```

// Load first tile
if ppu.cycles == 327 {
    ppu.reloadBackgroundBuffer()
    ppu.incrementLoopyVX()
}
// continua fazendo tudo que precisa fazer...

```

Agora, precisamos de um jeito de emular a interrupção NMI. Eu decidi utilizar o método chamado Polling, onde a CPU, a cada vez que tenta executar uma instrução, chama uma função da PPU, que retorna true caso o NMI esteja ativo, e o desativa. Dessa forma, quando o NMI for gerado, já na próxima instrução ela será executada, e o código desviado para o novo destino. A escrita no registrador OAMDMA será tratada da mesma forma: Quando a memória detecta uma escrita nesse registrador, gera uma interrupção que deve ser tratada na CPU, que então faz as escritas DMA, e avança a quantidade de ciclos necessários.

```

func ExecuteNext() {
    if ppu.GetPpu().PollForNmiInterrupt() {
        cpu.treatNmiInterrupt()
        return
    }
    if memory.PoolOamDmaInterrupt() {
        cpu.OamDmaWrite(memory.MainMemory.OamDmaPage)
        return
    }
// execucao de uma instrucao normal
// ...

// tratamentos interrupcao
func (cpu *Cpu) treatNmiInterrupt() {
    cpu.pushToStack16(cpu.Pc)
    cpu.pushToStack(cpu.Psts & 0b11101111)

    cpu.setFlag(InterruptDisable, 1)
    cpu.LastInstructionCycles = 7
    cpu.Pc = memory.MemRead16(0xFFFF)
}

func (cpu *Cpu) OamDmaWrite(page uint8) {
    for i := range uint16(256) {
        val := memory.MemRead((uint16(page) << 8) + i)
        memory.MemWrite(memory.OAMDATA, val)
    }
    cpu.LastInstructionCycles = 514
}

```

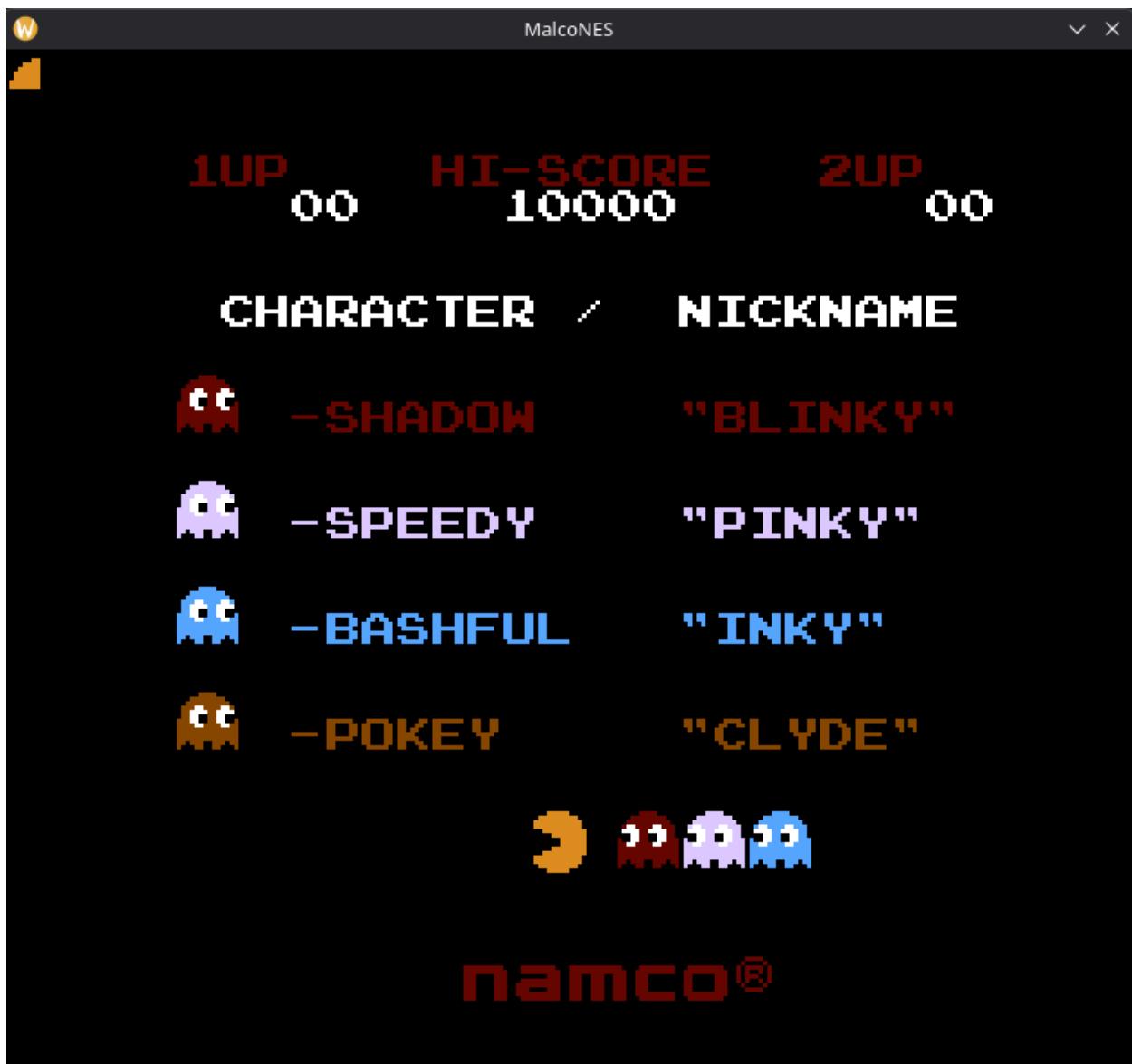
}

Por fim, ainda temos de dar um jeito de pegar esses pixels, e jogar na tela a uma taxa de 60fps. Vale lembrar que o nosso hardware atual é extremamente rápido, e se apenas deixar o emulador rodar a vontade, os jogos iriam estar extremamente acelerados. Para resolver esses dois problemas, irei utilizar a biblioteca ebiten do Go para implementar tanto a criação e envio de pixels para tela, quanto a taxa de atualização constante de 60fps.

O setup e a configuração do ebiten para desenhar pixels na tela não fazem parte do escopo, e pode ser facilmente consultado no repositório. Para conseguir limitar a execução do emulador para 60fps, é utilizada a função Update disponibilizada pelo Ebiten. Essa função é garantida que será executada apenas uma vez a cada 1/60 segundos, assim, cada vez que essa função é chamada, executamos a CPU um total de aproximadamente 29780 ciclos, o que equivale a 89340 ciclos de PPU, que é aproximadamente aos 262 scanlines x 340 dots = 89342 que a PPU leva para gerar um frame. Essa pequena diferença é arredondada pelo fato da CPU levar quantidades de ciclos variados para cada instrução, fazendo com que execute mais de 29780 ciclos algumas vezes.

Diferente da CPU, eu não encontrei nenhum teste comprehensivo da PPU. Na minha opnião, o melhor jeito de testar é simplesmente carregar um jogo e verificar se tudo aparece corretamente na tela. Recomendo iniciar testando com o jogo Pac-Man, dado que é um jogo que não faz uso de cartuchos especiais, e nem de scrolling da tela, sendo o mais simples. Em seguida, podemos testar Super Mario e Ice Climbers, que também são jogos que não utilizam cartuchos especiais, mas que fazem uso de scroll horizontal e vertical, respectivamente. Mesmo que ainda não tenhamos controles configurados, se esperarmos na tela inicial tempo suficiente, esses jogos iniciam uma sequência de demonstração, que faz uso de todas as features da PPU.





Estamos a poucos passos de conseguir jogar de fato os jogos. Para chegarmos em um estado jogável o mais rápido possível, irei deixar a parte de áudio para o último post. Neste post iremos implementar o Controle, que já nos permitirá jogar a maior parte dos jogos mais simples que não fazem uso de cartuchos especiais, e logo em seguida, iremos implementar um dos cartuchos especiais, para aumentar a gama de jogos que podemos jogar.

Controles

O clássico controle do nintendinho conta com 8 botões, os botões de ação A e B, reset, start e os 4 botões para controle direcional.



Como já foi dito anteriormente, o controle não funciona da mesma forma que os dispositivos de entrada e saída que temos atualmente, que geram interrupções ou eventos quanto pressionados. Ele na verdade é um componente totalmente passivo, que é apenas lido pela CPU quando o programador decide, sendo mais parecido com os esquemas antigos de Polling.

