Emulador de Nintendo implementado em Python

José Rafael F. P. de Carvalho[1], **Brivaldo A. S. Junior**[1]

[1]Faculdade de Computação - Universidade Federal de Mato Grosso do Sul (UFMS)
Caixa Postal 549 - 79.070-900 - Campo Grande - MS - Brazil

jose.carvalho@aluno.ufms.br, brivaldo@facom.ufms.br

Abstract. *In this project we aim to analyze the efficiency of an emulated software developed in an interpreted programming language. Thereunto, a functional NES emulator was developed with the Python language, and was used simple games to check the performance os the emulator. The developed emulator doesn't have a good performance and, despite it works with some games, you can't use the emulator other than for academic purpose, due to slowness. We came to the conclusion that Python is not the best language to create emulated systems, due to the fact that to develop such kind of system uses complex and slow algorithms that are usually performed by hardware.*

Resumo. *Neste projeto temos como objetivo analisar a eficiência de um software emulado desenvolvido em uma linguagem de programação interpretada. Para isso, foi desenvolvido um emulador funcional de NES com a linguagem Python e, utilizamos jogos simples para verificar o desempenho do emulador. O emulador desenvolvido não tem um bom desempenho e, apesar de funcionar corretamente com alguns jogos, não é possível utilizar o emulador sem ser para uso acadêmico, devido a lentidão. Chegamos a conclusão de que Python não é a melhor linguagem para criar sistemas emulados, devido ao fato de que para desenvolver esse tipo de sistema se utiliza algoritmos complexos e lentos que, geralmente, são executados por* hardware.

1. Introdução

Com o decorrer dos anos, a capacidade de processamento dos computadores tem aumentado de forma acelerada e, em 1965, Gordon E. Moore fez uma declaração sobre esse avanço [1], dizendo que o poder de processamento dos computadores dobraria a cada 18 meses. Essa observação ficou conhecida como Lei de Moore. Ela se mostrou verdadeira (Durante vários anos). Para criar processadores melhores foram desenvolvidas arquiteturas mais complexas, maior quantidade de instruções e mais elaboradas. Devido a esse avanço na arquitetura dos processadores, surgiram alguns problemas de compatibilidade. Além disso *softwares* legados precisavam continuar operando em novos sistemas. Para evitar que tais programas fossem considerados obsoletos, surgiu o conceito de emulador. Ele consiste em uma aplicação capaz de fazer com que um sistema computacional "imite" um outro ambiente computacional.

A principal motivação para se utilizar emuladores é para incorporar aplicações antigas em *hardwares* modernos e sofisticados. Como nem sempre os programas antigos são atualizados e disponibilizados para novas plataformas e *hardwares* pelo fabricante, isso acaba ocasionando um grande problema para empresas que utilizam esses *softwares*.

Essas empresas são obrigadas à conviver com problemas de segurança e desempenho, e uma solução para esses problemas são os emuladores. Por exemplo: os consoles de videogames antigos, os quais tem jogos exclusivos, podem ser executados em consoles modernos (prática comum da Nintendo), ou em outras plataformas como o computador pessoal.

Neste trabalho será criado um emulador do console de videogame *Nintendo Entertainment System* (NES), utilizando a linguagem de programação interpretada Python. O NES é um console lançado pela Nintendo em 1983 e na época foi o videogame de maior sucesso comercial, por ter jogos fabricados por terceiros. A escolha desse sistema se deu pelo fato da arquitetura ser simples, possuir muitos jogos com recursos diferentes, e possuir uma comunidade grande e ativa de desenvolvedores e entusiastas na Internet.

Um dos objetivos desse trabalho é compreender a desempenho de emuladores feitos em linguagens interpretadas. Python (objeto de estudo) foi lançada em 1991, com o objetivo de priorizar a legibilidade do código ao invés da velocidade. E uma linguagem compilada para *bytecode*, um conjunto de instruções que é interpretada depois pela Máquina Virtual do Python. Em comparação a outras linguagens, na maioria das aplicações Python perde em termos de velocidade para Java ou C++, como mostrado na Figura 1[2].

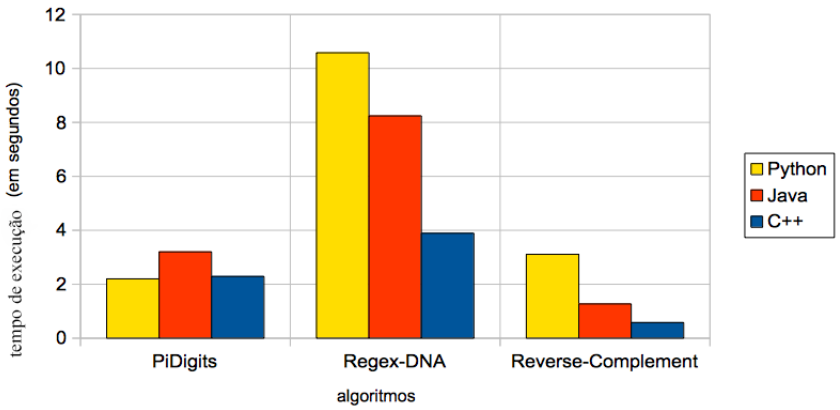

Figura 1. Comparação Python, Java, C++.

O objetivo desse projeto não é comparar a velocidade de Python com outras linguagens, e sim averiguar se Python pode ser usado para desenvolver uma aplicação capaz de emular *softwares* desenvolvido em outras plataformas.

A estrutura deste trabalho está organizado da seguinte forma: na Seção 2 temos o referencial teórico utilizado, conceitos de Emulador, do NES e de Python; na Seção 3 é explorado o desenvolvimento do emulador NES; na Seção 4 os resultados obtidos na execução do emulador; e na Seção 5 a conclusão do trabalho.

2. Referencial Teórico

2.1. Nintendo Entertainment System

História

O videogame foi desenvolvido e lançado pela Nintendo no Japão em 1983. Foi conhecido pelo nome de *Nintendo Family Computer* ou Famicom e, junto com o videogame, foi portado 3 jogos para o Famicom: Donkey Kong, Donkey Kong Jr. e Popeye.

No final de 1984 se tornou o console mais vendido do Japão e, devido a esse sucesso a Nintendo mostrou interesse em lançar o seu console no mercado mundial. Começaram a negociar com a Atari para lançá-lo nos Estados Unidos, mas no último instante a Atari desistiu do negócio e a Nintendo decidiu lançar seu sistema ela mesma.

A versão americana do Famicom foi apresentada em junho de 1985 na *Consumer Eletronic Show* (CES), o Famicom foi completamente redesenhado, como consta na Figura 1, e recebeu um novo nome, *Nintendo Entertainment System* ou NES. A principio a Nintendo disponibilizou um número limitado de consoles para teste de mercado na cidade de Nova York em 18 de outubro de 1985. O teste não fez muito sucesso, então a Nintendo tentou mais uma vez, em fevereiro de 1986, e em setembro do mesmo ano foi lançado para toda a América do Norte, junto com 17 jogos.

No Brasil o NES foi lançado e produzido pela *Playtronic*, associação da Estrela com a Gradiente, somente em 1993 junto com seu sucessor, o Super Nintendo. O NES não fez muito sucesso no Brasil, pois como o lançamento foi muito tardio, muitas empresas nacionais fabricaram clones do NES, que eram consoles compatíveis com as fitas do NES ou do Famicom sem a licença da Nintendo. Então quando o NES chegou ao Brasil, as pessoas estavam mais interessadas no Super Nintendo, o console de 16-bits da Nintendo.

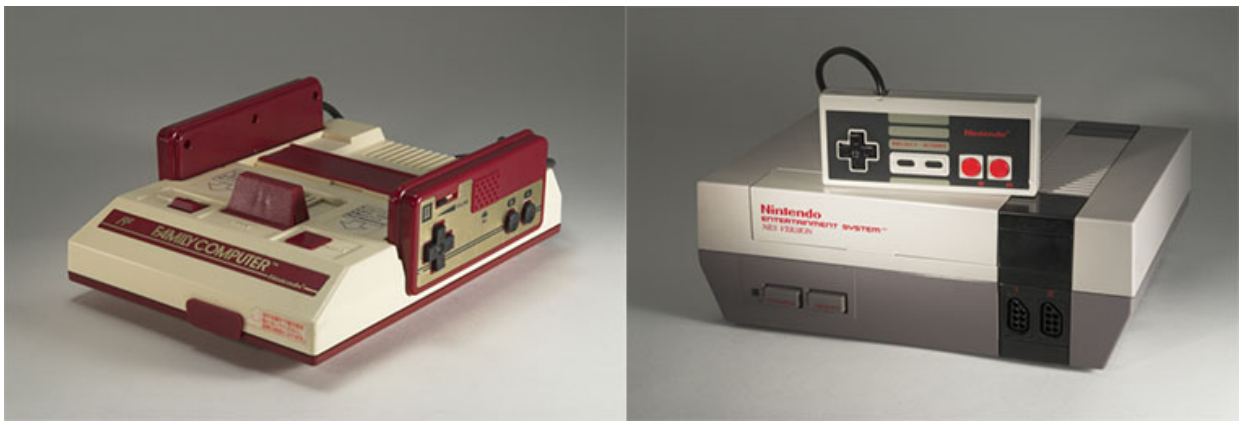

Figura 2. Famicom e Nintendo Entertainment System

O *hardware* do NES consistem em um cartucho do jogo, que armazena as instruções e padrões de imagens do jogo; a *Central Processing Unit* (CPU), que é responsável por ler o jogo do cartucho, interpretar as instruções, controlar o que a *Picture Processing Unit* PPU vai mostrar na tela, e lidar com o som; e a PPU, que é responsável por calcular e mostrar toda a parte gráfica do jogo.

O NES possui um processador somente para a parte gráfica, pois como a Nintendo quis fazer um console barato foi preciso utilizar um processador mais antigo para a CPU.

Essa CPU seria suficiente para executar os jogos mas não seria possível gerar os gráfico, então foi adicionado a PPU.

Cartucho

Os jogos do *Nintendo Entertainment System* eram distribuídos em cartuchos oficialmente chamados de *Game Pak*. O cartucho consiste de uma memória ROM com as instruções do jogo armazenado, um conector de 72 pinos, para a troca de dados com o console e, em alguns jogos, possui uma memória RAM que é alimentada por uma bateria, utilizada para salvar o estado do jogo.

O arquivo lido pelo emulador tem a extensão .nes, e é de um formato chamado iNES criado por Marat Fayzullin [3] para o seu emulador de NES. Esse formato foi criado pois somente com o conteúdo do ROM não era possível identificar qual parte do arquivo significa o que, então foi adicionado um cabeçalho de 16 *bits* no começo do arquivo para identificar várias caracteristicas do ROM, e foi organizado todo o arquivo com base no cabeçalho. Esse formato é o mais popular, utilizado pela maior parte dos emuladores de NES.

O cabeçalho do arquivo .nes no formato iNES está descrito na Tabela 1.

