Aulas de Assembly para NES (Motorola 6502)

Autor: Odin – hexagon_odin@hotmail.com

Grupo: Hexagon Traduções – http://hexagon.romhack.net/

Apresentação:

Já é hora da cena da romhack brasileira avançar, por muito tempo trabalhamos em cima de documentos que, quando muito, chegavam apenas a um nível intermediário de conhecimento. Este documento se propõe a avançar ainda mais levando a uma concepção mais aprofundada da engenharia reversa de jogos para o **N**intendo **E**ntertaiment **S**ystem (NES) ou mais conhecido no Brasil, após o lançamento do **S**uper **N**intendo **E**ntertaiment **S**ystem (SNES) como Nintendinho. Obviamente, aqui não irei tratar de como fazer a romhack, descompressão, entre outros para jogos de NES, mas sim, irei dar grandes subsídios para que o mesmo seja feito, comentando aspectos técnicos do console e principalmente como funciona o assembly do processador Motorola 6502 para o NES. Então, vamos lá!

Aula #01 – Aspectos Gerais

Aspectos Gerais do Console NES

5 de Janeiro de 2004

O NES é baseado em um processador 8-bit Motorola 6502 com pequenas modificações, utilizado também em outras plataformas tais como: Apple //, AIM 65, SYM-1, KIM-1 e outros, além disso conta com um controlador de vídeo otimizado chamado de Picture Processing Unit (Unidade de Processamento de Imagens) ou simplesmente PPU. A memória do PPU é separada e pode ser lida e escrita através de registradores de memória especiais que serão vistos mais tarde.

Aspectos Gerais da Linguagem Assembly

A linguagem para a programação do 6502 não é, de fato, complicada, estamos trabalhando com um antigo processador que não inclui comandos místicos e coisas impossíveis de serem decifradas. Claro que você deve ter alguma experiência com programação e bases numéricas, mas, mesmo assim, tentarei ser o mais claro possível.

Na linguagem, temos apenas 3 registradores básicos da CPU, são eles: registrador **A** (Conhecido como Acumulador), **X** (Um registrador para indexação) e **Y** (Um outro registrador, também para indexação).

Funções dos Registradores

A – Acumulador: Todas as operações matemáticas usam este registrador.
X – Registrador para indexação de endereços de memória
Y – Registrador para indexação de endereços de memória

Todos os três registradores podem ser carregados com valores, mas ainda assim conservam suas funções mais específicas. ***Update: 7 de Janeiro de 2004*** Temos ainda 3 registradores adicionais, mas estes não são importantes no momento, pois **a função deles é de marcação do programa**, são eles: **S** (Stack Pointer – Ponteiro para a Pilha, quando a rotina desvia (Jump), ele marca o endereço de retorno), **P** (Processor Status Register – Registrador do Status da CPU – Ele gera flags que dizem o que está ocorrendo com o processador: se houve overflow ou não e etc). **PC** (Program Counter – Contador do Programa: Ele aponta para o próximo endereço a ser executado).

Nosso Primeiro Comando: LD*

Antes de nos aprofundarmos um pouco mais nos aspectos gerais da linguagem, vamos conhecer o nosso primeiro comando, o LD*, onde o asterisco pode ser qualquer um dos três registradores acima citados: A, X ou Y. Assim temos LDA, LDX e LDY. O que isso significa, é o que vamos ver a seguir.

Semântica & Sintaxe

As instruções a seguir, são instruções tidas por instruções de transferência de dados e as abreviações são conhecidas também por Mnemônicos.

LDA – **L**oa**D** **A**cumulator with Memory
Esse comando carrega o acumulador com o valor contido no endereço.
Exemplo: LDA $2003
O valor contido no endereço $2003 foi carregado para o acumulador.
LDX – **L**oa**D** **X** Register with Memory
Carrega o Registrador X com o valor contido no endereço de memória.
Exemplo: LDX #40
O valor 40 foi carregado no registrador X.
LDY – **L**oa**D** **Y** Register with Memory
Carrega o Registrador Y com o valor contido no endereço de memória.
Exemplo: LDY #%00011110
O valor 00011110 foi carregado no registrador Y.

Notações

Como você pode perceber, temos alguns símbolos especiais que modificam o comportamento da instrução: #, $, %. Essas, são notações da linguagem que seguem a seguinte regra:

- Imediato:

Aqui, você não coloca um endereço de memória, mas sim o valor que você deseja que o registrador leve. Os valores só podem ser carregados em 1 byte por vez.

Existem outros modos de endereçamento além deste, como Indireto e Zero-Page, mas creio que este seja o mais importante.

$ - Hexadecimal:

Significa que o endereço ou valor estará em hexadecimal.

% - Binário:

Significa que o endereço ou valor está em binário.

- Decimal:

Quando não houver nada precedendo o endereço ou valor, estes serão tidos por decimais.

Registradores de Indexação X & Y

Uma pergunta que você pode estar se fazendo agora é, o que significa um Registrador ter a função de indexação. Bem, vou explicar isto agora com um código extremamente simples:

```
LDX #$04          ; O valor $04 é carregado no Registrador X
LDA $2000, X      ; O endereço indexado será $2004
```

Dessa forma ficou simples, indexar um endereço é isso, basta você somar o valor de X ao endereço carregado. Você pôde perceber também um ';' este símbolo precede um comentário, tudo que vem depois disto é ignorado por um programa Assembler, usado para compilar códigos assembly, como por exemplo, o nesasm.