Para comunicação com a CPU, temos dois endereços de memória mapeados para os controles, os endereços 0x4016 e 0x4017, um para o controle de cada player. Os controles funcionam de forma simples: Ao realizar a leitura de um dos endereços, o controle retorna um byte, onde o bit menos significativo vai indicar o estado de um botão. Ela funciona de uma forma serial, onde leituras subsequentes retornam os estados de botões na seguinte ordem:

- primeira leitura: Retorna estado do botão A no bit menos significativo (1 se pressionado, 0 se não pressionado)
- segunda leitura: Retorna estado do botão B no bit menos significativo
- terceira leitura: Retorna estado do botão select no bit menos significativo
- ...

e assim por diante, na ordem A, B, select, start, cima, baixo, esquerda, direita. Ao chegar no final das leituras, o controle deve retornar sempre o valor 1. Para reiniciar o ciclo de leituras de volta para o botão A, basta realizar a escrita de um byte com bit menos significativo igual a 1 no endereço 0x4016. Isso irá ativar o modo strobe, e leituras feitas ao controle retornarão apenas o valor de A enquanto ele estiver ativo. Para desativar o modo strobe, basta escrever um byte com o bit menos significativo igual a 0 no mesmo endereço, que o controle estará reiniciado, e as leituras vão ocorrer sequencialmente como explicado acima. Note que o endereço 0x4016 ativa o modo strobe para ambos os controles, não podendo haver escritas no endereço 0x4017.

Emulação

Como podemos ver, o controle é um dos componentes mais simples, bastando apenas codificar seu simples funcionamento interno, realizar o mapeamento no espaço de memória de CPU, e configurar o Ebiten para detectar o aperto de teclas no teclado, e transmitir essas teclas para o controle.

Inicialmente, será criado um struct para representar o controle, juntamente com seus métodos que serão mapeados na memória e para o detector de teclas do Ebiten:

```
type JoyPad struct {
    strobe      bool
    buttonShift uint
    buttonStatus [8]uint
}

func NewJoyPad() *JoyPad {
    var joyPad JoyPad
    return &joyPad
}

func (joyPad *JoyPad) ReceiveWrite(val uint8) {
    if val&0b1 == 1 {
        joyPad.strobe = true
        joyPad.buttonShift = 0
    } else {
        joyPad.strobe = false
    }
}

func (joyPad *JoyPad) ReceiveRead() uint8 {
    if joyPad.buttonShift > 7 {
        return 1
    }
    result := joyPad.buttonStatus[joyPad.buttonShift]
    if !joyPad.strobe {
        joyPad.buttonShift++
    }
    return uint8(result)
}

func (joyPad *JoyPad) SetButtonStatus(button uint, val uint) {
    joyPad.buttonStatus[button] = val
}
```

Mapear na memória é trivial:

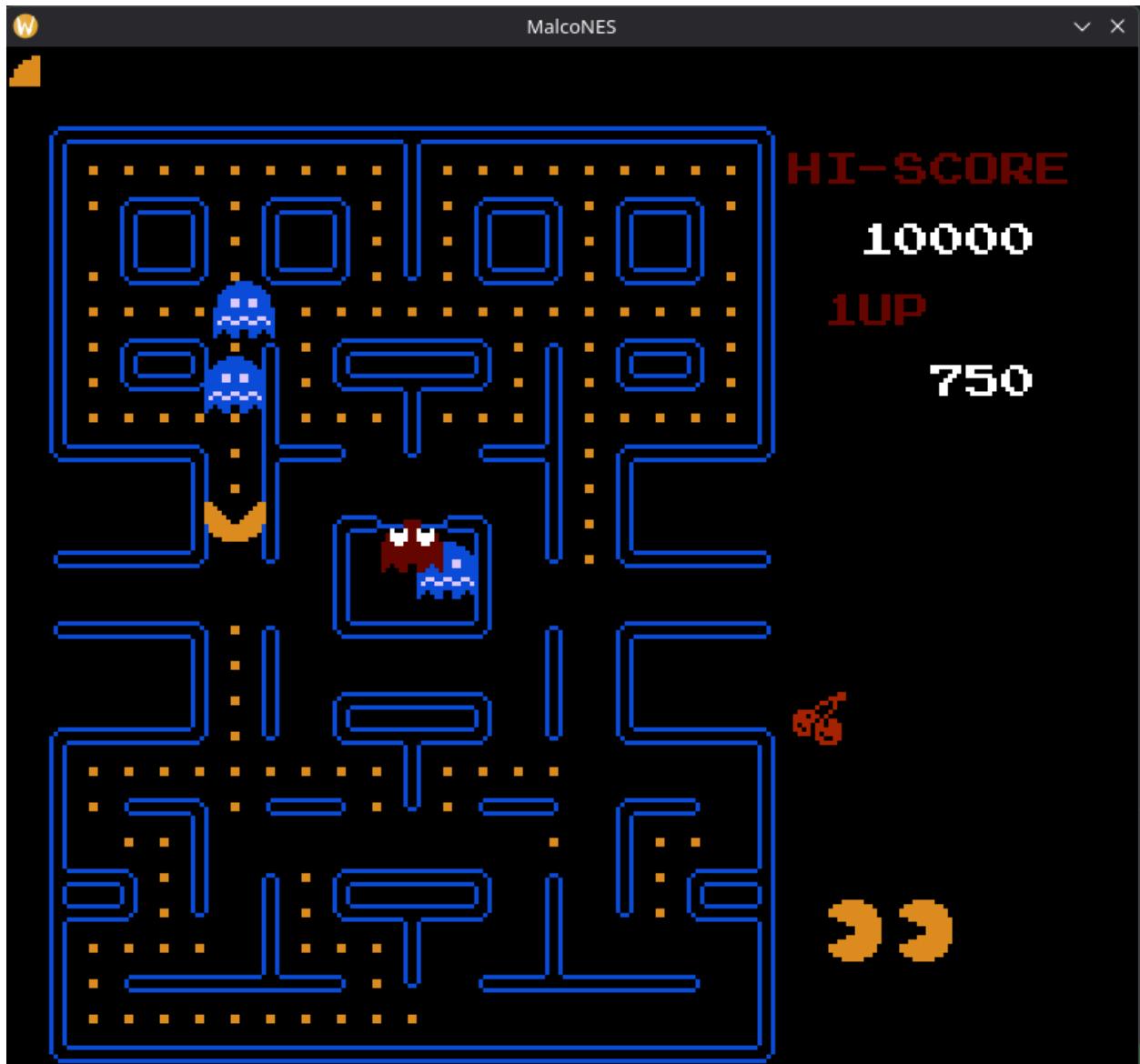
```
// método de Leitura
    case addr == CONTROLLER1:
        return joyPad1.ReceiveRead()
// ....
// método de escrita
    case addr == CONTROLLER1:
        joyPad1.ReceiveWrite(val)
// ....
```