Tabela 1. Cabeçalho de ROM iNes

Byte de Inicio	**Tamanho *(Byte)***	**Conteúdo**
0	3	Deve conter a *String* 'NES' para identificar que é um arquivo no formato iNES.
3	1	Deve conter o valor $1A, também utilizado para identificar o formato do arquivo.
4	1	Número de bancos de PGR-ROM. PGR-ROM significa *Program ROM*, e é a área do ROM utilizada para armazenar o código do jogo. Cada bloco de PGR-ROM contém 16 KB.
5	1	Contém o número de bancos de CHR-ROM. CHR-ROM significa *Character ROM* e é utilizada para armazenar informações gráficas. Cada bloco de CHR-ROM contém 8 KB.
6	1	*Byte* de Controle 1: Bit 0 - Indica o tipo de espelhamento utilizado no jogo, 0 é espelhamento horizontal e 1 é espelhamento vertical. Bit 1 - Indica a presença de memória RAM alimentada por bateria, localizada no endereço $6000 - $7FFF. Bit 2 - Indica a presença de *512-byte trainer* no endereço de memória $7000 - $71FF. *512-byte trainer* é utilizada em alguns Memory Mappers (mapeamento de memória). Bit 3 - Indica que o jogo é dividido em 4 telas. Substitui o Bit 0 Bits 4-7 - 4 *bits* menos significativos do *Memory Mapper*.
7	1	*Byte* de Controle 2: Bits 0-3 - Não são utilizados e devem sempre ser 0. Bits 4-7 - 4 *bits* mais significativos do *Memory Mapper*.
8	1	Número de bancos de memória RAM. Para compatibilidade com a versão antiga do formato iNES. Assume que utiliza 1 página se o *byte* for 0. Cada bloco de RAM contém 8 KB.
9	7	Não são utilizados e devem sempre ser 0.

Depois do cabeçalho encontra-se o *512-byte trainer*, se ele estiver presente, caso contrario são os bancos de PRG-ROM e em seguida os bancos de CHR-ROM. O formato iNES permite 256 *Memory Mappers*.

A princípio a memória do NES era suficiente para os primeiros jogos, mas conforme foram surgindo jogos mais complexos, a memória se mostrou insuficiente. Então, para que pudessem ser desenvolvidos jogos maiores e mais complexos, foi adicionada mais memória diretamente no cartucho. Essa prática foi chamada de *Memory Mapper*, ou MMC(*Memory Management Chip*).

O *Memory Mapper* funciona da seguinte maneira: o console carrega na memória alguns bancos de dados do cartucho, conforme o *software* solicita, é realizada a troca dos

bancos que estão carregadas na memória do console e do *Memory Mapper*. Cada *Memory Mapper* funciona de uma maneira diferente, alguns *Memory Mappers* foram criados pela própria Nintendo, outros foram criados pela fabricante dos jogos.

Alguns dos *Memory Mappers* mais comuns estão descritos abaixo.

- NROM, não é utilizado nenhum *Memory Mapper*. *Super Mario Bros e Donkey Kong* são dois dos jogos mais famoso de NES e não utilizam nenhum *Memory Mapper*. [4]
- UNROM, o primeiro *Memory Mapper* criado. Suporta somente a troca de bancos de PRG-ROM. Permite que até 8 bancos de 16KB de PRG-ROM utilizem esse mapeamento. Utilizado pelo *Bomberman 2, Castlevania e Final Fantasy 1 e 2*. [4]
- CNROM, suporta somente a troca de bancos de CHR-ROM, permite que jogos com gráficos melhores fossem criados. Utilizado pelos jogos *Paperboy e Dragon Quest*. [4]
- MMC1 é o *Memory Mapper* mais utilizado, permite a troca de bancos de PRG-ROM e CHR-ROM e permite que até 8 bancos de 16KB de PRG-ROM utilizem esse mapeamento. Alguns jogos famosos que utilizam esse mapeamento são *Castlevania 2, Legend of Zelda e Mega Man 2*. [4]

Há vários outros *Memory Mappers* para o NES, como MMC2, MMC3, MMC4, MMC5, criados pela Nintendo, e alguns criados pelos desenvolvedores de jogos utilizados somente pelos jogos da empresa, como *VRC4a, VRC2a*, desenvolvidos pela *Konami* e o *Bandai Chip*, desenvolvido pela *Bandai*.

CPU

O NES é um console de 8 *bits*, ou seja, tem os registradores e barramentos de memória de tamanho 8 *bits*. A Nintendo optou por um processador de 8 *bits* por ser mais barato em relação a um de 16 *bits*, e ela queria desenvolver um console mais barato devido ao *crash* (quebra) dos jogos eletrônicos em 1983 [5].

A CPU utilizada no NES é uma variação do processador 6502 da *MOS Technology*, chamado 2A03 produzido pela Ricoh ltda. As diferenças entre os processadores é que o 2A03 tem a habilidade de manipular som, sendo usado como pAPU e como CPU, e não possui Decimal codificado em Binário (BCD), um modo que permite representar cada dígito usando 4 *bits*. As instruções do 2A03 são as mesmas do 6502, e o processador é *little endian*, ou seja, os dados são armazenados no endereço de memória menos significativo primeiro.

Mapa da Memória

A mémoria utilizada pelo CPU é dividida em três: memória ROM do cartucho, memória RAM da CPU e *I/O registers* (registradores de entrada e saída). A memória ROM é somente para leitura, então, quando o Nintendo é ligado com um cartucho plugado, se não houver *Memory Mapper*, todo *PRG-ROM* do cartucho é transferido para a área específica da memória da CPU. Os *I/O registers* são utilizados para comunicar com os componentes do sistema, a PPU e dispositivos de controle. Essa comunicação é feita por meio de um barramento de dados que transmite números de 8 *bits*. Há também o

barramento de controle, utilizado para comunicar os componentes sobre algumas *flags* de controle, transmite números de 8 *bits*, e o barramento de endereço, que informa o endereço que será feita a leitura ou a escrita, transmite 16 *bits*.

A memória do CPU pode armazenar até 64KB com endereços entre $0000 - $FFFF, algumas áreas da memória são espelhadas, isso significa que o mesmo conteúdo é escrito em mais de um endereço da memória. A Figura 3 mostra como é organizada a memória a da CPU, a área em que cada dado específico é armazenado.

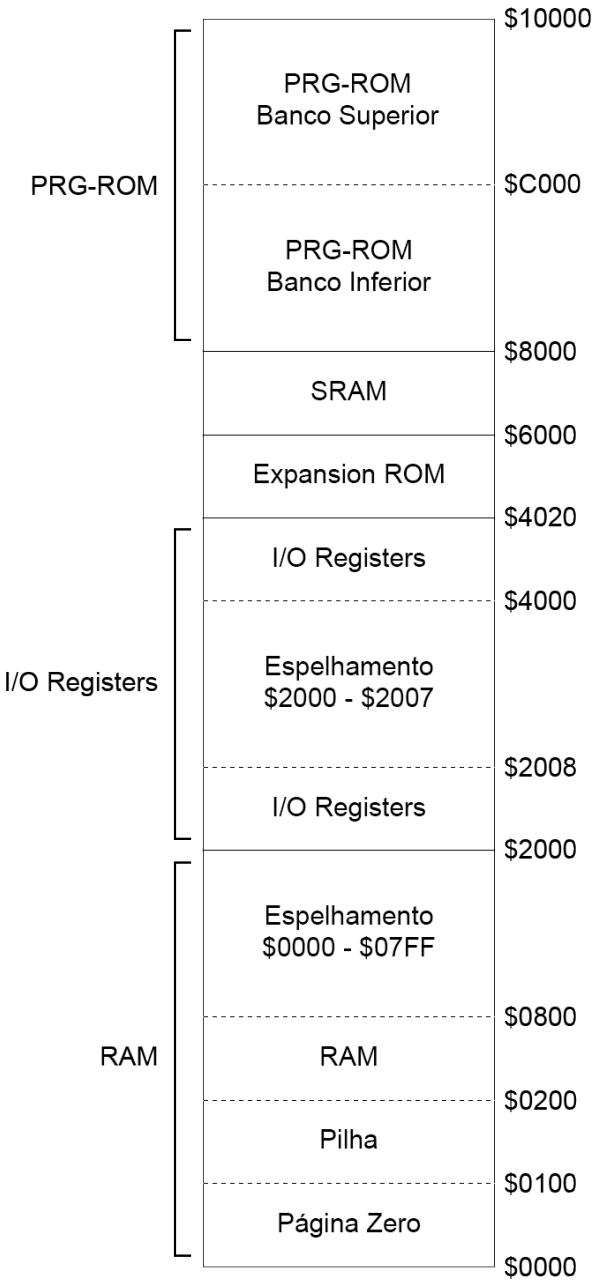

Figura 3. Mapa da memória da CPU.

Do endereço $0000 até $2000 encontra-se a memória RAM, composta por Página Zero, Pilha e RAM. Página Zero, $0000 - $00FF, é utilizado por alguns modos de endereçamento para uma execução mais rápida. Entre os endereços $0100 - $01FF ficam armazenados os dados que são da pilha. No restante de espaço usado pela memória RAM, $0200 - $1FFF, é utilizado para vários propósitos, como armazenamento de algumas variáveis e *arrays*, ou armazenamento temporário de alguns dados a ser transferido para outro local. Os dados armazenados entre $0000 - $07FF são espelhados três vezes entre $0800 - $1FFF.

Na região de $2000 - $401F estão localizados os *I/O registers*. Os 8 *bytes* iniciais, presentes entre $2000 - $2007, são utilizados para os *I/O registers* que se comunicam com o PPU, os *bytes* dessa região são espelhados a cada 8 *bytes* na região $2008 - $3FFF. Os *bytes* restantes entre a região $4000 - $401F são utilizados como *I/O registers* da APU (*Audio Processing Unit*).

A região de memória $4020 - $6000 é chamada *Expansion ROM*, e é utilizada por alguns *Memory Mappers*. Cada *Memory Mapper* utiliza essa região de forma diferente, sendo uma região de memória de uso opcional. Em seguida há a SRAM (*Save RAM*), localizada entre $6000 - $7FFF, é utilizada para armazenar o conteúdo da memória RAM presente no cartucho, caso exista.

A partir de $8000 é utilizado para armazenar o PRG-ROM que é lido do cartucho ao ligar o videogame, é dividido em dois bancos de 16 KB. Se o jogo contém apenas um banco de 16 KB, ele irá carregar em ambas as regiões. Jogos com dois bancos de 16 KB irão carregar um na região $8000 e outro na $C000. Se o jogo possui mais de 2 bancos será necessário utilizar algum *Memory Mapper*.

Registradores

No NES existem 6 registradores ao todo. São 3 registradores de propósito especial, *Program Counter*, *Stack Pointer* e *Status Register*, cada um desses registradores tem uso específico. E 3 registradores para propósito geral, *Accumulator* e *Index Register X e Y*, que são utilizados para armazenar dados ou controlar informação.

