Assembly para Super Mario World

Um tutorial feito para SMW Hackers

Ersanio
TRADUZIDO POR Insanit

Prefácio

Este é um tutorial de ASM escrito por mim, Ersanio. Eu escrevi esse tutorial a fim de ensinar a outras pessoas o Assembly 65c816.

No entanto, desta vez há uma reviravolta. Este tutorial é feito sob medida para SMW Hackers, a fim de tornar os exemplos mais “tangíveis”. Eu percebi que meu tutorial anterior é bastante.... a leitura, então este tutorial terá os capítulos curtos e agradáveis.

Meu tutorial anterior “Assembly para o SNES” focado principalmente em opcodes e o que acontece por trás das cenas. Este tutorial fará exemplos os quais tem efeitos claros no jogo Super Mario World. Pense em coisas do tipo: Alterar o Power-up do jogador ou o contador de vidas.

Este tutorial não irá ensinar como codificar os bosses mais absurdos ou um patch que torna SMW em um Rail Shooter, mas vai ensinar o mais básico do ASM para você começar sua aventura pelo assembly.

Antes de começar a ler este tutorial, eu recomendo fortemente que você baixe meu tutorial anterior e mantenha os poucos capítulos sobre Hexadecimal, Binário, Endereços & Valores e finalmente, ROM e RAM prontas para leitura. Esses são os conceitos básicos que você precisa entender, caso contrário este tutorial de ASM não fará tanto sentido.

Se você por qualquer razão quiser entrar em contato comigo, poderá me encontrar em:

https://www.smwcentral.net/?p=profile&id=3

http://ersan.io/ meu próprio website

https://twitter.com/Ersanio meu Twitter

Discord: Ersanio#9746

Agradecimentos especiais para certas pessoas que direta (e indiretamente) me ensinaram Assembly:

Bio, Killozapit, MiOr, schwa, Smallhacker, smkdan, Sukasa, Roy (Fuzzyfreak), p4plus2, e muitos outros.

Histórico da Versão

Versão 1.1

- Capítulo 6: Abordagem breve sobre SEC
- Capítulo 6: Menção muito breve sobre Indexação e o stack
- Capítulo 9: Consertado um botão de referência nomeado de forma errada
- Capítulo 9: Informação adicional sobre identificação de loops infinitos
- Capítulo 10: Abordagem do registro B mais clara
- Capítulo 10: Branch zero-flag mais clara

Versão 1.0

- Versão inicial

Tabela de conteúdos

Capítulo 1: Introdução ao Assembly 65c816

Você deve ter ouvido sobre ASM, ou 65c816, ou assembly. Então o que é ASM? ASM significa **A**s**S**e**M**bly. Dividindo em diferentes partes o acrônimo 65c816, o 816 significa que o processador pode ser tanto 8-bit ou 16-bit. O “c” significa CMOS, 65 significa que o processador é da família de CPU 65xx. O processador era supostamente o mais revolucionário para a época. Neste tutorial eu vou explicar mnemônicas / instruções (eu as chamo de opcodes), e como usar elas corretamente.

Com o ASM 65c816 você pode codificar conteúdo para jogos de SNES (como recursos personalizados para Super Mario World). ASM é uma linguagem de programação da 2ª geração (a qual é de baixo-nível comparado ao C# por exemplo). São códigos de máquina legíveis, eventualmente traduzidos para códigos de máquina Hexadecimal. Todos os opcodes consistem de 3 letras, juntamente com diversos parâmetros.

Capítulo 2: Como funciona o SMW

Super Mario World roda graças a ROM (Read-Only Memory) dentro de um cartucho sendo lido pelo SNES. O SNES tem uma área fixa de memória designada para a RAM do jogo (Random Access Memory). Os programadores dão à RAM **significado** usando ASM dentro da ROM.

Desta forma, alterando os valores dentro dos endereços de RAM resulta na manipulação do jogo com resultados visíveis, como por exemplo fazer o jogador crescer ou encolher, ou dar moedas a ele. Alguns endereços de RAM manipulam outros endereços de RAM quando um valor é escrito para eles.

Para exemplificar o parágrafo anterior, eis o que acontece se o jogador tocar em uma moeda:

1. O efeito sonoro de moeda toca
2. A moeda desaparece
3. Partículas de brilho aparecem
4. O jogo diz a si mesmo para adicionar 1 moeda ao contador

Para reproduzir um efeito sonoro, o jogo escreve no endereço de RAM de efeitos sonoros $7E1DFC. Este endereço é diretamente linkado ao SNES' SPC-700 graças a programação do jogo, dando ao endereço essa proposta.

Número #2 e #3 são ligeiramente complicados. Não é necessário entendê-los (ainda). O que eles fazem é chamar uma **rotina** para despawnar a moeda e spawnar partículas de brilho.