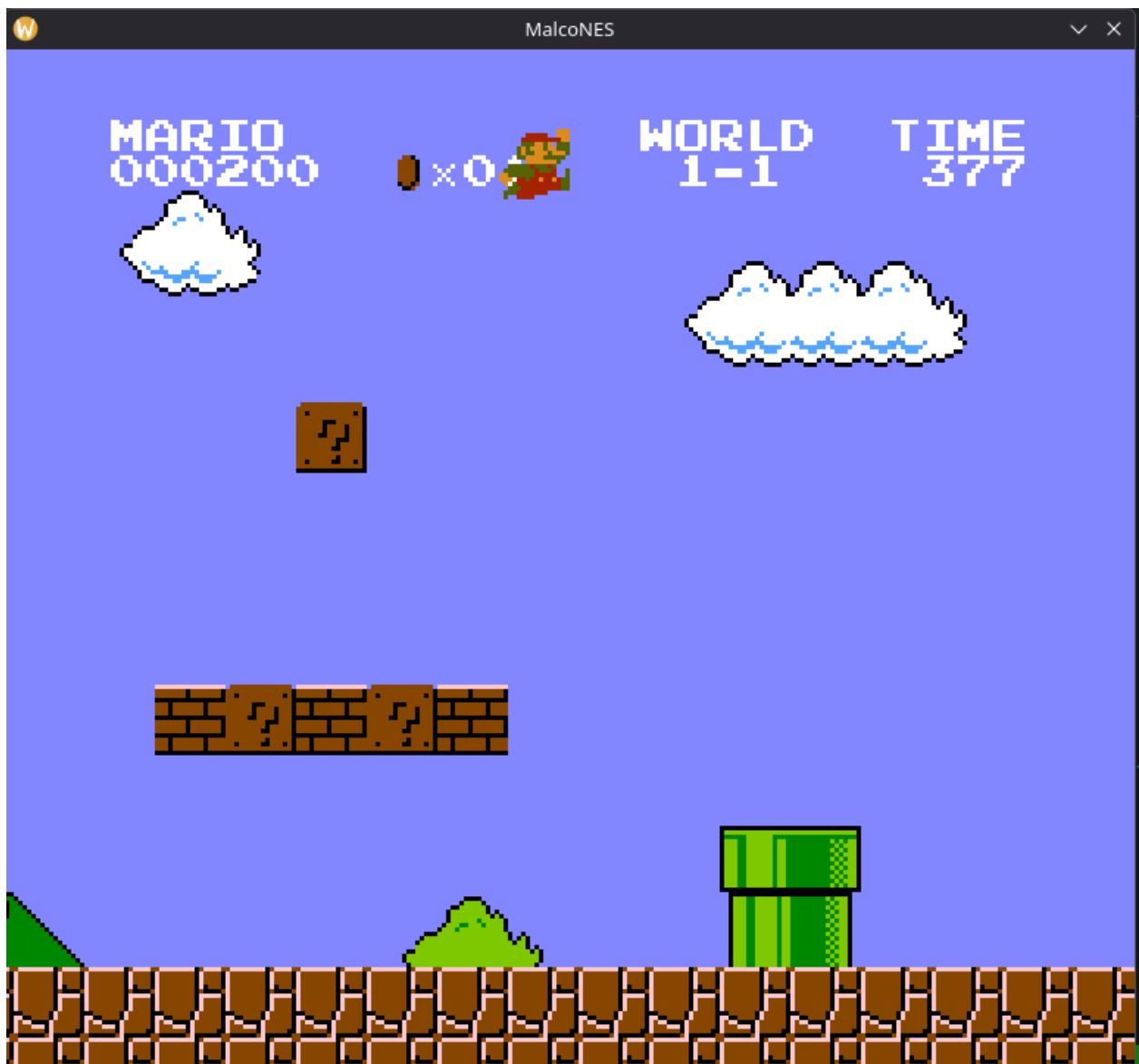
Novamente, fazer a interface com a biblioteca Ebiten é bastante trivial, bastando verificar os eventos de aperto de teclado a cada frame no método Update e atualizar o estado do controle:

```
func handleInput() {
    JoyPad1.SetButtonStatus(controller.LEFT, ifPressed(ebiten.KeyA))
    JoyPad1.SetButtonStatus(controller.DOWN, ifPressed(ebiten.KeyS))
    JoyPad1.SetButtonStatus(controller.RIGHT, ifPressed(ebiten.KeyD))
    JoyPad1.SetButtonStatus(controller.UP, ifPressed(ebiten.KeyW))

    JoyPad1.SetButtonStatus(controller.A, ifPressed(ebiten.KeyJ))
    JoyPad1.SetButtonStatus(controller.B, ifPressed(ebiten.KeyK))
    JoyPad1.SetButtonStatus(controller.START, ifPressed(ebiten.KeySpace))
    JoyPad1.SetButtonStatus(controller.SELECT, ifPressed(ebiten.KeyZ))
}
```

Com isso feito, já é possível jogar PacMan, Super Mario Bros e alguns outros jogos (não vão ter som ainda infelizmente)





Mappers e Cartuchos

Por mais que o nosso emulador já esteja no estágio em que conseguimos jogar alguns, se tentarmos carregar outros jogos, a maior parte deles simplesmente não irá executar e provavelmente irá crashar o emulador. Uma dica para entender o motivo, é olhar para o tamanho dos arquivos .nes de vários jogos:

```
-rw-rw-r-- 1 vitors vitors 129K jun 21 2017 castlevania.nes
-rw-rw-r-- 1 vitors vitors 129K jun 21 2017 contra.nes
-rw-rw-r-- 1 vitors vitors 129K jun 21 2017 ducktales.nes
-rw-rw-r-- 1 vitors vitors 25K jun 21 2017 iceclimber.nes
-rw-r--r-- 1 vitors vitors 257K jul 15 14:20 megaman2.nes
-rw-r--r-- 1 vitors vitors 840K jul 7 19:47 nestest.log
-rw-r--r-- 1 vitors vitors 25K jul 7 19:47 nestest.nes
-rw-rw-r-- 1 vitors vitors 25K jun 21 2017 pacman.nes
-rw-r--r-- 1 vitors vitors 257K jul 15 10:39 supermario2usa.nes
-rw-r--r-- 1 vitors vitors 385K jul 15 10:48 supermario3.nes
-rw-r--r-- 1 vitors vitors 41K jan 21 2000 supermario.nes
-rw-r--r-- 1 vitors vitors 129K jul 15 13:59 thelegendofzelda.nes
```

Podemos ver algumas coisas estranhas, como é possível alguns jogos terem tamanhos de centenas de kB, se o espaço de endereçamento que temos é igual à 32kb da PRG ROM na CPU + 8kB da CHR ROM na PPU = 40Kb? Além disso, vemos que os únicos jogos que conseguimos executar são esses jogos menores, como o Super Mario, Pacman e Ice Climber.

A resposta para as duas questões é uma só: Os cartuchos desses jogos mais complexos possuem circuitos especiais chamados Mappers, que poderiam prover várias funcionalidades a mais para os jogos. Uma dessas funcionalidades, é permitir que o sinal de escrita enviado pela CPU ou pela PPU ao tentar escrever na memória ROM seja utilizado para configurar e trocar os bancos de memória ROM que estão mapeados nos endereços da CPU, efetivamente fazendo uma espécie de Overlay primitivo, já que o programador deve configurar os bancos em si, e quando eles devem entrar. Alguns Mappers também permitem coisas como definir o modo de Mirroring em tempo de execução, gerar interrupções IRQ baseadas no estado da PPU e até adicionar novos canais de áudio.

Existem uma infinidade de Mappers, e alguns jogos tem Mappers especiais que foram feitos somente para eles. Para este projeto, foram implementados apenas alguns dos Mappers mais

comuns que eram padronizados pela própria Nintendo. Nos nossos arquivos iNES, o tipo de Mapper utilizado vem discriminado nos headers. A lista completa de Mappers e seus detalhes podem ser visualizados nesta página da wiki: [Mappers iNES](#)

iNES 1.0 mapper grid

This is the plane 0 table. These mappers do not require a NES 2.0 header.

N 000	N 001	N 002	N 003	N 004	N 005	B 006	R 007	B 008	N 009	N 010	T 011	F 012	HQ 013	F 014	X 015	
BAN 016	E 017	E 018	P 019	X 020	S 021	S 022	S 023	S 024	S 025	S 026	W 027	D 028	D 029	D 030	D 031	
H 032	A 033	H 034	M 035	G 036	N 037	B 038	M 039	W 040	M 041	X 042	X 043	X 044	X 045	T 046	N 047	
A 048	X 049	X 050	X 051	X 052	X 053	B 054	X 055	K 056	X 057	X 058	B 059	X 060	D 061	X 062	W 063	
T 064	R 065	N 066	S 067	S 068	S 069	BAN 070	E 071	M 072	S 073	E 074	S 075	P 076	H 077	R 078	T 079	
A 080	W 081	A 082	C 083	M 084	S 085	E 086	E 087	P 088	S 089	M 090	M 091	E 092	S 093	C 094	P 095	
BAN 096	R 097	E 098	N 099	X 100	M 101	M 102	B 103	E 104	N 105	X 106	B 107	B 108	M 109	M 110	B 111	
W 112	R 113	B 114	E 115	S 116	B 117	N 118	R 119	B 120	E 121	M 122	B 123	X 124	W 125	X 126	X 127	
X 128	M 129	M 130	M 131	M 132	S 133	B 134	M 135	S 136	S 137	S 138	S 139	E 140	S 141	K 142	S 143	
U 144	S 145	M 146	S 147	S 148	S 149	S 150	M 151	BAN 152	E 153	P 154	N 155	BAN 156	BAN 157	T 158	BAN 159	
M 160	M 161	X 162	M 163	E 164	X 165	M 166	M 167	Z 168	X 169	B 170	K 171	E 172	E 173	VID 174	K 175	
E 176	B 177	E 178	M 179	E 180	M 181	X 182	X 183	S 184	N 185	B 186	X 187	E 188	E 189	Z 190	E 191	
E 192	W 193	E 194	E 195	X 196	X 197	E 198	E 199	X 200	X 201	X 202	X 203	X 204	X 205	P 206	A 207	
F 208	M 209	P 210	M 211	X 212	M 213	X 214	RE 215	B 216	X 217	B 218	E 219	X 220	W 221	X 222	M 223	
E 224	X 225	X 226	B 227	E 228	X 229	X 230	X 231	E 232	X 233	E 234	X 235	X 236	X 237	E 238	E 239	
E 240	E 241	E 242	S 243	E 244	E 245	E 246	?	E 247	M 248	E 249	B 250	M 251	E 252	E 253	B 254	X 255

O Mapper padrão, que é o que utilizamos nesses jogos mais simples de menos de 40Kb, é o Mapper 0, que simplesmente não faz nenhuma troca de banco de memória de programa, nem memória gráfica. Para este post, irei explicar o funcionamento e a implementação do Mapper 2, que é um Mapper simples, mas sua lógica de funcionamento aplica para a maior parte dos outros Mappers. No projeto, realizei apenas a implementação dos Mappers 0, 1, 2, 4, que não eram suficientes para executar os jogos que eu queria.