O *Program Counter* (PC) é um registrador de 16 *bits* e armazena o endereço da próxima instrução a ser utilizada. O valor do *Program Counter* é atualizado sempre que uma instrução é executada, e quase sempre vai para a próxima instrução da sequência. O valor do registrador pode ser afetado por instruções de *branches* (instruções que desviam para outra instrução dependendo da condição), de *jumps* (instruções que pulam para outra instrução) ou interrupções da CPU.

O *Stack Pointer* (SP) é um registrador de 8 *bits* que aponta para o topo da pilha. A pilha encontra-se entre os endereços $0100 - $01FF e é *top-down*, ou seja, conforme os valores vão sendo inseridos na pilha, o valor do SP vai sendo decrementado, e quando os valores vão sendo retirados da pilha, SP vai sendo incrementado. O valor do *Stack Pointer* quando a pilha esta vazia é $01FF.

O registrador de status, conhecido como *Processor Status* (P) tem 8 *bits* de tamanho e armazena 7 *flags* utilizada pela CPU.

- *Carry Flag* (C) - A *Carry Flag* é definida como 1 se a última instrução resultar em *overflow* ou um *underflow*. Por exemplo, um *byte* pode armazenar no máximo

o valor 255, se somarmos 1 a esse valor ocorrerá um *overflow* e a *Carry Flag* será definida. Com essa *flag* podemos calcular um segundo *byte*. É representado pelo *bit* 0 do registrador P.

- *Zero Flag* (Z) - A *Zero Flag* é definida como 1 sempre que a última instrução resulta em um zero. É representada pelo *bit* 1.
- Desabilitar Interrupção (I) - Quando está *flag* estiver definida como 1, o sistema não irá responder a interrupções IRQ. Utiliza o *bit* 2.
- *Decimal Mode* (D) - Essa *flag* é utilizada para trocar o processador para o modo BCD (*Binary-coded Decimal*). Na CPU utilizada pelo NES, o 2A03, esse modo não é compatível. Então mesmo sendo possível definir esse *bit*, ele não será usado. Representado pelo *bit* 3.
- *Break* (B) - Quando está *flag* está em 1, indica que uma instrução BRK foi executada, e uma interrupção IRQ irá ocorrer. É representado pelo *bit* 4.
- *Overflow* (V) - Quando ocorrer um complemento de dois cujo resultado não foi esperado pela instrução anterior, essa *flag* é definida como 1. Podemos utilizar como exemplo a soma de dois números positivos, 64 e 64, como o tamanho desse inteiro é de 1 *byte* o valor do resultado será -128, que é um resultado não esperado, já que era esperado 128. *Bit* 6 do registrador P.
- *Negative* (N) - É definido como 1 sempre que a instrução anterior resulta em um valor negativo, ou seja, o *bit* 7 do valor é 1. É o *bit* 7 do registrador.

O *bit* 5 do registrador de status não é utilizado por nenhuma *flag*.

O *Accumulator* (A) tem tamanho de 8 *bits* e é utilizado para armazenar o resultado de operações aritméticas e lógicas, ou definido com um valor vindo da memória.

O registrador *Index Register X* (X) de 8 *bits* é utilizado como um contador ou como um *offset* para alguns modos de endereçamento. O X também poder ser definido com um valor vindo da memória, e poder ser usado para obter ou colocar o valor do *Stack Pointer*.

Index Register Y (Y) também é utilizado como contador e como *offset* de modos de endereçamento. Mas diferente do X, o Y não afeta o *Stack Pointer*.

Modo de Endereçamento

Os modos de endereçamento são métodos utilizados pelas instruções para obter endereços de acesso a memória. A CPU do NES possui 13 diferentes modos de endereçamento, e eles podem utilizar tanto registradores quanto a memória para obter os endereços.

1. ***Implied***
   Utilizado por instruções que não manipulam endereços, como a instrução NOP, que não possui nenhum operando.

2. ***Immediate***
   Instruções que utilizam o modo de endereçamento *Immediate* recebe um valor que é utilizado diretamente pela instrução. AND #$12

3. ***Zero Page***
   O modo de endereçamento *Zero Page* recebe somente um operando, presente no endereço da memória do programa PC+1, esse operando serve como um ponteiro para o endereço na região da memória *Zero Page* ($0000 - $00FF). Podemos ver como o *Zero Page* funciona na imagem Figura 4.

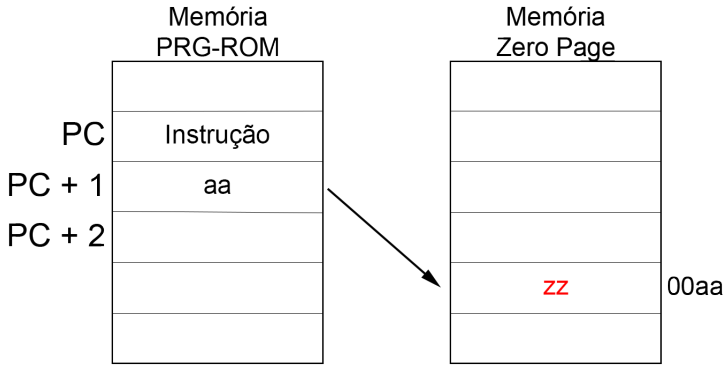

Figura 4. Modo de endereçamento *Zero Page*

4. ***Zero Page Indexed***
   Similar ao modo de endereçamento *Zero Page*, mas adiciona o conteúdo do registrador X ou do registrador Y ao ponteiro para o endereço na região da memória *Zero Page*. O registrador Y só pode ser utilizado pelas instruções LDX (*Load X Register*) e STX (*Store X Register*). O funcionamento do *Zero Page Indexed* está descrito na Figura 5.

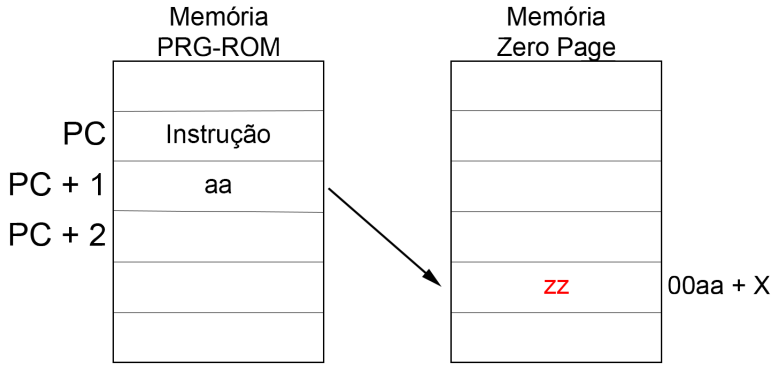

Figura 5. Endereçamento *Zero Page Indexed* adicionando registrador X

Quando o valor da soma for maior que $00FF é feito uma volta para que o endereço sempre esteja dentro do *Zero Page*. Um exemplo é ao adicionar o valor $0001 ao valor $00FF, o resultado será $0000, e não $1000, pois no caso desse *Memory Mapper*, deve ficar sempre na região do *Zero Page*.

5. ***Absolute***
   O modo de endereçamento *Absolute* recebe dois operandos, e o ponteiro para o endereço é obtido com a concatenação desses valores, o primeiro *byte* é o menos significativo. Podemos ver como o *Absolute* funciona na imagem Figura 6.

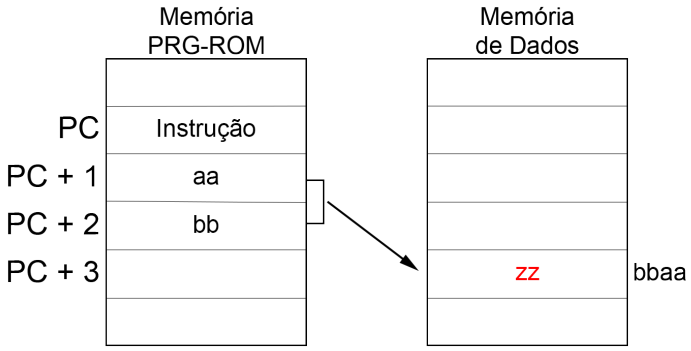

Figura 6. Endereçamento *Absolute*

6. ***Absolute Indexed***

   É similar ao *Zero Page Indexed*, redebe dois operandos, concatena os dois para obter o ponteiro e adiciona o valor do registrador X ou Y ao ponteiro. Podemos ver como o *Absolute Indexed* funciona na Figura 7.

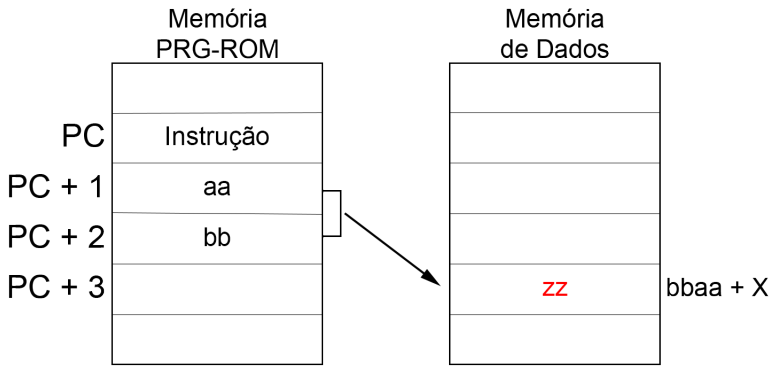

Figura 7. Endereçamento *Absolute Indexed* adicionando registrador X

7. ***Accumulator***

   Utiliza o valor do registrador *Accumulator* diretamente como o operando da instrução. É utilizado nas instruções de *Shift* (deslocamento de *bits*).

8. ***Indirect***

   O modo de endereçamento *Indirect* recebe dois operandos, concatena os operandos para formar um endereço de 16 *bits*, então utiliza esse endereço para encontrar mais dois valores, concatena-os para achar o dado utilizado pela instrução. Podemos ver como o *Indirect* funciona na imagem Figura 8.

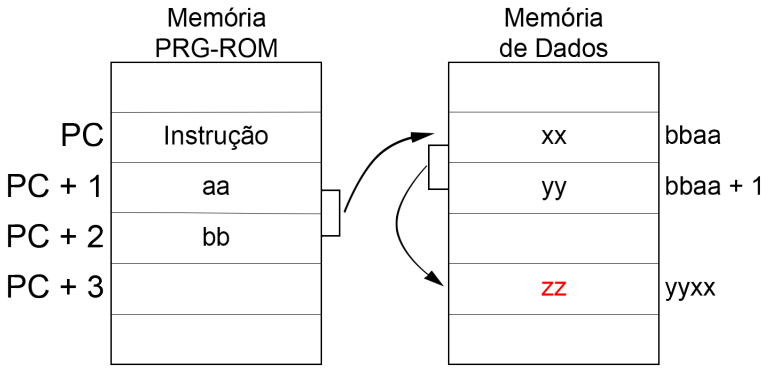

Figura 8. Endereçamento *Indiret*

9. ***Indexed Indirect***

   Esse método utiliza um operando e adiciona o valor do registrado X, esse valor deve acessar a região *Zero Page* da memória. É obtido um endereço de 16 *bits* que aponta para o endereço utilizado pela instrução. Esse modo de endereçamento também é chamado *Pre-Indexed Indirect*. Podemos ver como o *Indexed Indirect* funciona na imagem Figura 9.