Nossa Segunda Instrução: ST*

Essa instrução grava os valores ou valores contidos em endereços de memória. As instruções LD* e ST* são extremamente importantes, uma vez que a maior parte do código as usa com freqüência:

Sintaxe e Semântica:

STA – **ST**ore **A**cumulator in Memory

Grava no endereço de memória o valor carregado no acumulador.

Exemplo: LDA $2003

STA $2005

Pega o valor de $2003 que foi carregado no Acumulador e Grava no endereço $2005.

STX – **ST**ore **X** Register in Memory

Grava no endereço de memória o valor carregado no Registrador X.

Exemplo: LDX #40

STX $2000

Pega o valor decimal 40 carregado em X e grava no endereço $2000

STY – **ST**ore **Y** Register in Memory

Grava no endereço de memória o valor carregado no Registrador Y.

Exemplo: LDY #%01110011

STY $FFFF

Pega o valor binário 01110011 carregado em Y e grava no endereço $FFFF

Como você pode perceber, a instrução ST* nunca pode receber valores imediatos, apenas endereços de memória.

Aqui terminamos a primeira aula de Assembly para NES, espero que você tenha entendido tudo para que possamos continuar o nosso estudo, hoje você pôde ter uma visão geral do console, notações e duas instruções muito importantes, essas instruções são basicamente de leitura e escrita, fazendo uma analogia às linguagens de alto nível. É isso aí pessoal! Até a próxima aula.

Aula #02 – PPU

Configurando e Entendendo o PPU

6 de Janeiro de 2004

Talvez um dos tópicos mais importantes para um romhacker seja exatamente o PPU, uma vez que este trabalha principalmente com formas de compressão e descompressão gráfica e, no NES, isto se torna relativamente simples, uma vez que tudo o que acontece no console em termos de saída gráfica, tem de ter, de alguma forma relação com o PPU, seja com formas brutas de enviar a informação, seja com DMA (Direct Memory Acess – Veremos isto mais tarde).

Para desmistificar mais ainda o assembly, tido por uma linguagem altamente abstrata, veremos que não é bem assim e, pelo menos no NES, tudo funciona em forma de registradores de memória especiais, que são endereços predestinados pelo Mapeamento de Memória do Console a trabalhar com determinadas funções do mesmo.

Neste tópico, antes de trabalharmos com gráficos propriamente ditos: paletas (palletes), quadros (sprites) e fundos de tela (background), aprenderemos a configurar o PPU, dessa forma, deixaremos de lado a jogatina de sintaxe e semântica de comandos e, a cada comando novo, explicaremos.

Vamos ao que interessa, para configurar o PPU, utilizamos dois registradores de memória especiais que são $2000 e o $2001, que vão dizer para o console como o PPU vai funcionar através da escrita bit-a-bit nesses endereços.

Número binário qualquer:	0	1	0	1	1	0	0	1
Número do bit relativo:	7	6	5	4	3	2	1	0

Agora veremos o que cada um desses bits representa para então configurarmos os modos do PPU, se você não entendeu até agora, fique tranqüilo e leia até o final que tenho certeza que você vai entender. Se ainda assim não entender, leia novamente, que você com certeza vai acabar entendendo uma hora ou outra.

Registrador de Controle do PPU 1: $2000

Bit 7 – Executa NMI durante Vblank (Execute NMI on Vblank):

0 – Desabilitado

1 – Habilitado

Bit 6 – PPU Primário ou Escravo: ***Update: 7 de Janeiro de 2004***

0 – Primário

1 – Escravo

Isto aqui não é usado pelo NES.

Bit 5 – Tamanho do Sprite (Sprite Size):

0 – 8x8

1 – 8x16

Bit 4 – Endereço da Tabela de Moldes da Tela (Screen Pattern Table Adress):

0 - $0000 (VRAM)

1 - $1000 (VRAM)

Bit 3 – Endereço da Tabela de Moldes de Sprite (Sprite Pattern Table Address):

0 - $0000 (VRAM)

1 - $1000 (VRAM)

Bit 2 – Escrita Vertical / Incrementos na memória do PPU:

0 – Incrementos por 1

1 – Incrementos por 32

Bit 1-0 – Endereço da Tabela de Referências:

00 - $2000

01 - $2400

10 - $2800 (Espelhado)

11 - $2C00 (Espelhado)

Configurando o primeiro registrador:

```
LDA #%00011001
STA $2000
```

Pronto, configuramos o que queremos através de notação binária para facilitar o entendimento e escrevemos isso bem em cima do registrador de controle do PPU. Da mesma forma, devemos fazer para o registrador de controle do PPU 2: $2001, vamos entendê-lo agora.

Registrador de Controle do PPU 2: $2001

Bit 7-5 – Cores do Fundo da Tela (Background Color):

000 - Nenhum / Preto

001- Vermelho

010 - Verde

100 – Azul

Você pode selecionar apenas uma cor. Outros números danificam o PPU.

Bit 4 – Habilita Sprites (Sprite Display):

0 – Não Mostra Sprites

1 – Mostra Sprites

Bit 3 – Habilita Tela (Screen Display):

0 – Desligado (Tela em branco)

1 – Ligado

Bit 2 – Máscara de Sprite (Sprite Clip):

0 – Não mostra os Sprites nas 8 colunas à esquerda da tela.