Exemplo Mapper 2

Esse é um Mapper simples, mas que é utilizado em vários jogos famosos, como o primeiro Mega Man, Castlevania etc. Tirando diretamente da Wiki, ele divide os 32kB da memória ROM mapeada na CPU em dois bancos de 16kB, um banco fixo nos últimos endereços, e um banco que pode ser trocado nos 16k primeiros:

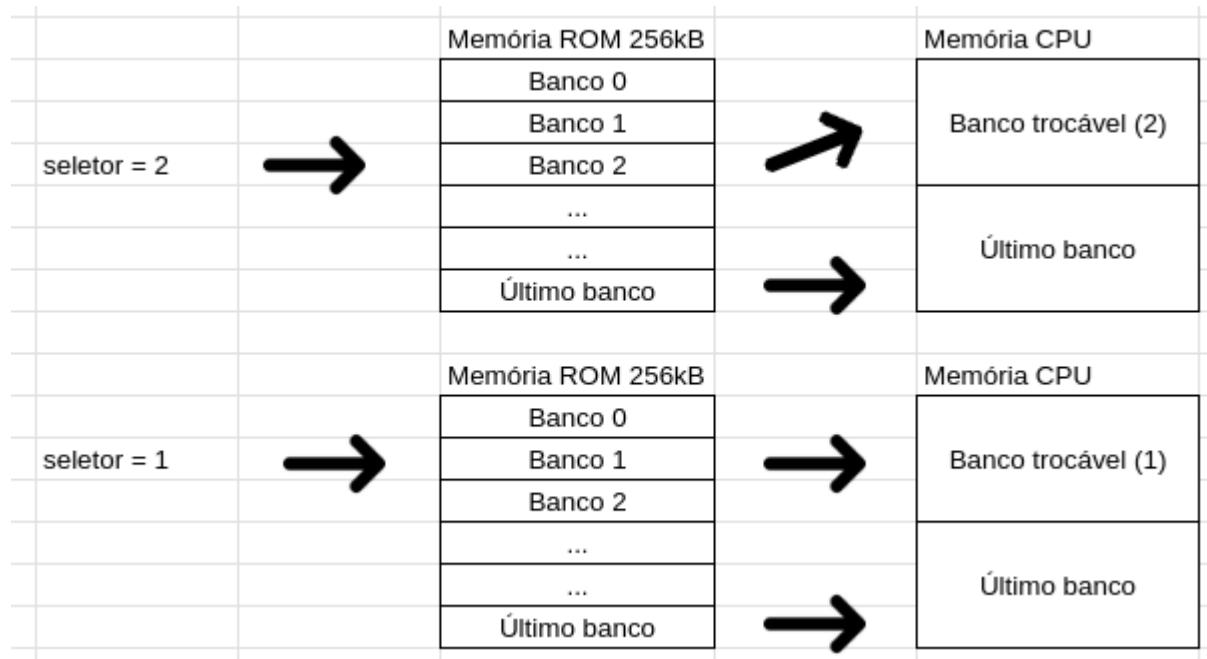
Banks

- CPU \$8000-\$BFFF: 16 KB switchable PRG ROM bank
- CPU \$C000-\$FFFF: 16 KB PRG ROM bank, fixed to the last bank

Para realizar a troca do primeiro banco, a CPU deve realizar uma escrita nos endereços de ROM 0x8000-0xFFFF, que na verdade estarão mapeados a um registrador interno do cartucho,

que fará a seleção do banco. Esse Mapper, na sua configuração mais comum, permite selecionar entre 16 bancos de 16kB para tomar essa posição, fazendo com que o programa possa ter efetivamente 256kB de memória de código, muito mais do que os 32kB anteriores.

Em Mappers mais complexos, existe também a troca dos bancos da CHR ROM que existem no espaço da memória da PPU, e eles funcionam da mesma forma. Alguns jogos até trocam um dos bancos de CHR ROM por um banco de CHR RAM, permitindo mais flexibilidades, podendo escrever na memória para trocar a cor ou os próprios tiles.



Emulando Mapper 2

A primeira coisa a se fazer, será implementar uma interface comum para todos os Mappers:

```
type Mapper interface {
    Read(address uint16) uint8
    Write(address uint16, val uint8)
    Mirroring() string
}

func NewMapper(cartridge *cartridge.Cartridge) Mapper {
    switch cartridge.MapperType {
    case 0:
        return NewMapper2(cartridge)
    case 2:
        return NewMapper2(cartridge)
    }
}
```

```

    return nil
}

```

Agora, basta implementar dentro do Mapper2, os métodos definidos na interface, utilizando corretamente o seletor para trocar os bancos:

```

func NewMapper2(cartridge *cartridge.Cartridge) Mapper {
    // each prg rom bank has 16kB
    totalBanks := int(cartridge.PrgRomSize) / 0x4000
    return &Mapper2{
        cartridge: cartridge,
        totalBanks: totalBanks,
        bankSelect: 0,
        fixedBank: totalBanks - 1,
    }
}

func (mapper *Mapper2) Read(address uint16) uint8 {
    switch {
    // chr rom unchanged in this mapper
    case address <= 0x1FFF:
        // is chr ram
        if mapper.cartridge.ChrRamSize > 0 {
            return mapper.cartridge.ChrRam[address]
        }
        return mapper.cartridge.ChrRom[address]
    case address >= 0x6000 && address <= 0x7FFF:
        return mapper.cartridge.SRam[address-0x6000]
    // prg rom switched bank
    case address >= 0x8000 && address <= 0xBFFF:
        return
    mapper.cartridge.PrgRom[(0x4000*mapper.bankSelect)+int(address-0x8000)]
    // fixed prg rom bank
    case address >= 0xC000:
        return
    mapper.cartridge.PrgRom[(0x4000*mapper.fixedBank)+int(address-0xC000)]
    default:
        fmt.Printf("Warning: cannot access address %04x with
mapper2\n", address)
    }
    return 0
}

```

```

func (mapper *Mapper2) Write(address uint16, val uint8) {
    switch {
        // chr rom unchanged in this mapper
        case address <= 0x1FFF:
            // is chr ram
            if mapper.cartridge.ChrRamSize > 0 {
                mapper.cartridge.ChrRam[address] = val
                return
            }
            fmt.Printf("Warning: cannot write to ROM address %04x with
mapper2\n", address)
        case address >= 0x6000 && address <= 0x7FFF:
            mapper.cartridge.SRam[address-0x6000] = val
            // write to bank select register
        case address >= 0x8000:
            mapper.bankSelect = int(val) % mapper.totalBanks
        default:
            fmt.Printf("Warning: cannot write to address %04x with
mapper2\n", address)
    }
}

```

Por fim, basta adaptar ambas as memórias da CPU e da PPU para utilizar o Mapper para fazer as leituras e escritas, ao invés de acessar o cartucho diretamente.

```

// memoria CPU Leitura
    case addr >= 0x6000:
        return MainMemory.Mapper.Read(addr)
    }
// CPU escrita
    case addr >= 0x6000:
        MainMemory.Mapper.Write(addr, val)
    }
// PPU Leitura
    if addr <= 0x1FFF {
        return PpuMemory.mapper.Read(addr)
    }
// PPU escrita
    if addr <= 0x1FFF {
        PpuMemory.mapper.Write(addr, val)
    }

```

Com isso, já deve ser possível jogar todos os jogos que utilizam o Mapper 2, e para executar os outros jogos, basta implementar os métodos da interface para os outros Mappers.