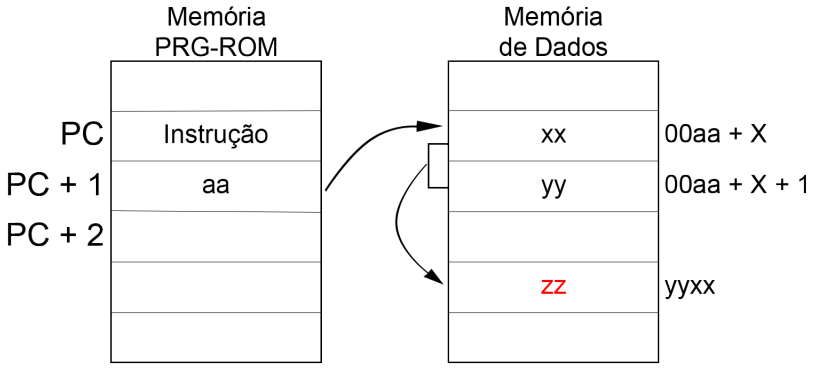

Figura 9. Endereçamento *Indexed Indiret*

10. ***Indirect Indexed***

    Esse modo de endereçamento é parecido com o *Indexed Indirect*, e é conhecido como *Pos-Indexed Indirect*. Neste método é somado o valor do registrador Y, e a soma é realizada depois que o endereço de 16 *bits* é obtido. Podemos ver como o *Indirect Indexed* funciona na imagem Figura 10.

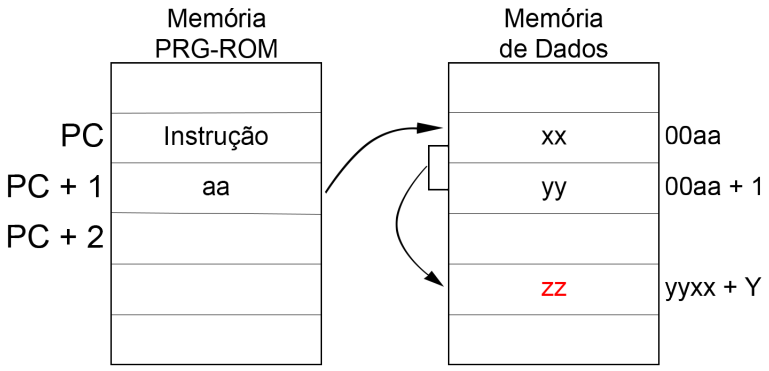

Figura 10. Endereçamento *Indiret Indexed*

11. ***Relative***
    É um modo de endereçamento utilizado exclusivamente para instruções de desvio condicional. Recebe um operando e, ao ter sua condição satisfeita soma o operando ao *Program Counter*. O *Program Counter* soma 2 independente da decisão tomada. Uma instrução de desvio condicional pode avançar ou retroceder, então o registrador PC aceita valores entre -128 até 127. Caso seja necessário um deslocamento maior que esse, é usado um desvio para um inscrição de salto.

Instruções

O 6502 possui 56 instruções diferentes, mas algumas utilizam diferentes modos de endereçamento, então no total são 151 combinações de instruções válidas.Cada instrução utiliza no máximo 3 *bytes*, um deles sendo da própria instrução. Depois da execução de cada instrução, é adicionado ao *Program Counter* o tamanho da instrução, para que ele avance para a próxima instrução. As instruções podem ser divididas em 7 grupos de diferentes. São eles:

- Operações Aritméticas - Realizam operações aritméticas nos registradores e na memória. Exemplo, ADC (Adição), SBC (Subtração).
- Operações Lógicas - Realizam operações de lógica em valores armazenados no *Accumulator* ou na memória. Exemplo, AND (Realiza um *and* da memória com o *Accumulator*).
- Operações de Incremento e Decremento - Incrementa ou decrementa em 1 nos registradores X ou Y ou em algum valor da memória. Exemplo, INX (Incrementa 1 no registrador X).
- Operações da Pilha - Realiza operações de empilhamento ou desempilhamento. Exemplo, PHA (Empilha o conteúdo de *Accumulator*) , PLP (Desempilhamento e armazena no *Processor Status*).
- Operações de *Load/Store* - Carrega ou armazena um valor na memória. Exemplo, LDX (Carrega memória no registrador X), STA (Armazena o conteúdo do *Accumulator* na memória).
- Operações de Transferência - Copia o conteúdo de um registrador para outro. Exemplo, TSX (Copia *Stack Pointer* para o registrador X).
- Operações de Deslocamento - Desloca os *bits* do registrador *Accumulator* ou da memória para a direita ou para a esquerda. Exemplo, LSR (Desloca para a direita 1 *bit*).

Interrupções

Interrupções geralmente são geradas por *hardware*, servem para interromper a execução sequencial do programa para realizar os procedimentos da interrupção. O NES tem 3 interrupções diferentes, NMI, IRQ e *Reset*. Quando uma interrupção é gerada, o NES empilha os registradores *Program Counter* (PC) e *Processor Status* (P) na pilha, carrega o endereço da rotina da interrupção no PC, executa as instruções, executa um RTI (*ReTurn from Interrupt*) que desempilha o PC e o P, e volta a execução do programa.

A interrupção IRQ é gerada por alguns mapeamentos de memória, mas pode ser gerado devido ao uso da instrução BRK (*Break*). Quando um IRQ ocorre, o sistema salta para os endereços $FFFE - $FFFF.

A interrupção NMI é gerada pela PPU, para marcar o inicio do *V-Blank*, no final de cada quadro. Quando um NMI ocorre, o sistema salta para os endereços $FFFA - $FFFB.

A interrupção *Reset* é a interrupção que tem a maior prioridade, seguida pela NMI e só então a IRQ. É gerada sempre que o sistema inicia ou reinicia, zera todos os registradores e as memórias do NES.

PPU

A Ricoh ltda também foi responsável pela fabricação da PPU (Picture Processing Unit), conhecido como 2C02 [5]. A comunicação entre a CPU e a PPU ocorre por meio de alguns registradores de entrada e saída, localizados na memória da CPU. Esses endereços servem para o controle de algumas configurações da PPU, como *pixel* em que será desenhado algum *sprite*, se ocorrerá alguma interrupção, etc. Os valores escritos nesses endereços são enviados à PPU de acordo com suas necessidades. Como os endereços da PPU são de 16 *bits* mas os valores do registrador de entrada e saída são de 8 *bits*, é feita duas escritas para definir um endereço.

Para renderizar os *pixels*, a PPU utiliza o método de *scanline*, que consiste em renderizar uma linha de cada vez. Apesar da tela do NES ter 256x240, é executado 262 *scanlines*. Do 0 ao 239 são utilizados para renderizar os *pixels*. No 240 não faz nada. Do 241 até 260 é quando ocorre o *V-Blank*, que é o momento em que a PPU não está em uso, então é possível acessar e transferir dados para a memória da PPU. Se a CPU tentar transferir dados em outro momento, a imagem da tela conterá vários *tiles* incorretos. O que marca o inicio do *V-Blank* é a interrupção NMI.

Mapa da memória

A PPU possui uma memória interna chamada VRAM (*Video RAM*) e pode endereçar até 64 KB, apesar de possuir somente 16 KB de memória físico. A VRAM armazena Tabelas de Padrões, Tabelas de Nomes e Paletas de Cores.

Do endereço $0000 até $1FFF são armazenadas as Tabelas de Padrões, que são padrões que definem as formas utilizadas nos *tiles* (caixa de *pixels* utilizada para formar uma imagem) para formar o *background* (plano de fundo) e os *sprites* (personagens e objetos que podem interagir com o jogador).

Entre os endereços $2000 - $2FFF são armazenados as Tabelas de Nomes, que são matrizes de números de *tiles*, que apontam para os *tiles* armazenados na região de memória das Tabelas de Padrões. Do endereço $3000 até $3EFF encontra-se o espelhamento da região de Tabelas de Nomes.

As Paletas são armazenadas entre $3F00 - $3FFF, que são as cores que serão utilizadas no *background* e nos *sprites* da tela atual. Do endereço $4000 para cima, é utilizado como espelhamento, pois esse endereço não existe fisicamente.

O NES tem uma memória separada utilizada para armazenar os atributos dos *sprites* chamada SPR-RAM e, é utilizada pela PPU para desenhar os *sprites*. Os *sprites* são armazenados nas Tabelas de Padrões. O mapa da VRAM e todas as regiões específicas podem ser vistas na Figura 11.

A PPU tem um mecanismo para transferência dos dados da SPR-RAM da CPU para a PPU, e é possível transferir 256 *bytes* de uma vez ao invés de 1 *byte* por vez. O mecanismo é chamado de DMA (*Direct Memory Access*), e consiste em passar o *byte* mais significativo de um endereço de memória da CPU por meio do *I/O register* $4014, então serão transferidos 256 *bytes* da CPU entre os endereços $XX00 - $XXFF por meio do barramento de dados para a PPU, armazenando na SPR-RAM.

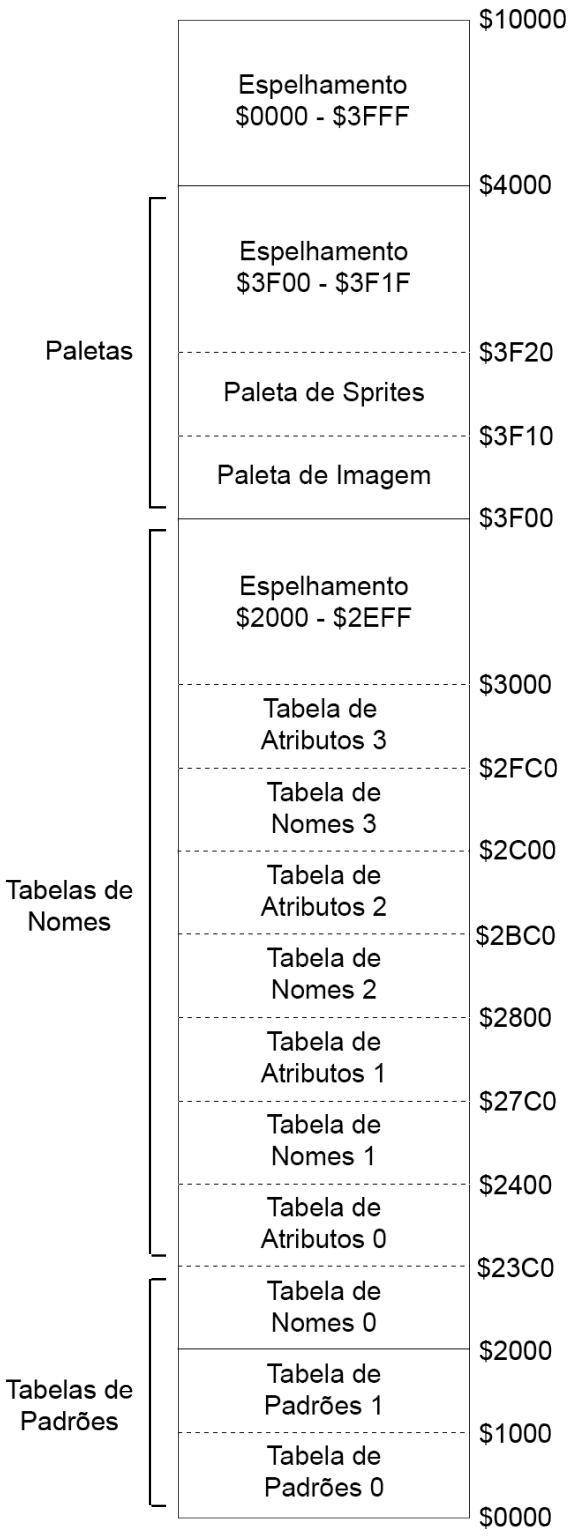

Figura 11. Mapa da memória da PPU.