1- Mostra Sprites em toda a tela

Bit 1 – Máscara de Imagem (Image Clip):

0 – Não mostra imagem nas 8 colunas à esquerda da tela

1 – Mostra imagens em toda a tela

Bit 0 – Cores (Colour Display):

0 – Monocromático

1 – Colorido

Configurando o Segundo Registrador:

```
LDA #%00011111
STA $2001
```

Fizemos da mesma forma que o primeiro aqui, se você comparar, vai perceber que configurei para a cor de fundo ser preta (Nenhuma), com sprites e tela habilitada e mostrada em todo o lugar com tela colorida. Assim, o meu código final para especificação do funcionamento do PPU ficou:

```
LDA #%00011001    ; Carregando o que selecionei para A
STA $2000         ; Escrevendo para o Registro de Controle do PPU 1
LDA #%00011111    ; Carregando o que selecionei para A
STA $2001         ; Escrevendo para o Registro de Controle do PPU 2
```

Eu tenho certeza que você achou isso bem simples mesmo! Assim, sempre que você, num debug notar algum valor sendo escrito para os endereços $2000 e $2001, você já sabe que, naquele ponto, o PPU está sendo configurado, claro que o funcionamento pode ser modificado a qualquer momento, em alguma situação do jogo.

E aqui terminamos a nossa segunda aula tosca de Assembly 6502, espero que você tenha gostado e, mais do que isso, entendido para que possamos continuar beleza com nossas aulas! Até mais!

Aula #2 Update

Interrupção NMI e Interrupções em Geral

*****Update: 7 de Janeiro de 2004*****

Uma pergunta que ficou no ar foi o que é "Executa NMI durante Vblank", ou seja, coisas que coloquei sobre o primeiro registrador de controle do PPU e não expliquei. Neste pequeno update vamos ter uma pequena noção de interrupções no NES.

NMI é a abreviação de Non-Maskable Interrupt (Interrupção não mascarável), ela é gerada em cada atualização da tela durante um período que chamamos de Vblank (Vertical Blank), obviamente você já pôde notar que a NMI pode ser totalmente desabilitada, pois o jogo pode ser tão tosco que não necessite esperar o ciclo da interrupção para que possa atualizar a tela. Por fim o endereço da NMI é $FFFA

Outras coisas: No NES temos apenas 3 interrupções: RESET, IRQ e NMI. Que seguem a seguinte prioridade: *RESET($FFFC) tem a maior prioridade, NMI($FFFA) e IRQ ($FFFE) tem a menor prioridade.*

As interrupções param o jogo (quase imperceptível – Exceto RESET) para realizar alguma tarefa e por isso devem retornar a rotina principal após isso, usamos, em geral a instrução:

RTI – **R**eturn from **I**nterrupt **R**outine

Esta instrução retorna para o código principal após uma determinada interrupção (Utilizando o endereço marcado por **S** – Stack Pointer, lembra?).

Aula #03 – Paleta de Cores

Funcionamento da Paleta de Cores

7 de Janeiro de 2004

Uma imagem não é nada sem suas cores, não é verdade? Por isso, antes de falar sobre carregamento de Quadros (Sprites) e Fundos de Tela (Background) irei falar das paletas de cores no NES. Como já foi dito, tudo no NES funciona através de Registradores de Memória Especiais e, no caso da paleta de cores, temos dois registradores: $2006 e $2007.

Uma paleta de NES é extremamente simples e só pode armazenar até 16 cores, uma vez que os endereços predestinados a recebê-la, segundo o mapeamento de memória do PPU são: $3F00 – que recebe as paletas de cores das imagens e #3F10 - que recebe a paleta de cores dos sprites. Como você pode notar a diferença entre eles é $10 (hexdecimal) que equivale a 16 (decimal), ou seja, 16 cores (cada cor leva 1 byte). Assim podemos ter duas paletas com fins distintos totalizando até 32 cores.

Funções dos registradores:

$2006: Recebe o endereço onde a paleta de cores deve ser armazenada. Escreve-se duas vezes neste endereço, pois você deve lembrar que nós só podemos armazenar um byte por vez no acumulador.

Exemplo:

```
LDA #$3F    ; Armazena o valor $3F no acumulador
STA $2006   ; Escreve o valor $3F no endereço $2006
LDA #$00    ; Armazena o valor $00 no acumulador
STA $2006   ; Escreve o valor $00 no endereço $2006 e assim $2006 marca o
            ;endereço $3F00 como o endereço onde os dados da paleta serão
            ;armazenados.
```

$2007: Ora, o que está faltando? Armazenar os dados da paleta! $2007 fará exatamente isto: Armazenará os dados da Paleta de cores no endereço que foi identificado por $2006. Vamos ao código:

```
        LDY #$00          ; Zera o indexador
$****: LDA $????, Y       ; Pega a cor
        STA $2007         ; Armazena a cor em $2007
        INY               ; Incrementa o indexador Y em 1
        CPY #16           ; A cada incremento Y é comparado a 16
        BNE $****         ; Se não for igual a 16 retorna para o endereço acima
```

Observação: $****: Endereço do início da rotina

$????: Endereço onde a paleta foi posta pelo programador, ela está, no mapeamento de memória da CPU no banco da rom do cartucho, onde o código do programa fica armazenado.

Análise detalhada da rotina:

Assim podemos analisar a rotina facilmente. Após ter dito para $2006 que o endereço de armazenamento da paleta seria $3F00, a rotina acima começou a carregar os dados da paleta neste endereço através do registrador de memória especial $2007. as duas primeiras linhas LDA $????, Y e STA $2007, Pegam a cor e gravam no registrador. Como você pode ver o endereço está indexado e toda vez no loop Y vai sendo incrementado através da seguinte instrução:

INY – **IN**crement **Y** register by one:

Esta instrução adiciona 1 ao valor carregado em Y. Da mesma forma temos a instrução INX – **IN**crement **X** register by one.