APU (Audio Processing Unit)

Finalmente chegamos ao último capítulo desse livro-post. Agora iremos falar sobre a última que foi implementada no projeto: A Audio Processing Unit, ou APU. Como o próprio nome sugere, é a unidade de processamento que cuida de gerar e tocar os sons e músicas presentes em todos os jogos. Esse processador vem acoplado com a CPU, e assim como a PPU, é controlado por registradores mapeados no espaço de memória da CPU.

Para gerar o áudio, a APU tem internamente 5 canais digitais de áudio: 2 canais de pulsos, 1 canal triângulo, 1 canal de ruído e 1 canal modulado por delta (DMC). Cada um desses canais é configurado pelos registradores mapeados na memória da CPU, de forma que cada uma gera um formato de onda no tempo. A cada período de amostragem, a saída de todos os canais digitais são combinadas em um mixer não linear (é um circuito analógico no chip), que então gera um valor entre 0 e 1, que é utilizado pelo alto-falante para tocar uma nota.

A APU tem alguns registradores gerais, que servem para ativar ou desativar os canais:

\$4015 write	---D	NT21	Enable DMC (D), noise (N), triangle (T), and pulse channels (2/1)
--------------	------	------	---

É relevante notar também, que assim como a PPU, a APU não funciona com o clock normal da CPU, utilizando um clock com frequência igual a metade da frequência de clock da CPU.

Também é importante notar que nem todos os componentes utilizados nos canais são ativados em cada clock, alguns são ativados apenas nos chamados Quarter Frames ou Half Frames:

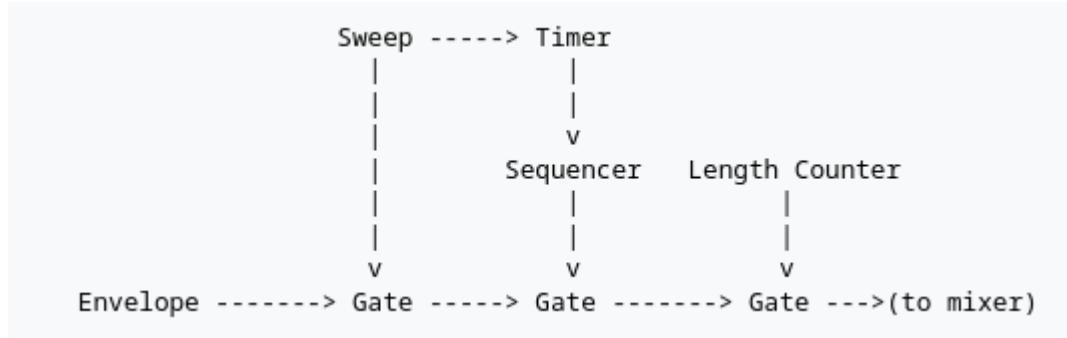
Step	APU cycles		Envelopes & triangle's linear counter (Quarter frame)	Length counters & sweep units (Half frame)	Frame interrupt flag
	NTSC	PAL			
1	3728, PUT	4156, PUT	Clock		
2	7456, PUT	8313, PUT	Clock	Clock	
3	11185, PUT	12469, PUT	Clock		
4	14914, GET	16626, GET			Set if interrupt inhibit is clear
	14914, PUT	16626, PUT	Clock	Clock	Set if interrupt inhibit is clear
	0 (14915), GET	0 (16627), GET			Set if interrupt inhibit is clear

O número de clocks de APU para gerar um Quarter Frame ou um Half Frame são valores um pouco arbitrários hardcoded diretamente no hardware e teremos apenas de aceitá-los.

Canais de pulso

Existem 2 canais de pulso na APU, os dois são praticamente iguais. Esses são possivelmente os canais mais importantes, responsáveis por boa parte das melodias e dos efeitos sonoros

dos jogos. Aqui vai um diagrama dos componentes que são utilizados para implementar um canal de pulso:

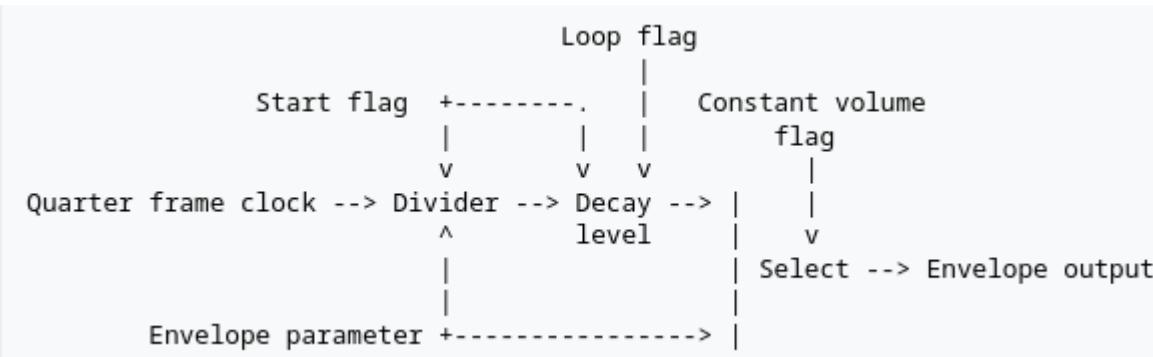


E aqui vão os registradores mapeados em memória que são utilizados para configurar a unidade de áudio:

\$4000 / \$4004	DDLC	VVVV	Duty (D), envelope loop / length counter halt (L), constant volume (C), volume/envelope (V)
\$4001 / \$4005	EPPP	NSSS	Sweep unit: enabled (E), period (P), negate (N), shift (S)
\$4002 / \$4006	TTTT	TTTT	Timer low (T)
\$4003 / \$4007	LLLL	TTTT	Length counter load (L), timer high (T)

Vários desses componentes também são utilizados para implementar os outros canais, sendo blocos básicos.

O envelope é o componente responsável por controlar a variação do volume do canal no tempo, internamente possui um contador interno decay counter que começa no valor 15, e é decrementado toda vez que um outro contador divider atinge zero. Este contador pode ser configurado pela CPU, fazendo com que aconteça um efeito de fade, com o valor do volume diminuindo. Também é possível configurá-lo para utilizar um valor constante, setando a flag de volume constante no registrador em 0x4000 ou 0x4004 para cada pulso respectivamente. Diagrama do envelope:



Em seguida, o valor do envelope passa por um Gate, que nada mais é que um componente que zera o valor caso algumas condições sejam verdadeiras, efetivamente silenciando o canal. O primeiro gate é controlado pelo componente Sweep. Antes de explicar o componente de Sweep, é necessário explicar o timer e o sequencer, para então fazer mais sentido.

O timer nada mais é que um simples temporizador: A cada vez que recebe um Clock, seu valor é decrementado, e quando atinge zero, é recarregado para o valor de período definido nos registradores. O Sequencer no contexto dos canais de pulso, é efetivamente quem vai definir se o pulso é alto, ou baixo. Ele é ativado toda vez que o timer é recarregado, e baseado no valor do Duty Cycle que é gravado pela CPU em um dos registradores, vai ativar ou desativar o gate, efetivamente criando pulsos. Aqui são os possíveis Duty Cycles:

Duty	Output waveform
0	0 1 0 0 0 0 0 0 (12.5%)
1	0 1 1 0 0 0 0 0 (25%)
2	0 1 1 1 1 0 0 0 (50%)
3	1 0 0 1 1 1 1 1 (25% negated)

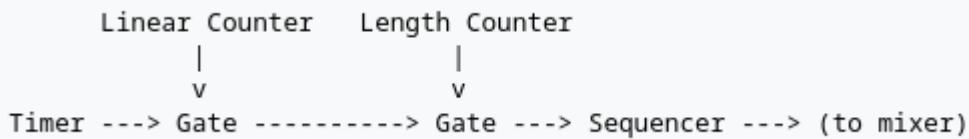
A unidade de Sweep é utilizada para modificar o timer, incrementando ou decrementando seu valor a cada ciclo que está ativo, efetivamente fazendo uma varredura (sweep) de frequências e controlando efetivamente a Altura do som, variando entre grave e agudo. O jeito que o sweep interfere no período do timer não me parece fazer sentido lógico, então deixarei aqui a página da wiki que explica as contas: [Sweep](#). Além disso, o sweep também pode acionar o gate, caso o período atual do timer seja menor do que 8, ou caso o período alvo que o sweep carregaria seja maior que 0x7FF, o que faz sentido, pois períodos muito pequenos ou muito pequenos praticamente não são audíveis. Todos os seus parâmetros são configurados a partir dos registradores mapeados.