Registradores

Os registradores utilizados pela PPU estão localizados na memória da CPU, entre os endereços $2000 - $2007, são os *I/O registers*. Também é utilizado o registrador no

endereço $4014, utilizado para acesso direto a memória. Na tabela abaixo mostra as funções de todos os *I/O registers* utilizados pela PPU.

A PPU é controlada pela CPU por meio dos registradores de controle 1 e 2 ($2000 e $2001), cada *bit* dos registradores é responsável por uma configuração específica da PPU. O registrador $2002, chamado de registrador de *Status*, é utilizado para manter a CPU e a PPU sincronizadas, para que a escrita da CPU seja feita no tempo correto. Os *sprites* são escritos na PPU por meio dos registradores $2003 e $2004, o primeiro registrador passa o endereço da SPR-RAM que será utilizado para escrever o dado vindo do segundo registrador.

O registrador $2005 é utilizado para alterar a posição do *scroll*, que diz a PPU qual *pixel* da Tabela de Nomes estará no topo esquerda da tela renderizada. Geralmente é escrito nesse registrador durante *V-Blank*, para que no próximo quadro comece a ser renderizado a partir da posição desejada, mas pode ser modificado durante a renderização para fazer o efeito de tela dividida. É feita duas escritas nesse registrador, na primeira escrita é passada a posição X, e na segunda a posição Y.

A transferência entre a CPU e a PPU é feita pelos registradores $2006 e $2007, o endereço de escrita é passado para o por meio de 2 escritas no registrador $2006, em seguida é feita várias escritas no registrador $2007 com os valores a serem transferidos para a VRAM. A cada escrita feita, é incrementado o endereço de escrita de acordo com o valor passado no Registrador de Controle 1, Bit 2. Na Tabela 2 consta mais detalhes sobre os *I/O registers*.

Tabela 2. Funções dos *I/O registers utilizados pela PPU*

Endereço	**Nível de acesso**	**Descrição**
$2000	Escrita	Registrador de Controle da PPU 1: Bits 0-1 - Endereço da Tabela de Nomes, é selecionada uma entre as quatro Tabelas de Nomes disponíveis. $2000 (0), $2400 (1), $2800 (2), $2C00 (3). Bit 2 - Específica a quantidade que irá incrementar no endereço, 1 se o *bit* for 0 ou 32 se for 1. Bit 3 - Identifica em qual Tabela de Padrões os *sprites* estão armazenados, $0000 (0) ou $1000 (1). Bit 4 - Identifica em qual Tabela de Padrões o *background* está armazenado, $0000 (0) ou $1000 (1). Bit 5 - Específica o tamanho dos *Sprites* em *pixels*, 8x8 se o *bit* for 0, 8x16 se for 1. Bit 6 - Não é utilizado pelo Nes. Bit 7 - Indica quando a interrupção NMI deve ocorrer após um *V-Blank*.
$2001	Escrita	Registrador de Controle da PPU 2: Bit 0 - Indica se o sistema em cores (0) ou modo monocromático (1). Bit 1 - Indica quando mostrar os 8 *pixels* da esquerda do *background* (1), e quando escondê-los (0). Bit 2 - Indica quando mostrar os 8 *pixels* da

		esquerda dos *sprites* (1), e quando escondê-los (0). Bit 3 - Se o *bit* é 0, o *background* não será exibido. Bit 4 - Se o *bit* é 0, os *sprites* não serão exibidos. Bit 5-7 - Indica a cor do *background*, caso esteja no modo monocromático, ou a intensidade das core, no modo de cor.
$2002	Leitura	Registrador de Status da PPU: Bit 4 - Se o *bit* for 1, indica que escrita no VRAM deve ser ignorado. Bit 5 - Se o *bit* for 1, indica que há mais de 8 *sprites* por *scanline*. Bit 6 - Define 1 quando um *pixel* não transparente do *sprite 0* sobrepões um *pixel* não transparente do *background*. Bit 7 - Indica quando ocorre *V-Blank*.
$2003	Escrita	Endereço do registrador SPR-RAM: Armazena o endereço do SPR-RAM a ser acessado na próxima escrita pelo $2004.
$2004	Escrita	SPR-RAM registrador de entrada e saída: Escreve um *byte* no endereço indicado pelo $2003.
$2005	2 x Escrita	Posição do *pixel* que estará no topo esquerdo da tela renderizada. Utilizado para *scroll* (rolagem) da tela.
$2006	2 x Escrita	Endereço da PPU que será utilizada para fazer escrita de dados vindos da CPU.
$2007	Leitura/Escrita	Escreve ou lê dados da VRAM. Depois de escrito, é feito um incremento no endereço do valor definido no registrador $2000.
$4014	Escrita	*Byte* mais significativo do DMA do SPR-RAM.

Paleta de Cores

O NES possui um sistema de paletas de 64 cores, e para cada quadro é escolhido 16 cores para os *tiles* do *background* e 16 cores para os *tiles* dos *sprites*. O NES não possui um sistema de cores específico, podendo variar de acordo com a televisão utilizada. A Tabela 3 indica como é organizado as 32 cores disponíveis na memória [6].

A primeira coluna da tabela é para os endereços que são utilizados para a cor de fundo padrão do quadro, ou seja, caso o *pixel* for transparente, a cor do fundo será da cor apontada pelo endereço armazenado em $3F00. Os endereços $3F04, $3F08, $3F0C, $3F10, $3F14, $3F18, $3F1C são espelhamentos do endereço $3F00.

Tabela 3. Paletas de Cores

$3F00 Cor de fundo (transparencia)	$3F01 Cor 1	$3F02 Cor 2	$3F03 Cor 3	Paleta do *Background* 0
$3F04 Espelhamento $3F00	$3F05 Cor 1	$3F06 Cor 2	$3F07 Cor 3	Paleta do *Background* 1
$3F08 Espelhamento $3F00	$3F09 Cor 1	$3F0A Cor 2	$3F0B Cor 3	Paleta do *Background* 2
$3F0C Espelhamento $3F00	$3F0D Cor 1	$3F0E Cor 2	$3F0F Cor 3	Paleta do *Background* 3
$3F10 Espelhamento $3F00	$3F11 Cor 1	$3F12 Cor 2	$3F13 Cor 3	Paleta dos *Sprites* 0
$3F14 Espelhamento $3F00	$3F15 Cor 1	$3F16 Cor 2	$3F17 Cor 3	Paleta dos *Sprites* 1
$3F18 Espelhamento $3F00	$3F19 Cor 1	$3F1A Cor 2	$3F1B Cor 3	Paleta dos *Sprites* 2
$3F1C Espelhamento $3F00	$3F1D Cor 1	$3F1E Cor 2	$3F1F Cor 3	Paleta dos *Sprites* 3

Tabela de Padrões

As Tabelas de Padrões são áreas da memória utilizadas para armazenar formatos de *tiles* utilizados para formar o *background* e os *sprites*. Cada *tile* na Tabela de Padrões é representado por 16 *bytes* divididos em duas partes de 8 *bytes*. A primeira parte controla o *bit* 0 da cor do tile, e a segunda parte controla o *bit* 1 da cor. Na Tabela 4 está exemplificado como é formado cada *Tile*.

Tabela 4. Exemplo Tabela de Padrões [7]

$X000	0	0	0	0	0	0	0	0	$X008	0	0	1	1	1	1	0	0
	0	0	0	0	0	0	0	0		0	1	1	1	1	1	1	0
	0	0	0	0	0	0	0	0		0	1	1	1	1	1	1	0
	0	0	0	0	0	0	0	0		1	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0		1	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0		1	1	1	1	1	1	1	1
	0	1	1	1	1	1	1	0		0	1	0	0	0	0	1	0
$X007	0	0	1	1	1	1	0	0	$X00F	0	0	0	0	0	0	0	0

0	0	2	2	2	2	0	0
0	2	2	2	2	2	2	0
0	2	2	2	2	2	2	0
2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2
0	3	1	1	1	1	3	0
0	0	1	1	1	1	0	0

É utilizado o *bit* da primeira tabela como *bit* menos significativo e o da segunda

como o mais significativo entre os dois. Como é utilizado dois *bits*, é possível três resultados diferentes, 0, 1, 2 ou 3. Esses dois *bits* são os dois menos significativos entre os 4 *bits* utilizados para definir a Paleta de Cores e as cores.

Os *bits* obtidos na Tabela de Padrões irão representar a cor, que será uma das quatro cores disponíveis em uma Paleta de Cores, onde 0 indica que o *pixel* é transparente e utilizara a cor padrão de fundo e 1, 2 ou 3 representa o endereço de uma cor específica, como consta na tabela 3.

Na terceira tabela é utilizada a concatenação de todos os *bits* de ambas as tabelas para formar um padrão de imagem.

Tabela de Nomes

Uma Tabela de Nomes é uma matriz de números de *tiles* apontando para os padrões armazenados nas tabelas de padrões. Tabelas de nomes são divididas em 30 colunas e 32 linhas de 8x8 *pixels*. Isso da uma resolução de 30x32 *tiles* ou 240x256 *pixels*. Cada *tile* é representado por um valor entre 0 e 255.