Dessa forma como o endereço está sendo indexado, a cada loop o endereço vai aumentando em 1: $???? + 1, 16 vezes. Porque só até 16 vezes? Porque Y está sendo comparado com #16 através da instrução:

CPY – **C**om**P**are memory and **Y** register:

Compara o valor de Y com o valor de um determinado endereço de memória ou um valor imediato. Da mesma forma temos a instrução CPX – **C**om**P**are Memory and **X** register.

Assim, se o valor de Y não for igual a 16 (quando chega a $3F10), ele executa a rotina a partir do endereço $**** (um endereço qualquer), através da instrução:

BNE – **B**ranch if **N**ot **E**qual

Se a comparação anterior não for igual (CPY #16), essa instrução envia o **C** (Program Counter, Lembra?) para um outro endereço. Da mesma forma temos a instrução BEQ – **B**ranch if **EQ**ual.

Assim, entende-se que essa rotina diz basicamente o seguinte: Enquanto todas as cores da paleta não forem carregadas (16) continue executando o carregamento de uma por uma a partir do endereço $3F00.

Bem galerinha do #romhacking (BRASNET a.k.a. BRASMERDA), por hoje é só... Odin está cansado e vai dormir um pouco. Até nossa próxima aula milombeira de ASM 6502!

Aula #04 – Sprites (Quadros)

Trabalhando com Quadros

8 de Janeiro de 2004

Heh, nem acredito que a minha preguiça nos permitiu chegarmos à nossa 4ª aula sobre Assembly 6502 para NES! Mas aqui estamos e vamos tratar hoje de algo extremamente importante para o universo romhacker: os sprites! Que são quadros de animação que deixam o jogo mais emocionante... aquele chute, poder especial, aqueles inimigos chatos que vem do nada e ficam se repetindo sempre que a gente volta e até aqueles textos que contém alguma animação nas fontes ou não, são Sprites.

Bem, em termos de sprites o NES é um tanto místico, por isso, peço atenção especial a esta aula e principalmente na aula sobre background (Fundos de Tela). O NES pode armazenar até 64 Sprites (independente se tenham 8x8 tiles ou 8x16 tiles) que serão armazenados, segundo o mapeamento de memória do PPU, nas tabelas de moldes (Temos duas, nos endereços: $0000 e $1000 – se você quiser chamar estes endereços de VRAM/VROM, fique à vontade).

Mas, um sprite não é nada sem os seus **atributos**: Em que posição ele vai aparecer? Qual tile irei carregar? Quais cores ele terá? Bom, para isso, o NES tem 256 bytes de memória reservados no que chamamos de SPR-RAM (você já percebeu que se chama **SPR**ite-**RAM**), este espaço especial, **não** está mapeado nem na CPU e nem no PPU, é exclusivo para receber os atributos dos Sprites. Ora, vamos pensar um pouco... Se nós temos 256 bytes para armazenamos informações para até 64 Sprites, quanto cada informação sobre cada Sprite terá? 256 / 64 = 4bytes! Exato, cada um dos 64 Sprites tem direito a 4 bytes para armazenar seus atributos na SPR-RAM. O que significa cada um desses bytes, é o que veremos a seguir:

Byte 1 – Posição vertical do Sprite na tela

Byte 2 – Número do Tile (Tile Index #)

Byte 3 – Armazena informações bit-a-bit:

- Bit 7 – Virado Verticalmente (0 = Normal ; 1 = Virado)
- Bit 6 – Virado Horizontalmente (0 = Normal; 1 = Virado)
- Bit 5 – Prioridade do Fundo de Tela (0 = Na frente; 1 = Atrás)
- Bit 4, 3 e 2 – Desconhecido – Escreva – 000
- Bit 1 e 0 – os 2 bits superiores da cor (?) (bit7 – bit7)

Byte 4- Posição Horizontal do Sprite na Tela

Tendo essas informações disponíveis, temos que aprender agora a armazenar estes sprites no game e, mais uma vez repito: Tudo no NES é feito a partir de registradores especiais e para os sprites temos dois: $2003 e $2004.

Funções dos registradores:

$2003: Creio que você já deva imaginar para que serve este registrador... exato! Para receber o endereço onde os sprites serão armazenados na tabela de moldes do PPU. Escrevemos da mesma forma que as paletas:

```
LDA #$00      ; Carrega o valor $00 para o Acumulador
STA $2003     ; Guarda no registrador especial
LDA #$00      ; Carrega o valor $00 para o acumulador
STA $2003     ; Vamos começar com $0000, endereço da tabela de moldes 1
```

$2004: Aqui é o registrador especial que vai gravar os **atributos** dos Sprites na SPR-RAM, no caso, sprite por sprite... super fácil:

```
LDA #50           ; Posição vertical do Tile (coluna 50)
STA $2004         ; Guarda o atributo que será enviado para SPR-RAM
LDA #00           ; Número do Tile: Vamos começar com o primeiro
STA $2004         ; Enviando a informação para SPR-RAM
LDA #%00000000    ; Escrito em binário para facilitar – Tudo desabilitado
STA $2004         ; Guardando na SPR-RAM
LDA #20           ; Posição horizontal do tile (linha 20)
STA $2004         ; Enviando informação para SPR-RAM
```

O registrador $2004 é incrementado a cada carregamento, mas mesmo assim, carregar coisas para 64 sprites é inviável, ou seja, jogos comerciais, nunca usarão este modelo de carregamento de sprites, pois iriam ficar extremamente lentos para 1.79 MHz do nosso querido Nintendinho. Jogos comerciais, que são os que interessam para nós, usam transferência por DMA (Direct Memory Acess – Acesso Direto à Memória), que para sprites são acionadas através do registrador especial $4014, mas, DMA é assunto para outra aula...