Por fim, temos o Length Counter, que também é um simples contador, que a cada Clock enviado, é decrementado, e quanto atinge zero, o canal é silenciado. Muitos jogos o utilizam para gerar sons com duração fixa, como o pulo do Mario, por exemplo, enquanto outros preferem silenciar o canal de outras formas, zerando o volume ou pelo Sweep. Para carregar um valor nesse registrador, utilizamos novamente um dos registradores mapeados na CPU, e utilizamos este valor para selecionar um valor da tabela de Length Count, que existe para permitir mais flexibilidade dos comprimentos:

		If the enabled flag is set, the length counter is loaded with entry L of the length table:																																																		
bits 7-3	LLLL L---	<table border="1"> <thead> <tr> <th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>A</th><th>B</th><th>C</th><th>D</th><th>E</th><th>F</th></tr> </thead> <tbody> <tr> <td>00-0F</td><td>10,254</td><td>20,</td><td>2,</td><td>40,</td><td>4,</td><td>80,</td><td>6,</td><td>160,</td><td>8,</td><td>60,</td><td>10,</td><td>14,</td><td>12,</td><td>26,</td><td>14,</td></tr> <tr> <td>10-1F</td><td>12,</td><td>16,</td><td>24,</td><td>18,</td><td>48,</td><td>20,</td><td>96,</td><td>22,</td><td>192,</td><td>24,</td><td>72,</td><td>26,</td><td>16,</td><td>28,</td><td>32,</td><td>30</td></tr> </tbody> </table>		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	00-0F	10,254	20,	2,	40,	4,	80,	6,	160,	8,	60,	10,	14,	12,	26,	14,	10-1F	12,	16,	24,	18,	48,	20,	96,	22,	192,	24,	72,	26,	16,	28,	32,	30
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F																																				
00-0F	10,254	20,	2,	40,	4,	80,	6,	160,	8,	60,	10,	14,	12,	26,	14,																																					
10-1F	12,	16,	24,	18,	48,	20,	96,	22,	192,	24,	72,	26,	16,	28,	32,	30																																				

Canal de triângulo

Só existe um canal de triângulo na APU. Esse canal é um dos mais silenciosos quando jogamos, sendo mais responsável pelos tons mais baixos da música, atuando mais como um fundo. Aqui vai um diagrama de como ele é constituído:



Seus componentes são controlados pelos seguintes registradores mapeados na CPU:

\$4008	CRRR RRRR	Length counter halt / linear counter control (C), linear counter load (R)
\$4009	---- ----	Unused
\$400A	TTTT TTTT	Timer low (T)
\$400B	LLLL LTTT	Length counter load (L), timer high (T), set linear counter reload flag

Esse canal novamente possui um timer utilizado para ativar um sequencer. Esse sequencer é diferente do sequencer utilizado no canal de pulso: Ao invés de gerar 1 ou 0 dependendo do Duty Cycle, ele simplesmente gera sequencialmente o valor presente em sua tabela, gerando efetivamente um triângulo na saída:

The sequencer sends the following looping 32-step sequence of values to the mixer:

```

15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

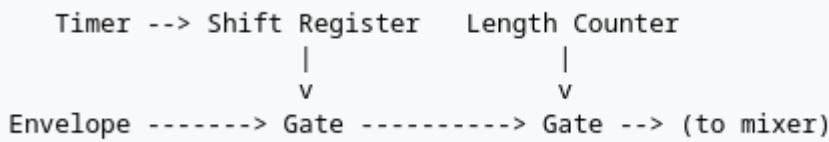
```

O Linear counter e o Length counter são basicamente equivalentes, cada um é carregado com um período, e quando algum dos dois é igual a 0, silencia o canal.

Canal de Ruído

Só existe um canal de ruído na APU, e ele é utilizado pela maioria dos jogos para gerar a batida da música, efetivamente como um instrumento de percussão. Aqui vai o diagrama de

como ele é constituído:



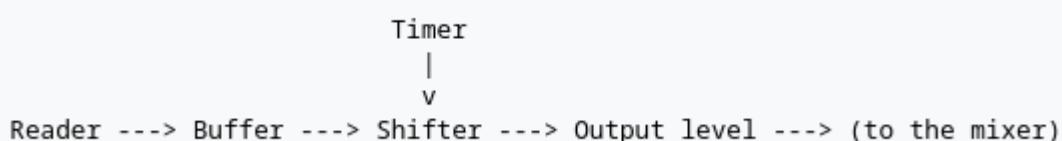
Aqui novamente temos a presença do envelope, de um timer e de um Length Counter. A única diferença ocorre no timer, onde o período não é escrito diretamente pelo valor no registrador, mas é obtido através da segunda tabela, baseado no valor escrito:

		The timer period is set to entry P of the following:																
		Rate	\$0	\$1	\$2	\$3	\$4	\$5	\$6	\$7	\$8	\$9	\$A	\$B	\$C	\$D	\$E	\$F
bits 3-0	---- PPPP	NTSC	4,	8,	16,	32,	64,	96,	128,	160,	202,	254,	380,	508,	762,	1016,	2034,	4068 (2046 in old)
		PAL	4,	8,	14,	30,	60,	88,	118,	148,	188,	236,	354,	472,	708,	944,	1890,	3778

O elemento mais interessante desse canal, é o shift register, pois é a partir dele que é gerado a pseudo-aleatoriedade utilizada para gerar o ruído. Cada vez que o timer chega a zero, é calculado um bit de feedback, que dependendo do modo é igual ao OR entre o bit 0, e um outro bit, bit 1 ou 6, dependendo do modo gravado no registrador. Com esse feedback calculado, o registrador é shiftado para direita uma vez, e o feedback é colocado no bit 14 do registrador. Isso resulta em uma cadeia pseudo-aleatória de bits, que começa a repetir depois de alguns milhares de shifts. O valor do bit 0 então é utilizado para ativar ou desativar o gate, silenciando ou liberando o canal.

Canal DMC

Por fim, temos o canal DMC. Este canal é utilizado para tocar alguns samples de áudios gravados, mas é raramente utilizado em jogos, pois gasta bastante memória do cartucho. Ele utiliza a técnica de Delta Modulation, onde o sinal inicia em um valor, e a cada 1 ou 0 lidos, somamos ou decrementamos um delta deste valor, respectivamente. Aqui vai o diagrama deste canal:



E aqui os registradores utilizados para configurá-lo:

\$4010	IL-- RRRR	IRQ enable (I), loop (L), frequency (R)
\$4011	-DDD DDDD	Load counter (D)
\$4012	AAAA AAAA	Sample address (A)
\$4013	LLLL LLLL	Sample length (L)

O funcionamento desse canal também é relativamente simples. Quando o canal é ativado, o Reader fará uma leitura de 1 byte da memória da CPU, utilizando o valor em Sample Address no formato: %11AAAAAA.AA000000, que então é incrementado ao mesmo tempo que o registrador de sample length é decrementado. Esse valor é então armazenado em um buffer. A cada vez que o timer atingir zero, obtemos o bit menos significativo do buffer, e se for 0, subtraímos 2 do Output Level, e se for 1, somamos 2 ao Output Level e shiftamos o buffer para a direita. O Output Level vai de 0 até 127, qualquer tentativa de incrementar acima ou abaixar desse valor apenas mantém ele como está. Isso se repete até o buffer esvaziar, onde o Reader faz outra leitura para recarregá-lo. Quando o Sample Length atinge zero, o canal é silenciado e nenhum byte é lido.

Esse canal também conta com algumas funcionalidades de gerar interrupções IRQ, mas decidi por não implementá-las, já que são usadas por pouquíssimos jogos, sendo a maior parte deles desconhecidos e irrelevantes.

Mixer

Este componente é responsável por agregar todas as saídas de todos os canais em um só valor real de 0 a 1, que é então enviado para os speakers para gerar o som naquele instante. No hardware, cada canal tem seu próprio conversor Digital/Analógico implementados de forma que ocorra não-linearidades e também que um canal influencia o outro de forma bastante complexa. Felizmente, a comunidade entusiasta já fez o trabalho difícil, e analisou as formas de onda da saída, e a partir dela construíram equações que conseguem aproximar o valor da saída dado a saída digital dos canais:

```

output = pulse_out + tnd_out

95.88
pulse_out = -----
          (8128 / (pulse1 + pulse2)) + 100

159.79
tnd_out = -----
          1
          ----- + 100
          (triangle / 8227) + (noise / 12241) + (dmc / 22638)

```

Lembrando que nesse caso, o canal DMC tem valores de 0 a 127, enquanto os outros canais vão de 0 a 15. Novamente, aparecem vários números estranhos e aparentemente aleatórios nas fórmulas, que vamos apenas aceitar.