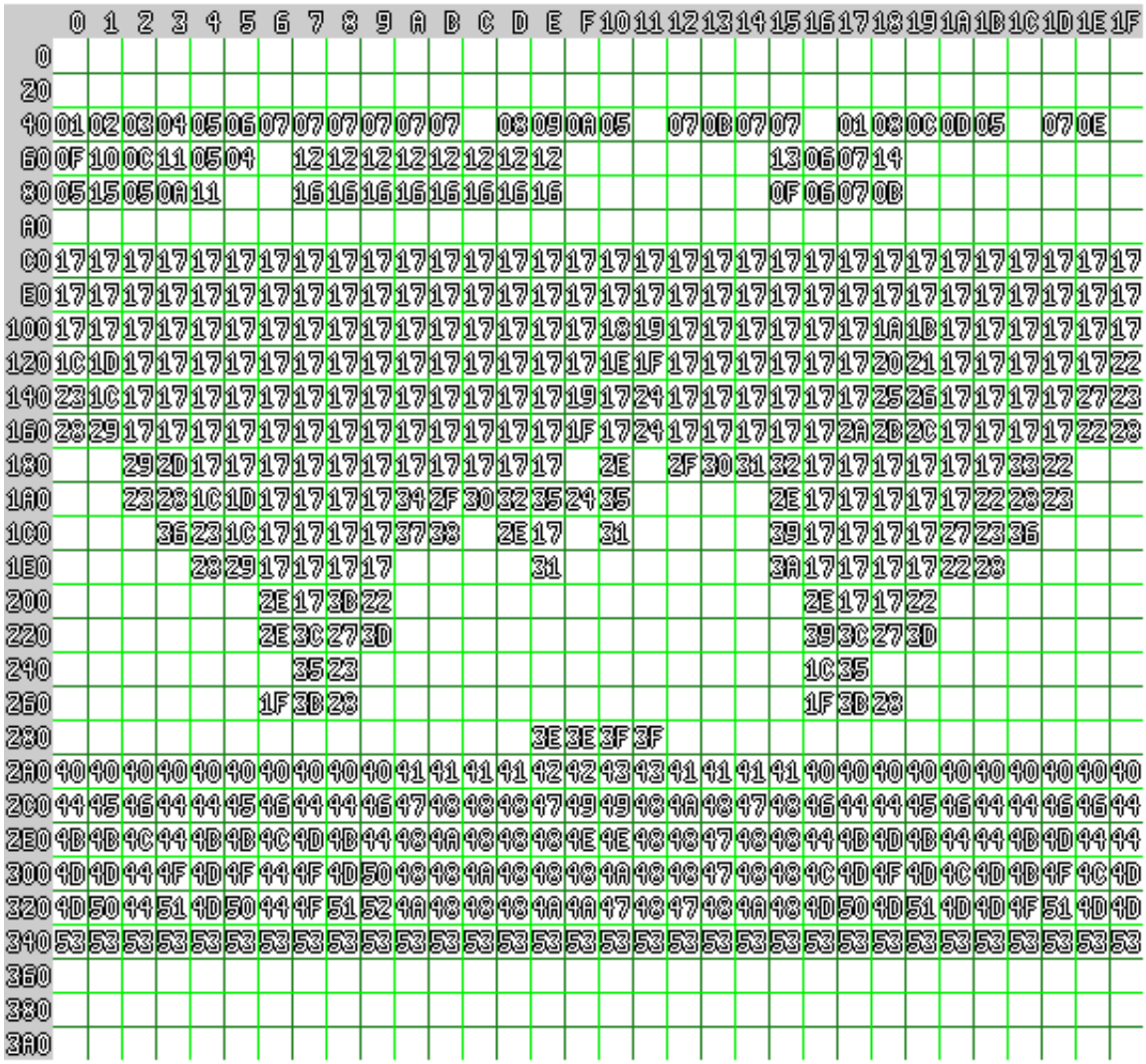

Figura 12. Tabela de nomes com ponteiros de cada *tile*.[8]

Cada *tile* possui um *byte*, então a Tabela de Nomes inteira contém 960 *bytes*. Cada *byte* representa um endereço da Tabela de Padrões, como consta na Figura 12.

Uma Tabela de Nomes também é dividida em blocos, cada bloco é composto por 2x2 *tiles*. Na Figura 13 está uma cena da abertura do jogo Castlevania (e o resultado final depois dos cálculos utilizando a Figura 12) dividida em blocos e em *tiles*. Os blocos estão divididos usando as linhas verde claro, e os *tiles* usando as linhas verde escuro.

Figura 13. Tabela de nomes dividida em blocos e *tiles*.[8]

Os números nos eixos indicam a posição de cada *tile* na memória, soma-se o eixo X com o eixo Y, e adicionamos ao endereço da Tabela de Nomes que está sendo utilizada no momento, de acordo com o *I/O register* $2000, e obtemos o endereço do *tile* específico.

Tabela de Atributos

A Tabela de Nomes é responsável por controlar quais padrões da Tabela de Padrões será utilizado em que posição da tela, mas não controla cores. A Tabela de Atributos é encarregada de controlar qual Paleta de Cores disponível na memória cada *tile* irá utilizar.

Cada Tabela de Atributos possui 64 *bytes*, cada *byte* corresponde a uma região de

32x32 *pixels*, ou 4x4 *tiles* ou 2x2 blocos. Cada *byte* é dividido em valores de 2 *bits* que representam a Paleta de Cor usada em um determinado bloco. Na Figura 14 é mostrado como a Tabela de Atributos funciona.

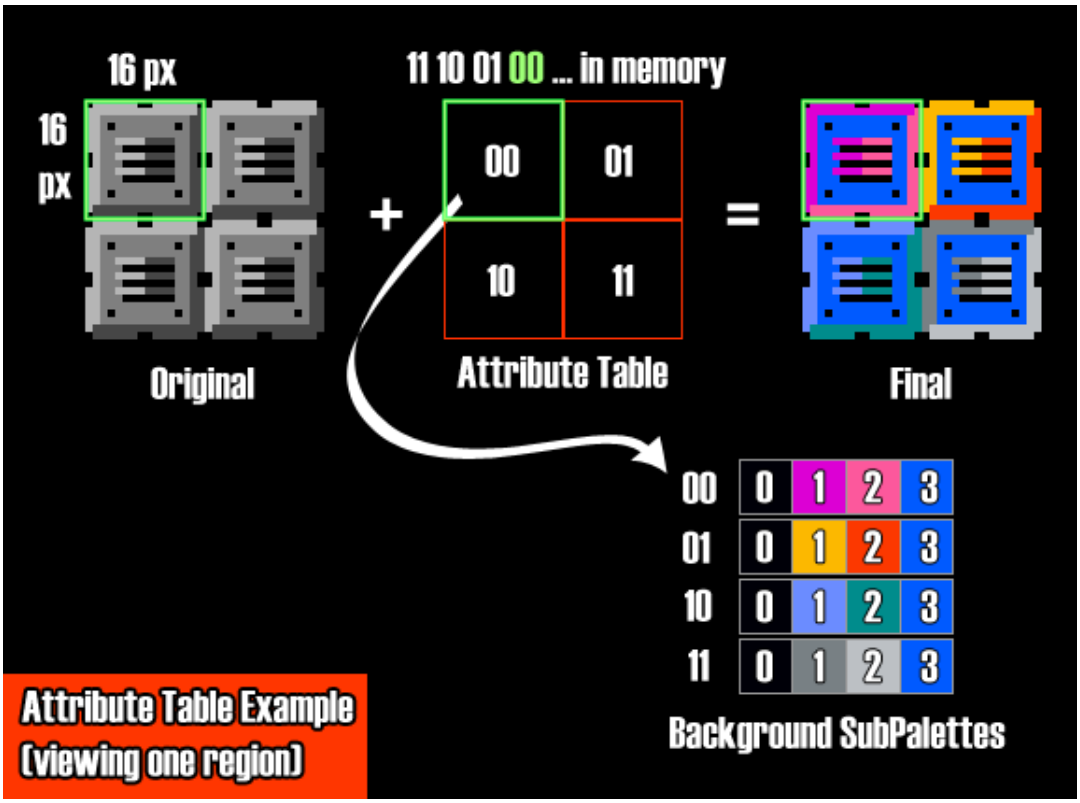

Figura 14. Exemplo do funcionamento da Tabela de Atributos.[9]

Então utilizando a tabela de padrões, a tabela de nomes e a tabela de atributos, conseguimos um *background* colorido. A tabela de nomes especifíca quais *tiles* utilizar em qual posição da tela, a tabela de atributos especificá qual paleta de cor utilizar para cada bloco, e a tabela de padrões especificá qual cor utilizar para cada *pixel*, de acordo com a paleta de cores atribuída a esse bloco.

Scrolling

Scrolling é um efeito utilizado em jogos 2D que, conforme o jogador anda com o personagem, o *background* se movimenta junto. O background pode se mover tanto horizontalmente quanto verticalmente. Para que esse efeito seja possível, é utilizado duas tabelas de nomes ao mesmo tempo para compor a imagem.

A imagem final começa na primeira Tabela de Nomes e se estende por meio da segunda. O *scrolling* é executado na forma de laço, então quando a posição de rolagem chega até a segunda Tabela de Nomes, começa a mostrar a primeira novamente ao final da tela. Na Figura 15 mostra o conteúdo de duas tabelas de nomes durante o *scrolling*, entre as linhas azuis está o conteúdo que é mostrado na tela, e a linha vermelha é a divisão entre as tabelas de nomes.

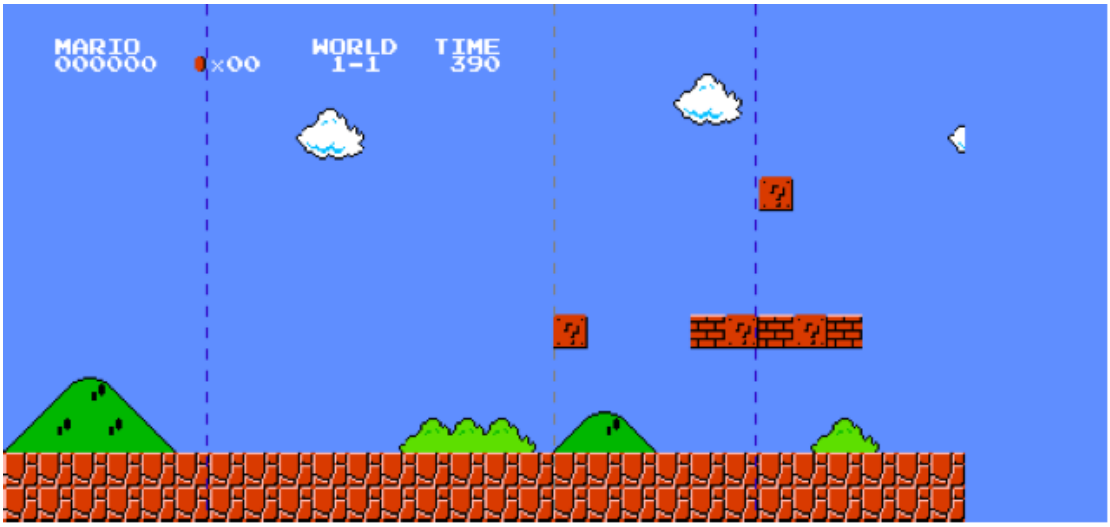

Figura 15. Exemplo de *scrolling* com uma cena do Super Mario Bros.[5]

É feita duas escritas no *I/O register* $2005 para informar a PPU qual o *pixel* que estará na posição esquerda superior da tela. Assim, ao escrever nesse registrador sempre que a PPU for renderizar um quadro, terá o efeito de que a tela se move constantemente. A escrita é sempre feita durante o V-Blank, para não interferir no quadro que já está sendo renderizado.