Aqui, meus Morloks, terminamos mais uma aula sobre o básico de sprites. Assim, para o romhacker, o que interessa é como estes sprites estão sendo enviados para

a tabela de moldes (VRAM), pois, se estiverem comprimidos, o programa terá de executar uma rotina de descompressão antes de poder enviá-los para a tabela de moldes (VRAM) e aí que entramos, desmistificando através do assembly esta rotina.

Bem, agora vou descansar um pouco... see you in next class!

Aula #05 – Backgrounds (Fundos de Tela)

Trabalhando com Fundos de Tela

9 de Janeiro de 2004

E aí? Entenderam bem como trabalhar de forma bruta com sprites? Espero que sim! Hoje vamos entender os Fundos de Tela, que, diferentemente dos Sprites, eles são figuras estáticas que servem de cenários para as tramas dos jogos, neles os Sprites vivem suas aventuras e interagem com o Fundo da Tela ou com outros Sprites.

Depois de algumas aulas, já sabemos basicamente como o processo funciona, para fundos de tela, temos 2 registradores de memória especiais que são $2006 e $2007, como você pode notar, são os mesmos registradores que usamos para carregar as paletas, isso porque este dois registradores, não são específicos para paletas ou para fundos de tela, mas são registradores da VRAM. Como de práxis, o endereço $2006 é usado para configurar o endereço do PPU que será acessado por $2007, da mesma forma que os outros:

```
LDA $20
STA $2006
LDA $00
STA $2006
```

Se você notar o mapeamento da memória do PPU (VRAM), vai notar que vamos cair no endereço Table Name 1 (Tabela de Referências 1), você pode ler isto como localização dos mapas/backgrounds e temos 4 endereços para este fim (Veja o mapeamento de memória). No exemplo anterior, ajustamos o endereço para o início da Tabela de Referência 1.

Bem, para sabermos como o fundo da tela é carregado temos que entender o que é um mapa. Isso é um tanto complicado de explicar, por isso preste bem atenção!

Num editor de tiles, geralmente temos 16 tiles por linha:

$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

Assim sucessivamente...

Sabendo que **o NES tem 32 x 32 tiles** na tela, um mapa que queira fazer uma figura com o que está desenhado em $00, $01, $02, $10, $11 e $12, como por exemplo:

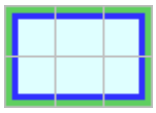

Figura 1: A imagem no editor de tiles

o mapa faria o seguinte:

00 01 01 01 01 01 02 - 03 03 ... até escrevermos 32 vezes

10 11 11 11 11 11 12 - 03 03 ... até escrevermos 32 vezes

Observação: o tile 3 contém nada (pega a cor do fundo que foi configurada pelo PPU)

Assim, eis a figura que seria mostrada no jogo:

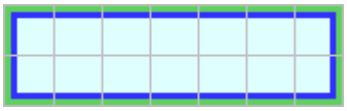

Figura 2: A imagem mostrada no jogo segundo o mapa que a definiu

Entendeu o espírito da coisa?

Bem, para as tabelas de referências do PPU, nos trazemos exatamente isto, os mapas! Então vamos enviá-los para o PPU (VRAM):

```
                ; Depois de termos trazido configurado o endereço via $2006
        LDX $00
$****: LDA $????, X; Carrega o tile do mapa
        INX         ; Incrementa X
        STA $2007   ; Guarda o tile do mapa no endereço $2000
        CPX #??     ; Número de tiles a serem puxados do mapa
                    ; Se estivéssemos trabalhando com o exemplo anterior, este valor
                    ; seria #64 pois precisamos apenas de 2 linhas
        BNE $****   ; Se não tiver carregado todos os tiles, continue a rotina
```

Observação: $****: Endereço do código do programa

$????: Endereço onde o mapa foi posto no programa

Fácil, não? Mas ainda não acabamos nossa aula. Voltemos à aula de configuração do PPU (Aula #2). No **Registrador de Controle 1, no Bit 2** existe a tal de **escrita vertical**, onde você pode configurar se quer incrementos por 1 ou incrementos

por 32. É exatamente isto, se escolher incrementos por 1, o mapa vai ser desenhado horizontalmente:

****

----

*Observação: * > lugar onde está sendo escrito*

- > lugar aonde ainda vai ser escrito

Se escolher incrementos por 32, o mapa vai ser desenhado verticalmente, já que cabem 32 tiles por linha, assim, quando o incremento atinge 32, já vai direto para a próxima linha.

* -------

* -------

* -------

*Observação: * > lugar onde está sendo escrito*

- > lugar aonde ainda vai ser escrito

Agora sim, terminamos mais uma aula: Aleluia! Esta deu uma preguiça de fazer... putz! Nem vou comentar o que um romhacker deve fazer, pois você já deve estar tendo as suas próprias idéias... Eu realmente espero que sim!

Ainda bem que hoje é sexta e dia de sábado e domingo professor não dá aula, vocês sabem, não é? Dia de sábado a escola suga os professores chamando-os para fazer o planejamento semanal e... dia de domingo é dia de descanso! Aproveitem para assistir a uma missa (ou culto), confessar os pecados (rezar) e purificar suas almas pecaminosas e assim se prepararem espiritualmente para continuarmos com nossas aulas de ASM NES. E isso é realmente necessário! Pois vocês sabem que ASM é coisa do demo! Até a próxima aula no dia: 12 de Janeiro de 2004. Fui....