Assim, dada uma taxa de amostragem do sinal de áudio, normalmente 44.1KHz, fazemos a consulta de uma amostra da saída do mixer e enviamos para o speaker tocar.

Emulação

O primeiro passo para emularmos a APU é implementar os blocos básicos presentes na maior parte dos canais, sendo eles envelope, timer e length counter:

```
package apu

// ===== //  
// || //  
// || ENVELOPE //  
// || //  
// ===== //  
//  
  
type Envelope struct {  
    reload      bool  
    period      uint  
    constVolume uint  
    loop        bool  
    isConstant  bool  
  
    // internals  
    decayCounter uint  
    value        uint  
}  
  
// standard start volume when resetting envelope  
const envelopeStartVolume = 15  
  
func (envelope *Envelope) Clock() {  
    // if the start flag is set, load the decay counter and the divider  
    // with the respective values  
    if envelope.reload {  
        envelope.decayCounter = envelopeStartVolume  
        envelope.value = envelope.period  
        envelope.reload = false  
    } else {  
        // envelope clocked while 0, we reload the period  
        if envelope.value == 0 {  
            envelope.value = envelope.period  
            // now we clock the decay counter  
            // we decrement if it is not already 0  
            // if the length counter halt flag is active, we just  
Load decay with 15
```

```

        if envelope.decayCounter > 0 {
            envelope.decayCounter -= 1
        } else if envelope.loop {
            envelope.decayCounter = 15
        }

    } else {
        // if it is not zero, we decrement the divider
        envelope.value -= 1
    }
}

func (envelope *Envelope) getVolume() uint {
    if envelope.isConstant {
        return envelope.constVolume
    } else {
        return envelope.decayCounter
    }
}

// ===== //
// ||
// ||          LENGTH COUNTER          ||
// ||
// ===== //
//


var lengthLookUpTable = []uint{
    10, 254, 20, 2, 40, 4, 80, 6, 160, 8, 60, 10, 14, 12, 26, 14,
    12, 16, 24, 18, 48, 20, 96, 22, 192, 24, 72, 26, 16, 28, 32, 30,
}

type LengthCounter struct {
    value  uint
    halted bool
}

func (lengthCounter *LengthCounter) Clock(channelEnabled bool) {
    if lengthCounter.value > 0 && !lengthCounter.halted && channelEnabled
    {
        lengthCounter.value--
    }
}

```

```

}

func (LengthCounter *LengthCounter) setValue(val uint) {
    LengthCounter.value = lengthLookUpTable[val]
}

// ===== //
// ||          RAW TIMER           ||
// ||                                ||
// ||                                ||
// ===== //

type RawTimer struct {
    value  uint
    period uint
}

type ExecuteOnTimerZero func()

// functions for use when the timer is 11 bit
func (timer *RawTimer) setTimerHigh(val uint8) {
    timer.period = (uint(val&0b111) << 8) | (timer.period & 0x00FF)
}

func (timer *RawTimer) setTimerLow(val uint8) {
    timer.period = uint(val) | (timer.period & 0xFF00)
}

// clocks timer, when it is 0 do something
func (timer *RawTimer) Clock(fc ExecuteOnTimerZero) {
    if timer.value == 0 {
        timer.value = timer.period
        fc()
    } else {
        timer.value--
    }
}

```

Em seguida, podemos trabalhar em emular cada um dos canais, começando com os canais de pulso:

Declaramos as funções que serão mapeadas na memória da CPU

```

// Write to register 0x4000 / 0x4004 of pulse registers
func (pulse *Pulse) WriteToDutyCycleAndVolume(val uint8) {
    // val = DDLC VVVV
    // DD -> dutyCycle
    // L -> LengthEnable : 1 = infinite ; 0 = enable counter
    // C -> constantEnvelope : 1 volume = constant ; 0 = use the envelope
    // VVVV -> constant volume if C = 1 or envelope decay if C = 0
    pulse.dutyCycle = uint((val >> 6) & 0b11)
    pulse.envelope.loop = (val>>5)&0b1 == 1
    pulse.envelope.isConstant = (val>>4)&0b1 == 1
    pulse.envelope.constVolume = uint(val & 0x0F)
    pulse.envelope.period = uint(val & 0x0f)
    pulse.lengthCounter.halted = (val>>5)&0b1 == 1
}

// write to register 0x4001 / 0x4005 of pulse registers
func (pulse *Pulse) WriteToSweep(val uint8) {
    // val = EPPP.NSSS
    // E -> pulse sweep enable
    // PPP -> sweep divider period
    // N -> sweep negate
    // SSS -> shift amount
    pulse.sweepEnabled = (val>>7)&0b1 == 1
    pulse.sweepDividerPeriod = uint((val>>4)&0b111) + 1
    pulse.sweepNegate = (val>>3)&0b1 == 1
    pulse.sweepShiftCount = uint(val & 0b111)
    pulse.sweepReload = true
}

// write to register 0x4002 / 0x4006 of pulse registers
func (pulse *Pulse) WriteToTimerLow(val uint8) {
    // val = LLLL.LLLL
    // LLLL.LLLL -> Lower 8 bits of sequencer timer
    pulse.timer.setTimerLow(val)
}

// write to register 0x4003 / 0x4007 of pulse registers
func (pulse *Pulse) WriteToTimerHigh(val uint8) {
    // val = LLLL.LHHH
    // LLLL.L -> Length counter Load
    // HHH -> high 3 bits of sequencer timer
    // also set the start envelope flag
    pulse.envelope.reload = true
}

```

```

        pulse.lengthCounter.setValue(uint(val) >> 3)
        pulse.timer.setTimerHigh(val)
        pulse.sequencerStep = 0
    }
}

```

Em seguida, basta seguir o diagrama que foi explicado anteriormente. O código completo pode ser consultado no repositório: [Repositório](#)

Definimos também algumas funções ajudantes, como ClockHalfFrame e ClockQuarterFrame, que avançam os devidos componentes, e a função GetSample, que retorna o valor atual da saída.

A implementação é análoga para todos os outros canais, e não convém explicar a mesma coisa 4 vezes, sendo mais relevante falar sobre o mixer, e como integrar a APU com o resto do emulador.

Para o mixer, não é uma boa ideia executar aqueles cálculos todo momento que for solicitada uma sample, pois são cálculos de ponto flutuante com várias divisões e multiplicações, e para algo que será chamado com bastante frequência, podemos ter problemas. Para evitar isso, a própria comunidade bolou algumas funções aproximadoras que podemos utilizar para gerar a Lookup Tables dos valores de saída do mixer:

Lookup Table

The APU mixer formulas can be efficiently implemented using two lookup tables: a 31-entry table for the two pulse channels and a 203-entry table for the remaining channels (due to the approximation of tnd_out, the numerators are adjusted slightly to preserve the normalized output range).

```

output = pulse_out + tnd_out
pulse_table [n] = 95.52 / (8128.0 / n + 100)
pulse_out = pulse_table [pulse1 + pulse2]

```

The tnd_out table is approximated (within 4%) by using a base unit close to the DMC's DAC.

```

tnd_table [n] = 163.67 / (24329.0 / n + 100)
tnd_out = tnd_table [3 * triangle + 2 * noise + dmc]

```

Basta pré calcular todos os valores possíveis para essas tabelas, sendo as entradas de 0 a 30 para a pulse_table, e de 0 a 202 para a tnd_table (triangle noise dmc table). É importante notar que a entrada zero equivale a todos os canais silenciados, então é natural que nas fórmulas o valor dessa entrada não seja indefinida, mas seja 0.

```

// Build the mixer pulse value LookUp table for faster processing
func BuildPulseLookupTable() []float64 {
    // LookUpTable approximation for the mixer output
    lookupTable := make([]float64, 31)
    lookupTable[0] = 0
    for i := range 30 {
        lookupTable[i+1] = 95.52 / ((8128.0 / float64(i)) + 100.0)
    }
    return lookupTable
}