Alterações nas tabelas de nomes podem ser feitas durante o *scrolling*, desde que feitas durante o V-Blank e na área que não está sendo renderizada no momento. Essas alterações são sempre feitas logo antes de serem exibidas na tela.

Sprites

Qualquer coisa que se move separado do *background* é feito de *sprites*. Um *sprite* é um *tile* de 8x8 *pixels* que pode ser renderizado em qualquer posição da tela. Geralmente, objetos e personagens são feitos de vários *sprites* próximos, que formam uma imagem. Um exemplo a imagem do Megaman na Figura 16, que é formada por 10 *sprites* diferentes.

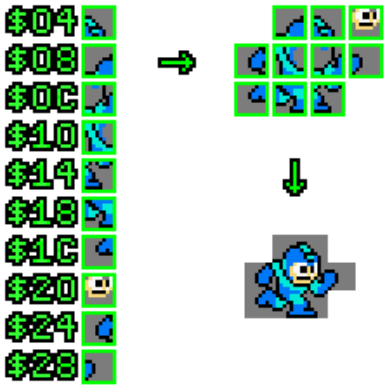

Figura 16. Exemplo de *sprites* com o personagem Megaman.[10]

Diferente das tabelas de nomes, os dados de cada *sprite* são compostos por 4 *bytes*.

1. Posição do Y - A posição vertical que o *sprite* será colocado na tela, a posição $00 é o topo da tela.
2. Número do *tile* - É o ponteiro desse *sprite* na Tabela de Padrões.
3. Atributos - Informações sobre cor e informações e exibição.
    - Bit 0-1 - Paleta de cores utilizada;
    - Bit 2-4 - Não são utilizados;
    - Bit 5 - 0 indica que está na frente do *background*, 1 indica que está atrás;
    - Bit 6 - Se definido em 1 indica que o *sprite* deve ser girado (90°) horizontalmente;
    - Bit 7 - Se definido em 1 indica que o *sprite* deve ser girado (90°) verticalmente;
4. Posição do X - A posição horizontal que o *sprite* será colocado na tela, a posição $00 é o lado esquerdo.

Na SPR-RAM podem ser armazenado até 64 *sprites*. Durante um *scanline*, a PPU verifica em toda a SPR-RAM se algum *sprite* deverá ser desenhado, ela faz isso verificando a posição do Y de todos os *sprites*.

A PPU escolhe os 8 primeiros *sprites* para serem desenhado, pois o NES não suporta mais de 8 *sprites* por *scanline*. Então renderiza eles na tela, de acordo com a prioridade de cada *sprite*. A prioridade dos *sprites* é de acordo com a ordem que eles estão armazenados na SPR-RAM, o primeiro *sprite* tem prioridade sobre os outros, então caso algum *sprite* seja desenhado sobre o outro, o que estava primeiro na SPR-RAM será desenhado na frente.

Então a PPU utiliza os 4 *bytes* da SPR-RAM referentes a cada *sprite*, para desenhar somente os *pixels* necessários no *scanline*. Esse processo é repetido para cada *scanline* toda vez que a tela for renderizada, para dar o efeito de que o personagem está se mexendo.

Para detectar colisões de *sprites* com o *background*, a PPU utiliza uma técnica chamada *Sprite 0 Hit*, que consistem em verificar, se algum *pixel* do *sprite* na posição 0 da SPR-RAM estiver fazendo uma sobreposição a algum *pixel* não transparente do *background*, e o *pixel* do *sprite* também for não transparente, ocorre uma colisão e a *flag* de *sprite 0 hit* é definida como 1. Essa técnica é fundamental para o bom funcionamento de alguns jogos, como *Super Mario Bros*.

Controle

O controle do NES possui 8 botões, A, B, *Start, Select* e quatro direcionais (cima, baixo, esquerda, direita). A comunicação entre o controle do NES e a CPU acontece pelos registradores de entrada e saída. O controle 1 utiliza o endereço $4016 e o controle 2 utiliza o $4017. Na Figura 17 mostra como é o controle do NES.

Para verificar se algum botão está pressionado, o sistema faz várias leituras no endereço $4016. Nas 8 primeiras leituras, se o *bit* menos significativo estiver definido em 1 em alguma leitura, o botão referente a leitura efetuada está pressionado. A ordem dos botões é A, B, *Select*, *Start*, Cima, Baixo, Esquerda e Direita.

Figura 17. Foto do controle do NES.[11]

3. Desenvolvimento

3.1. Bibliotecas utilizadas

Para o desenvolvimento do projeto foi necessário utilizar uma biblioteca externa chamada *PyGame* na versão 1.9.1 [12]. *PyGame* é uma biblioteca desenvolvida em C, utilizada em conjunto com a linguagem python para desenvolvimento de jogos ou aplicações multimédia. Possui um conjunto de módulos responsáveis por diversas funcionalidades de um motor de jogo, como renderização em 2D ou 3D, mecanismo de detecção de colisão, suporte a sons, inteligência artificial e manipulação de eventos, como um clique do *mouse*, ou de uma tecla do teclado.

Foi utilizado o *PyGame* somente para a renderização dos gráficos e a manipulação dos eventos do teclado. Na Tabela 5 encontram-se todas as funções do *PyGame* utilizadas no desenvolvimento do emulador.

Tabela 5. Métodos do *PyGame* utilizados no desenvolvimento do emulador.

Método	**Descrição**
pygame.event.poll()	Retorna um evento da fila de eventos.
pygame.key.get_pressed()	Retorna o estado de todas as teclas do teclado. O valor *True* significa que a tecla foi pressionada.
pygame.init()	Inicializa todos os módulos importados do *PyGame*.
pygame.display.set_mode((largura, altura))	Inicializa a tela que será exibido todo o gráfico do jogo.
pygame.Surface.fill(cor)	Preenche a tela (ou parte dela) com uma cor.
pygame.Surface.set_at((x, y), cor)	Define a cor de um único *pixel* da tela.
pygame.display.flip()	Atualiza todo o conteúdo da tela.

3.2. Programa

Para o desenvolvimento do emulador foi utilizado o Python 2.7.11 [13]. O emulador não foi testado com o Python 3, podendo não funcionar se não for utilizado a versão do Python utilizada no desenvolvimento.

O emulador foi desenvolvido em 5 classes, são elas *console* (a classe principal), *romLoader*, CPU, *joypad* e PPU, e dois arquivos, *instructions* e *addressingModes*. Na classe *console* são instanciados dois objetos das classes *romLoader* chamado *cartridge*, e a CPU, e é passado o caminho do ROM para *romLoader*. Na Figura 18 consta o diagrama de classes do emulador desenvolvido.

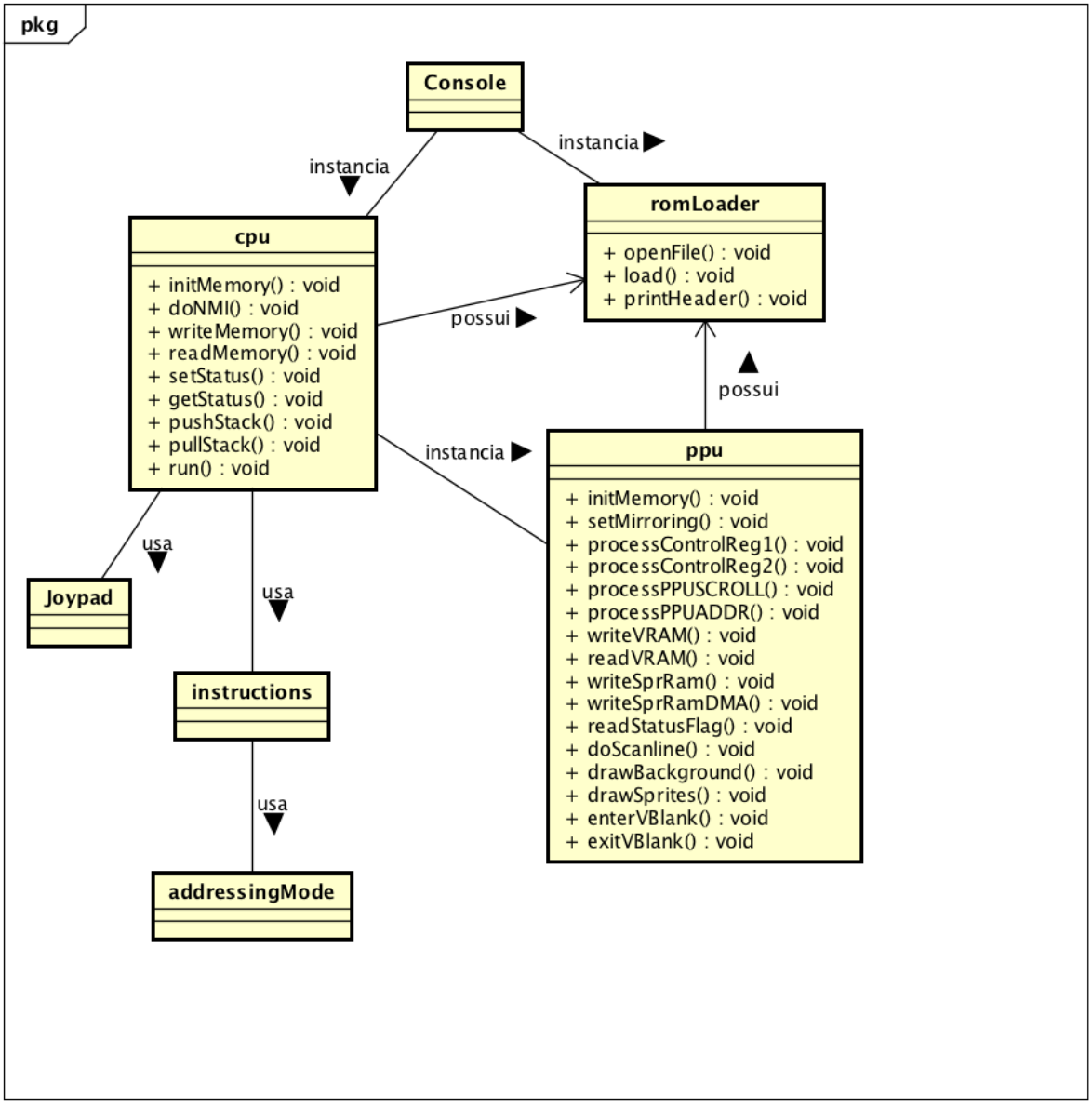

Figura 18. Diagrama de Classes do emulador

O romLoader abre e lê todo o ROM, e o divide de acordo com o modelo *iNes*.