Aula #06 – Mapeamento de Memória

Analisando o Mapeamento de Memória do NES

12 de Janeiro de 2004

Antes de entramos nos tópicos que faltam trabalhar sobre gráficos, que são: Sprites e DMA e deslizamento de fundos de tela, vamos antes entender algo que eu já devia ter explicado a bastante tempo: o mapeamento de memória do console. Já o citei várias vezes, mas agora já está mais que em tempo de o analisarmos.

Para falarmos de mapeamento de memória do 6502 para NES, devemos levar em conta duas partes: O mapeamento de memória da CPU (Central Processing Unit) e o mapeamento de memória do PPU (Picture Processing Unit), uma vez que eu deixei bem claro (Aula #1) que a memória destinada a imagens é separada da RAM principal do console.

Mapeamento de Memória da CPU

O NES tem 64Kb ($10000 = 65536 e 65536/1024 = 64Kb) de memória principal que são divididos da seguinte forma:

Descrição	*Endereço*
6. Banco Superior da ROM	$C000 - $10000
5. Banco Inferior da ROM	$8000
4. RAM do Cartucho	$6000
3. Módulo de Expansão	$5000
2. Entrada e Saída	$2000
1. RAM Interna	$0000

1. *RAM Interna*: 2Kb de RAM Interna, que é espelhada 3 vezes totalizando 8Kb de RAM Interna. Aqui são colocadas coisas temporárias.
2. *Entrada e Saída*: 3Kb para alocar os registradores que referem-se a entrada e saída do console. Como você pode notar todos os registradores de memória especiais como: controle do PPU, Sprites, Rolagem de cenário, VRAM , Som (veremos mais tarde), DMA (veremos mais tarde) e Joysticks (veremos mais tarde), tudo isso é controlado por este espaço de endereço.

3. *Módulos de Expansão(1Kb)*: Relacionado particularmente a o Famicom Disk System, que foi a única expansão lançada para o NES e talvez você nem conheça, mas ela tem seu próprio mapeamento de memória.
4. *RAM do Cartucho(2Kb)*: Basicamente os Save Games, claro que este espaço pode estar armazenado em uma bateria interna no cartucho.
5. *Banco Inferior da ROM(4Kb)*: Conhecido também por PRG-ROM (**PR**o**G**ram-**ROM**), é onde o código-fonte do jogo fica armazenado.
6. *Banco Superior da ROM(4Kb)*: PRG-ROM, mesma função do Banco Inferior.

Mapeamento de Memória do PPU

Contamos com 16Kb para armazenamento de coisas relacionadas a imagem, note que ainda temos mais 256bytes que pertencem à SPR-RAM (explicada na Aula #4).

Descrição	*Endereço*
14. Espaço vazio	$3F20- $4000
13. Paleta de Cores de Sprites	$3F10
12. Paleta de Cores de Imagem	$3F00
11. Espaço vazio	$3000
10. Tabela de Atributos #4	$2FC0
9. Tabela de Referências #4	$2C00
8. Tabela de Atributos #3	$2BC0
7. Tabela de Referências #3	$2800
6. Tabela de Atributos #2	$27C0
5. Tabela de Referências #2	$2400
4. Tabela de Atributos#1	$23C0
3. Tabela de Referências #1	$2000
2. Tabela de Moldes #2	$1000
1. Tabela de Moldes #1	$0000

1 e 2) Tabelas de Moldes: Onde os tiles básicos de fundos de tela e quadros (backgrounds e sprites) são carregados.Veja que 8Kb dos 16Kb disponíveis foram destinados a isso!.

3, 5, 7 e 9) Tabelas de Referências) Onde se armazena os mapas de tiles na tabela de referências #1, as outras tabelas são espelhadas e são usadas quando o cartucho usa mappers (veremos mais tarde).

4, 6, 8, 10) Tabelas de Atributos) Será explicada de forma aprofundada em outra oportunidade, mas, basicamente controla as cores dos tiles de forma bem detalhada, baseando-se nas paletas de cores. É controlada pela tabela de atributos #1, o resto é espelhamento usado quando o cartucho usa um mapper (veremos mais tarde)

11 e 14) Espaço Vazio) A própria denominação explica a função.

12 e 13) Paletas de Cores) Usado para armazenar paletas de cores dos tiles e sprites.

Assim concluímos mais uma aula, que não acrescenta muito, mas não deixa de ser importante, pois um romhacker precisa pelo menos saber em que local da memória do console ele deve procurar no debug as rotinas do jogo, não é? Então, falou macacada, espero que tenham gostado e, principalmente, entendido mais uma aula. Até mais.

Aula #07 – DMA

Trabalhando com os Sprites utilizando DMA

13 de Janeiro de 2004

Bem pessoal, já estamos bem avançados em nossas aulas. Se você vem entendendo tudo até aqui, parabéns! Você já pode se considerar um iniciado em assembly de NES. Na aula de hoje trataremos de DMA e sprites, simplesmente porque o NES usa DMA apenas para esta tarefa: Transferir atributos dos sprites de um endereço de memória da CPU para a SPR-RAM.