```

```

}

// Build the mixer triangle, noise and dmc Lookup table
func BuildMixerLookupTable() []float64 {
    // LookUpTable approximation for the mixer output
    lookupTable := make([]float64, 203)
    lookupTable[0] = 0
    for i := range 202 {
        lookupTable[i+1] = 163.67 / ((24329.0 / float64(i)) + 100.0)
    }
    return lookupTable
}

```

Agora, na função GetSample da APU, obtemos as amostras de cada canal, e combinamos utilizando nossos Lookup Tables, e retornamos este valor.

```

func GenSample() float32 {
    pulse1Sample := apu.Pulse1.getSample()
    pulse2Sample := apu.Pulse2.getSample()
    triangleSample := apu.Triangle.getSample()
    noiseSample := apu.Noise.getSample()
    dmcSample := apu.Dmc.getSample()

    mixedSample := apu.filterchain.Step(
        float32(pulseLookUpTable[pulse1Sample+pulse2Sample])) +
    float32(mixerLookUpTable[3*triangleSample+2*noiseSample+dmcSample])

    return mixedSample
}

```

Agora podemos considerar uma frequência de 44.1KHz para a amostragem e utilizar a biblioteca OTO para gerenciar o tocador de áudio. Entretanto surge aqui um problema: A nossa estratégia de executar todos os ciclos de CPU por frame não funciona mais. Se executarmos todos os ciclos de CPU, PPU e APU em uma fração do tempo de frame, não será feita uma amostragem verossímil do áudio, já que a maior parte das amostras vão ser do estado final da APU e serão todas iguais. Isso não seria um problema tão grande se a nossa audição não fosse tão aguçada ao ponto desse comportamento causar incômodos, e sons de pipocadas e estouros.

Uma forma de resolver esse problema, seria reestruturar nosso programa de forma que ao invés de executar a quantidade de ciclos necessária para um frame de vídeo, executarmos a quantidade de ciclos necessária para gerar as 735 amostras ($44.1\text{KHz} / 60 \text{fps} = 735$), assim garantimos sempre que teremos a quantidade correta de amostras, evitando pipocadas.

O primeiro passo para implementar essa estratégia foi converter todos os componentes para utilizarem o Clock da PPU como base, já que é o mais rápido, aumentando ainda mais a precisão da nossa emulação. Isso é bastante simples de fazer, bastando adicionar contadores e condições no início da CPU e APU, para contar um ciclo apenas quando a função Clock foi chamada 3 vezes ou 6 vezes, para a CPU ou APU respectivamente.

```
// cpu
func Clock() {
    cpu.clockCounter++
    if cpu.clockCounter == 3 {
        cpu.cpuCycle++
        cpu.clockCounter = 0
    }
    // continua
}

// apu
func Clock() {
    apu.clockCounter++
    if apu.clockCounter == 3 {
        // triangle clocks at cpu speed
        apu.Triangle.clockTimer()
        apu.Noise.clockTimer()
    }
    if apu.clockCounter == 6 {
        apu.apuCycle++
        apu.Pulse1.clockTimer()
        apu.Pulse2.clockTimer()
    }
    // continua ...
}
```

Agora para ajustar o Loop do ebiten, basta fazer algumas continhas: ciclos por amostra = clock ppu / 44100Hz = 5369318Hz / 44100Hz = 121.75 ciclos por amostra de áudio. Assim, na função Update do ebiten bastamos executar todos os componentes por essa quantidade de ciclos, gerar uma amostra e enviar para o buffer de reprodução de áudio, executar mais alguns ciclos, até gerarmos 735 amostras. Note também que o número de ciclos é quebrado em 121.75, para aumentar a precisão podemos utilizar um contador float, que gera uma amostra toda vez que passa de 121.75, e então subtraí 121.75, fazendo uma auto correção no número de ciclos da próxima amostra.

Infelizmente nem tudo são flores, com essa solução agora criamos um outro problema, desta vez na PPU. Se deixarmos do jeito que está, acabaremos pegando alguns frames da PPU renderizados pela metade, gerando um efeito de scan e uma linha desincronizada na tela, já que o tempo para gerar as 735 amostras não é exatamente o mesmo para gerar um frame de vídeo. Mas é simples resolver isso, utilizamos uma tática chamada Front Buffer Rendering, que consiste em criar 2 Buffers, onde um sempre vai conter um frame completo, enquanto o outro é renderizado. Dessa forma, mostraremos na tela sempre o buffer completo, e por mais que ocorra o atraso e repetição de um frame para o outro, essa ocorrência não é perceptível para

nós (pense quando um jogo cai de 60 para 59 fps, impossível de perceber ...) e é bem mais aceitável que as pipocadas ao perder amostras.

```
func (g *Game) Update() error {
    start := time.Now()
    handleInput()

    // Emulation step
    sampleCount := 0

    // sync to audio because it is easier
    for {
        tick()
        if audioRate >= cyclesPerSample {
            audioRate -= cyclesPerSample
            sample := apu.GenSample()
            bs := math.Float32bits(sample)

            binary.LittleEndian.PutUint32(g.audioBuffer[sampleCount*4:], bs)
            sampleCount++
            if sampleCount == samplesPerFrame {
                break
            }
        } else {
            audioRate++
        }
    }

    buf := make([]byte, len(g.audioBuffer))
    copy(buf, g.audioBuffer)

    select {
    case g.audioChan <- buf:
    default: // drop if channel is full to avoid blocking
    }

    rgb := ppu.GetPpu().GetPixelData()
    convertRGB24ToRGBA(g.pixels, rgb)
    duration := time.Since(start)
    if duration > time.Second/60 {
        log.Printf("Slow frame detected: %v", duration)
    }
    return nil
}
```

Na nova função Update temos a nova lógica supracitada, e também o envio das amostras de áudio para o canal de áudio configurado e providenciado pela biblioteca Oto, que não será explicada pois foge do escopo, mas sua implementação está presente no repositório.

Implementar o Front Buffer rendering é trivial na PPU, basta criarmos mais um struct Frame, e uma variável para selecionar entre qual está sendo renderizada:

```
// frame control campos do struct PPU
bufferFrames [2]*Frame
frontBuffer uint
```

Na função que chama o método setPixel do frame, trocamos para utilizar o frame indexado pela nova variável:

```
copy(ppu.bufferFrames[1-ppu.frontBuffer].PixelData[address:], rgb[:])
```

E toda vez que atingirmos VBlank, indica que um frame ficou pronto, então simplesmente trocamos o valor da variável seletora de buffer:

```
if vblank {
    // swap front buffer
    ppu.frontBuffer = 1 - ppu.frontBuffer
    if ppu.getControlSetting(VBLANK_NMI_ENABLE) == 1 {
        ppu.NmiInterrupt = true
    }
    ppu.setVblankStatus(1)
}
```

Com isso já é possível jogar a maior parte dos jogos agora com som e sem pipocadas ou distorções no vídeo!

Executando na placa RockPro64

Como um adicional ao projeto, foi sugerido pelo professor executar o emulador e seus jogos em uma placa arm, validando que o GO e suas bibliotecas realmente funcionem corretamente quando compiladas para a arquitetura que utilizamos em aula.

O processo foi bastante simples, dado que foi instalado o Armbian na placa, uma distribuição do Linux que é basicamente o Debian, mas feito para executar em processadores ARM. Com o Linux na placa, basta realizar a instalação do GO utilizando o apt, e executar os comandos padrões para gerenciar dependências e compilar o código.

Entretanto, utilizando a biblioteca Ebiten e Oto até agora, o emulador acabou ficando bastante lento, mesmo em uma placa mais poderosa como o RockPro64. A lentidão infelizmente era perceptível, e acaba causando estouros no buffer de áudio, fazendo com que o áudio ficasse extremamente desagradável.

Para resolver esse problema, foi realizada a mudança das bibliotecas utilizadas até o momento, para uma biblioteca mais simples e de baixo nível, o SDL2. Não convém explicar as mudanças aqui, dado que foi basicamente uma conversão 1 para 1 não muito interessante. O que é interessante foi o aumento de performance fazendo com que o jogo rode de forma mais lisa e consistente. O problema com o áudio ainda existia, pois ele ainda tinha alguns pequenos delays. Isso foi resolvido simplesmente aumentando o buffer de áudio para alguns frames a mais, que ajudou a estabilizar o áudio.

Para fechar o projeto, foi comprado um controle USB retrô de Nintendinho, que foi conectado na placa, e foi feito o mapeamento de seus botões para o controle virtual emulado utilizando também o SDL2, fazendo com que ele se tornasse verdadeiramente um pequeno console. Essas novas modificações foram colocadas na branch sdl2 do repositório: [Repositório](#)