Todos os dados são armazenados em variáveis e *arrays* no *romLoader* e são acessados pelas outras classes.

O *console* passa para o construtor da CPU o objeto *cartridge*. A CPU instancia um objeto da classe PPU e passa a instância do *romLoader* também. Ainda no construtor é inicializado todas as variáveis e *arrays*, e é chamado um método para iniciar a memória, que irá transferir os dados do *cartridge* para a memória RAM da CPU. Para as instruções é utilizado um dicionário, que é uma estrutura de dados que guarda pares de chave/valor. Então para cada instrução a chave é o número da instrução e o valor é o endereço da função da instrução.

Após o *console* instanciar a CPU, é chamado o método *run()* da CPU, que é responsável pelo laço principal do emulador, que irá executar até o encerramento do programa. Esse método faz 3 operações, verifica se alguma tecla foi pressionada, chama a instrução a ser executada de acordo com o registrador PC e retorna os ciclos que essa instrução demora para executar no NES, e, de acordo com a quantidade de ciclos que já foi executada, executa certos métodos da PPU. Cada *scanline* leva 113 ciclos da CPU, do *scanline* 0 até o 240 ele executa o método *doScanline* da PPU, quando chega ao *scanline* 241, indica que entrou no vBlank, então a CPU poderá transferir dados para o PPU nos próximos 20 *scanlines*.

O valor de 113 é utilizado para sincronizar a CPU com a PPU, a PPU funciona 3 vezes mais rapido do que a CPU, demorando 341 ciclos para executar 1 *scanline*. Então dividimos 341 por 3 para emularmos quantos ciclos a CPU terá que esperar para chamar novamente o método *doScanline*.

Sempre que ocorrer escrita nos registradores de entrada e saída, é chamado algum método da PPU para passar os dados e realizar os procedimentos necessários. Quando é necessário ler dados desses registradores, também é chamada funções da PPU para retornar os dados.

O arquivo *instructions* contém a implementação de todas as 151 instruções da CPU, e mais algumas instruções não oficiais criadas pela comunidade. Essas instruções utilizam os modos de endereçamento armazenados no arquivo *addressingModes*.

Para testar essas instruções, foi executado um ROM que testa todas as instruções. Esse ROM foi feito pela comunidade do NES. Quando a PPU ainda não está desenvolvida, é definido o endereço do registrador PC para $C000 diretamente no código fonte, e é feito um log com todas as instruções que são executadas e seus respectivos endereços no PC, e o estado de algumas flags. Esse log é comparado com um log distribuído junto com a ROM, feito por um emulador já completamente funcional. A Figura 19 mostra o ROM de teste em funcionamento.

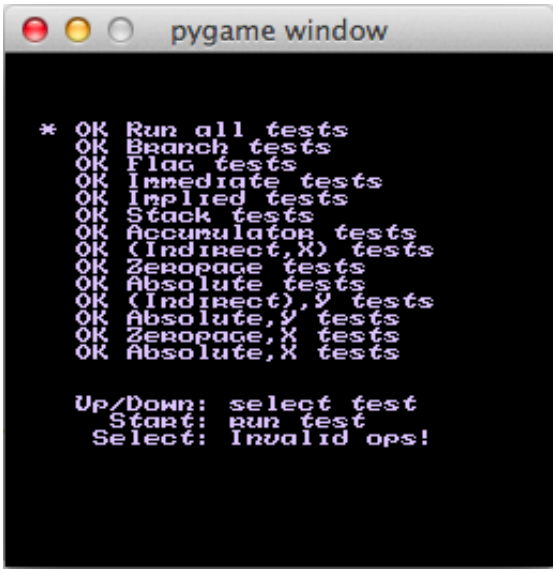

Figura 19. O ROM nestest funcionando depois da PPU estar pronta.

A PPU se comporta de maneira similar a CPU em seu construtor, inicializa todas as variáveis, *arrays* e a Paleta de Cores. As cores do NES dependem da televisão a ser utilizada, mas em um emulador ela precisa ser definida. Para definirmos a Paleta de Cores utilizamos uma lista de tuplas, cada tupla é uma cor no modelo RGB. Na Figura 20 está a tabela das cores utilizadas.

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0E
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

Figura 20. Tabela de cores e índice.[5]

O construtor chama o método *initMemory*, que carrega o resto do conteúdo do cartucho na VRAM, e define um tamanho para a tela e preenche toda a tela de preto. Então a PPU passa a operar somente quando a CPU chamar algum de seus métodos.

Os métodos mais importantes da PPU são *doScanline*, *drawBackground* e *drawSprites*. O *doScanline* é chamado pela CPU sempre que ocorre um *scanline* que deve desenhar na tela, ele verifica se pode desenhar o *background* e os *sprites*, em caso positivo, chama os métodos *drawBackground* e *drawSprites* respectivamente. Em seguida, depois de todos os 240 *scanlines*, é passado todos os *pixels* para a tela por meio de funções do *PyGame* e é atualizado a tela, para que os novos *pixels* fiquem visíveis.

Cada *pixel* da tela é representado por uma cor da Paleta de Cores, antes de desenhar todos os *pixels* na tela, essas cores são todas armazenadas em uma matriz ao longo da execução da PPU. Ao final de cada quadro, a matriz de cores estará cheia, então é

desenhado tudo na tela. A renderização do gráfico é feita dessa maneira para que haja menos chamadas a função de atualização da tela do *PyGame*.

O método *drawBackground* é responsável por definir a cor de cada *pixel* do *scanline* atual. Utilizando as Tabelas de Nomes, Tabelas de Atributos, Tabelas de Padrões e Paletas de Cores, conforme visto anteriormente. Após o calculo das cores, elas são armazenas na matriz, nas coordenadas corretas para depois serem desenhadas na tela.

O método *drawSprites* é similar ao *drawBackground*, mas desenha *sprites*. Ao final do método é sempre verificado se o *pixel* do *sprite* é transparente, caso não seja, a cor do *pixel* é armazenada sobre a cor do *background* na matriz.

Foi realizado testes com os jogos *Balloon Fight*, *Super Mario Bros*, *Tennis* e *Sqoon*. Apesar do *Super Mario Bros* apresentar a tela inicial, ele não funciona. Por outro lado, os outros 3 jogos testados funcionam corretamente, apesar da lentidão. Na Figura 21 podemos ver o jogo *Balloon Fight* sendo executado, e na Figura 22 o jogo *Super Mario Bros* parado na tela inicial.

Figura 21. O jogo *Balloon Fight* em execuçao.

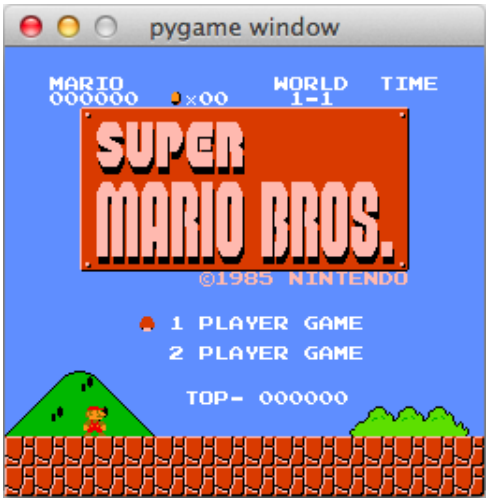

Figura 22. Tela inicial do jogo *Super Mario Bros*

4. Conclusão

Nesse projeto, foi desenvolvido um emulador de *Nintendo Entertainment System* utilizando a linguagem de programação Python. O emulador emula com sucesso alguns jogos mais simples do NES, apesar de apresentar lentidão. A lentidão ocorre devido ao fato de ser executada sobre uma máquina virtual Python e de certos trechos de código não estarem otimizados.

Foi feito a análise do emulador utilizando um software específico para Python chamado cProfiler para verificar os métodos que mais consumiram mais tempo, e foi constatado que os métodos responsáveis pela lentidão do emulador eram *doScanline* e *drawBackground*, métodos que são chamados muitas vezes durante cada quadro. Para que seja feita uma melhora significativa do emulador, deverá ser estudado a melhor abordagem para otimizar esses dois métodos.

A utilização do *PyGame* ajudou no desenvolvimento, por ser uma biblioteca extremamente fácil de ser utilizada e, como é uma biblioteca desenvolvida em C não prejudicou mais o desempenho do emulador. Além disso, o projeto foi implementado para que funcionasse diversos jogos, mas como o *hardware* do NES é complexo e o *hardware* dos cartuchos vária bastante, somente os jogos mais simples são compatíveis com o emulador.

5. Trabalhos futuros

Como possíveis trabalhos futuros, poderá ser feito otimizações no código fonte para que o emulador fique com uma velocidade razoável. Estudar se será eficiente o uso de *Threads* na CPU e na PPU. Se necessário implementar alguns métodos utilizando a linguagem de programação C.

Deverá ser implementado os *Memory Mappers* mais comuns, para que uma maior variedade de jogos sejam compatíveis com o emulador. Implementar também a pAPU (processador de áudio), a funcionalidade de *Save Game* (salvar jogo), o controle do segundo jogador e permitir que a janela do emulador seja redimensionada.

Referências

[1] Gordon Earle Moore. Cramming more components onto integrated circuits. *Eletronic Magazine*, 1965.

[2] Benchmark. `http://benchmarksgame.alioth.debian.org`. [Online; acessado 11-Feb-2016].

[3] Marat Fayzullin. Home page of marat fayzullin. `http://fms.komkon.org/`. [Online, acessada 10/04/2016].

[4] Lista de memory mappers do nes. `http://tuxnes.sourceforge.net/nesmapper.txt`. [Online; acessado 16-Mar-2016].

[5] Patrick Diskin. Nintendo entertainment system documentation. 2004.

[6] Paletas. `http://wiki.nesdev.com/w/index.php/Blargg_PPU#Palette`. [Online; acessado 28-Mar-2016].

[7] Scott Ferguson. Tabela de padrões. `https://opcode-defined.quora.com/How-NES-Graphics-Work-Pattern-tables`, 2013. [Online; acessado 10-Mar-2016].

[8] Dustin Long. Tabela de nomes. `http://www.dustmop.io/blog/2015/04/28/nes-graphics-part-1`, 2015. [Online; acessado 10-Mar-2016].

[9] Jarrod Parkes. Tabela de atributos. `http://jarrodparkes.com/2014/01/20/nes-attribute-tables`, 2014. [Online; acessado 10-Mar-2016].

[10] Dustin Long. Sprites. `http://www.dustmop.io/blog/2015/06/08/nes-graphics-part-2`, 2015. [Online; acessado 26-Mar-2016].

[11] Renato Noviello. Segredos por trás do sucesso. `http://nesarchive.net/v4/artigo/segredos-por-tras-do-sucesso`, 2012. [Online; acessado 28-Mar-2016].

[12] Pygame. `http://www.pygame.org/`. [Online, acessada 10/04/2016].

[13] Welcome to python.org. `https://www.python.org/`. [Online, acessada 10/04/2016].