Antes de irmos além precisamos entender o que é DMA. DMA é o acrônimo para **D**irect **M**emory **A**ccess – Acesso Direto à Memória e tem por função fazer transferências Diretas entre endereços de memória. Na nossa aula sobre Sprites, nós transferimos os atributos dos Sprites em tempo de execução do programa, através do registrador especial $2004, como já foi dito, isto é inviável num programa (jogo) comercial, é como se você tivesse um saco (DMA) com espaço para colocar 256 bytes para serem levados para a SPR-RAM, mas você teimasse em levar 4 bytes por vez! O DMA vai fazer exatamente isto, em vez de você levar atributos de sprites um a um, você vai levar todos os 256 bytes (lembra?) de uma vez para a SPR-RAM.

Como funciona? Simples, através do registrador especial $4014, vamos acessar um endereço da RAM interna (Entre $0000 e $0700, que já tem armazenado previamente todos os atributos). O acesso se dá da seguinte forma: Se os atributos de Sprites estão armazenados na RAM interna no endereço $0400, então deverei escrever 4 em $4014, dessa forma o endereço $0400 será acessado por DMA.

Função do Registrador $4014: Faz com que 256 bytes do Endereço de Memória da CPU, especificado por N x 100 (Onde N é número que será escrito no registrador), sejam transferidos da RAM interna para a SPR-RAM.

Exemplo:

```
; Os atributos dos sprites estão previamente alocados em $0200
LDA #2          ; Endereço a ser acessado é $0200
STA $4014       ; Transfere 256 bytes de uma vez de $0200 para a SPR-RAM
```

Fácil não? Sempre que você notar uma sintaxe assim, você já sabe que os atributos dos sprites estão sendo transferidos. Estou com pressa hoje! Até mais ver!!!

Aula #08 – Rolagem de Cenários

Rolando os Backgrounds

14 de Janeiro de 2004

O herói vai progredindo no jogo e a tela o acompanha... não, a tela não o acompanha! Ela desliza independentemente se há ou não sprites na tela. Esse processo se chama rolagem de cenários. No NES, o cenário pode deslizar de duas formas apenas: horizontalmente e verticalmente, esse procedimento se dá através da escrita em um registrador exclusivo para a rolagem do cenário: o registrador $2005.

Eis que entramos em mais um pouco de teoria. Você já deve ter visto aqueles desenhos animados, como por exemplo o Corrida Maluca, onde o tempo todo o cenário deve estar em movimento e, também já deve ter notado que, os cenários ficam se repetindo o tempo inteiro, uma árvore que estava aqui no cenário, em pouco tempo aparece de novo e de novo e de novo... por que? Para poupar trabalho do desenhista... ele faz um só cenário e o repete por várias telas. No jogo, não é diferente: o programador cria apenas alguns fundos de tela e os faz deslizar por várias **Tabelas de Referências** (Você deve estar lembrado).

Nós temos apenas duas tabelas de referências: #1 e # 2, se o jogo precisar de mais, ou precisar rolar o cenário verticalmente, ele pode usar a #3 e #4, mas para um jogo simples, essas duas bastam. As tabelas de referências estão colocadas da seguinte forma:

Tabela de Referência #3	Tabela de Referência #4
Tabela de Referência #1	Tabela de Referência #2

Na escrita horizontal, nós guardamos um valor entre 0 e 256, na escrita vertical, um valor entre 0 e 239. Qualquer valor acima disto torna-se negativo.

Exemplo:

```
; Claro que a rolagem do cenário vai depender dos joysticks (outra aula!)
; Mas basicamente é da seguinte forma:
; PPU com escrita vertical desabilitada (horizontal habilitada)
LDA #$??    ; Deve haver um código relacionado ao joystick para incrementar ou
STA $2005   ; decrementar o valor #$??. Se chegar a passar, vai para a Tabela de
            ; Referências #2, onde estará aguardando outra cópia do fundo de tela.
```

Fim da aula! Podem ir para o recreio! Aqui, terminamos o básico sobre gráficos, coisa que é essencial para um romhacker! Daqui a diante, vamos aprender mais sobre outros aspectos do NES, como joysticks e som. Não irei trabalhar nestas aulas o Famicom Disk System, mas incluirei partes especiais destinadas à complementação da sintaxe de assembly, que ainda não mostramos quase nada, apenas algumas instruções e ao debug de rotinas utilizando o FCEUD e NESten. Falou, macaquinhos!

Aula #09 – Joysticks

Como funcionam os joysticks

15 de Janeiro de 2004

Sem uma forma de controle dos usuários do jogo, ele não seria nada, apenas figuras imóveis, mas enfim, existem os joysticks que são a forma mais comum de interação entre o usuário e o jogo, aqui, vamos trabalhar como os joysticks funcionam no NES.

Existem dois registradores de memória especiais para os joysticks, que são: $4016 para o joystick #1 e $4017 para o joystick #2, veremos agora, como eles são trabalhados em uma rotina assembly.

A primeira coisa a fazer, é dar um reset nos joysticks, uma espécie de "curto circuito", que faz com que os joysticks comecem a funcionar. Isso se faz escrevendo #$0100 para os registradores.

```
LDA #$01      ; 1 byte por vez! Lembrando...
STA $4016     ; Joystick #1, se fosse o dois seria $4017, óbvio!
LDA #$00
STA #$00      ; O joystick #1 está preparado para funcionar!
```

Após a inicialização dos joysticks, se faz leituras progressivas no registrador do mesmo a fim de **ler** o estado dos botões. Se o botão estiver pressionado, o bit #0 do registrador será 1, ou seja, #$00000000 – se o botão **não** estiver pressionado e #$00000001 – se o botão estiver pressionado. A ordem de leitura é a seguinte:

1. Botão A
2. Botão B
3. Botão SELECT
4. Botão START
5. Botão CIMA
6. Botão BAIXO
7. Botão ESQUERDA
8. Botão DIREITA

Vamos a um código de demonstração do controle #1:

```
LDA #$01
STA $4016
LDA #$00
STA $4016          ; Inicializando o joystick #1

LDA $4016          ; Primeira Leitura: Lê estado do botão A
AND #1             ; Verifica se o bit #0 está setado
BNE <código_de_A>; Se sim, vai para uma rotina que faz algo se A for pressionado.
LDA $4016          ; Segunda Leitura: Lê estado do botão B
AND #1             ; Verifica se o bit #0 está setado
BNE <código_de_B>; Se sim, vai para uma rotina que faz algo se B for pressionado.
```

E assim sucessivamente checa se SELECT, START, CIMA, BAIXO, ESQUERDA OU DIREITA, nesta ordem, foram pressionados e faz algo se forem pressionados.

Instruções utilizadas:

AND – **AND** with Acumulator

Esta instruções é um 'E' lógico (&&), ela compara o seu valor com o valor do Acumulador bit a bit e retorna um resultado lógico se iguais, baseado na tabela a seguir:

Instrução AND		
A	B	R = A *AND* B
1	1	1
1	0	0
0	1	0
0	0	0

Como você pode notar, só será verdadeiro, se os bits 1 e 1 forem setados.

Exemplo:

A = 1 0 0 0 0 0 0 1

B = 0 0 0 0 1 0 0 1

R = 0 0 0 0 0 0 0 1

Mais informações sobre BNE:

Eu já havia falado anteriormente que BNE significa: **Branch if Not Equals**, mas em uma comparação lógica, você **não** deve levar isto em conta, em uma comparação lógica, entenda BNE da seguinte forma: **Branch on result not zero**. Assim sendo, vamos ver porque nós usamos BNE ao invés de BEQ, se na verdade nós queríamos que os dois fossem iguais (BEQ - $00000001 e $00000001), para que pudéssemos partir para a rotina.

Estado do joystick (pressionado): 0 0 0 0 0 0 0 1 = 1
AND **:** 0 0 0 0 0 0 0 1 = 1

BNE **:** 0 0 0 0 0 0 0 1 = 1
Vai para a rotina, por que o resultado foi diferente de zero (Branch on result not zero)

Estado do joystick (não pressionado): 0 0 0 0 0 0 0 0 = 0
AND **:** 0 0 0 0 0 0 0 1 = 1

BNE **:** 0 0 0 0 0 0 0 0 = 0
Não vai para a rotina porque o resultado foi zero (Branch on result not zero)

Da mesma forma, em uma comparação lógica, você deve entender **BEQ** como **Branch on result zero**. Eu sei que essa última parte foi um tanto complicada para quem está começando, mas se você ler com atenção e reler se não entender, tenho certeza que vai acabar entendendo. De qualquer forma, minha consciência está tranqüila, afinal, eu avisei que as nossas aulas iriam requerer conhecimento de bases numéricas e programação! Essa aula tem um fim aqui! Até mais....

Aula #10 – Som

Como funciona BASICAMENTE o som

16 de Janeiro de 2004

Não vou aqui me aprofundar no som e em suas peculiaridades, vou falar o básico, mesmo porque você, neste ponto das aulas já deve estar bastante familiarizado com o funcionamento do NES e seu 6502 e, assim como as outras coisas no NES, o som funciona através de registradores especiais.

Os valores relativos ao som, são postos em blocos de registradores especiais: $4000 ao $4003, que são chamados "Square Wave 1". Temos também o "Square Wave 2" que funciona da mesma forma do primeiro, mas pode alcançar freqüências diferentes e está nos registradores: $4004 ao $4007. Nos Registradores $4008 - $400B, temos os registradores para o "Triangle Wave" e no $400C - $400F, temos o "Canal Noise", ou seja, temos 4 canais de som no NES.

Cada um desses registradores recebe valores bit a bit, que controlam variadas funções relativas ao som, como o volume, por exemplo. Não vou aqui entrar nesses bits, se você estiver interessado em saber o que cada bit faz, recomendo que leia algum documento relacionado ao som do NES.

Após a escrita dos valores do som, nós escrevemos esses valores em um registrador especial, que faz o som tocar, este registrador é o $4015, também conhecido por permutador de som, porque além de habilitar o som, ele é que faz as trocas entre canais, ou faz sons com que os canais interajam. Ele é controlado bit a bit também, obviamente que cada bit habilita um um canal de som(leia documentação sobre o som)
Exemplo:

```
lda #% 11111111  ; Este registrador seta alguns modos no "Canal
sta $400C        ; Noise". Só de mal habilitei tudo
lda #$50         ; Este seria o "Play Rate" do "Canal Noise". Aqui eu configurei o
sta $400E        ; o pior play rate possível.
lda #$AB         ; Aqui é a nota propriamente dita. Coloquei uma aleatoriamente.
sta $400F
lda #%00001000   ; Aqui o bit setado habilita o "Canal Noise"
sta $4015        ; Escrever neste registrador faz o canal setado tocar.
```

Assim é mais ou menos a configuração de som para todos os canais possíveis. Sei que não está detalhado, mas realmente não achei necessário, mesmo porque, se explanasse tudo, a aula se estenderia por várias páginas e ficaria um tanto cansativa.... Mas se você quiser MESMO saber o que cada bit de cada registrador faz, consulte algum documento sobre o som no NES. Até a próxima aula.