#4 é ainda mais complicado. Para acrescentar 1 ao contador de moedas, o jogo escreve 01 em $7E13CC. Você poderia pensar que o jogo apenas **acrescentaria** 1 ao contador de moedas ao invés de **alterar/definir** para 1, mas isso é o que o ponto #4 significa: O jogo dizendo a si mesmo para adicionar 1 moeda ao contador, porque onde quer que você escreva em $7E13CC, o jogo lê por trás das câmeras e aumenta as moedas atuais do jogador, $7E0BDF, por aquela quantidade.

Isto deve soar confuso, mas considere o seguinte: Existem diversas fontes de moedas em Super Mario World. Os blocos de moeda, Yoshi comendo bagas e inimigos, não tocar na barra da Goal Tape lhe dá uma moeda, POW cinza transforma inimigos em moedas, e assim por diante. Também considere que o jogo dá ao jogador uma vida sempre que o contador tem 100 moedas e então o contador reseta.

Se o jogo tivesse que checar pelas 100 moedas toda vez que o jogador obtiver uma moeda, isso causaria diversos códigos duplicados. Portanto, os desenvolvedores programaram um endereço que diz ao jogo "hey, aumente as moedas do jogador em X e se for 100, dê uma vida e resete o contador de volta para 0". Foi assim que a RAM $7E13CC teve seu propósito: Tornar-se um manipulador geral de moedas.

Em uma nota semi-relacionada, existe um sprite inutilizado que dá 5 moedas ao jogador de uma vez: Sprite $7E, a moeda vermelha voadora. Provavelmente outra razão pela qual o $7E13CC existe.

Se os programadores quisessem, eles poderiam ter usado outros endereços para essa proposta, como o $7E1358, $7E1734, e assim por diante. Poderia ser **qualquer um**, mas o jogo foi completado e publicado, então não existe alteração neles. Cada um dos endereços tem uma proposta (até mesmo “inutilizado” é uma proposta. Portanto cada endereço pode ter seus efeitos documentados. E é aqui que eu introduzo...

Capítulo 3: Seu novo melhor amigo: RAM map

Certo. Considerando que os endereços RAM do Super Mario World têm uma proposta definida, é possível compilar uma gigante lista de endereços com seus efeitos no jogo. A *Super Mario World Central* possui um RAM map completo. Assim é como um endereço RAM documentado parece:

$7E0019	1 byte	Player	Player powerup/status. #$00 = small; #$01 = big; #$02 = cape; #$03 = fire.

Um endereço RAM consiste de pelo menos 3 partes: O endereço em si (primeira coluna), o tamanho (segunda coluna) e finalmente a descrição (quarta coluna). A terceira coluna é o tipo de endereço. Tipo uma tag/rótulo. Nada obrigatório.

Este é um perfeito exemplo de um endereço RAM bem documentado. Ele diz em poucas palavras o que ele é, e também dá uma lista de valores válidos que o endereço pode conter.

Talvez é notável que o valor termina em #$03. Se um endereço RAM tem uma lista de valores, mas abruptamente termina desse jeito, normalmente significa que os outros valores (04,05, ... FE, FF) são inválidos. Eles não fazem absolutamente nada, ou causam glitch/crash no jogo.

Você pode fazer algumas coisas com os endereços RAM. Você pode até ler seus valores. Ou sobrescrever eles com novos valores.

No caso deste endereço, o status de powerup é lido quando o jogador bate em um  que contém um powerup. Se o jogador está pequeno, o bloco spawna um super mushroom. Se o jogador está qualquer coisa exceto pequeno, o bloco spawna uma pena ou uma flor de fogo, dependendo de qual bloco você colocou.

O endereço é também escrito quando o jogador realmente toca em um powerup. Se o jogador toca uma pena, o endereço é definido para 02, dando a entender que o jogador agora tem o powerup de capa.

Capítulo 4: Operações Básicas

Assembly é capaz de muitas e muitas coisas, mas para iniciar do básico e manter as coisas simples, você será introduzido as operações básicas primeiro.

Carregando valores

Assembly é capaz de “operações de carregamento”. Você pode por exemplo carregar o valor de um endereço RAM. Acessível se você quer **ler** o contador de moedas ou o status de powerup por exemplo, ou se você quer **carregar qualquer número que quiser** armazenar em algum lugar depois.

Armazenando valores

Assembly também é capaz de “operações de armazenamento”. Com isto você pode manipular as mecânicas do jogo, como configurar o powerup para cape, ou mudar a música da fase.

Acrescentando/reduzindo valores

Assembly é capaz de acrescentar e reduzir números. Um simples exemplo disso é aumentar a vida do Mario em 1, ou diminuir o tempo em 1.

Comparando valores e pulando para outros bits de código

Finalmente, assembly é capaz de comparar valores. Um exemplo disso é dar ao player uma vida extra depois de coletar 100 moedas, porque o jogo basicamente checa “**se** o jogador possui exatamente 100 moedas **então** dê uma vida extra e defina suas moedas para 0”. Se a condição de 100 não é encontrada, o código pula o código de vidas extra e o ignora.

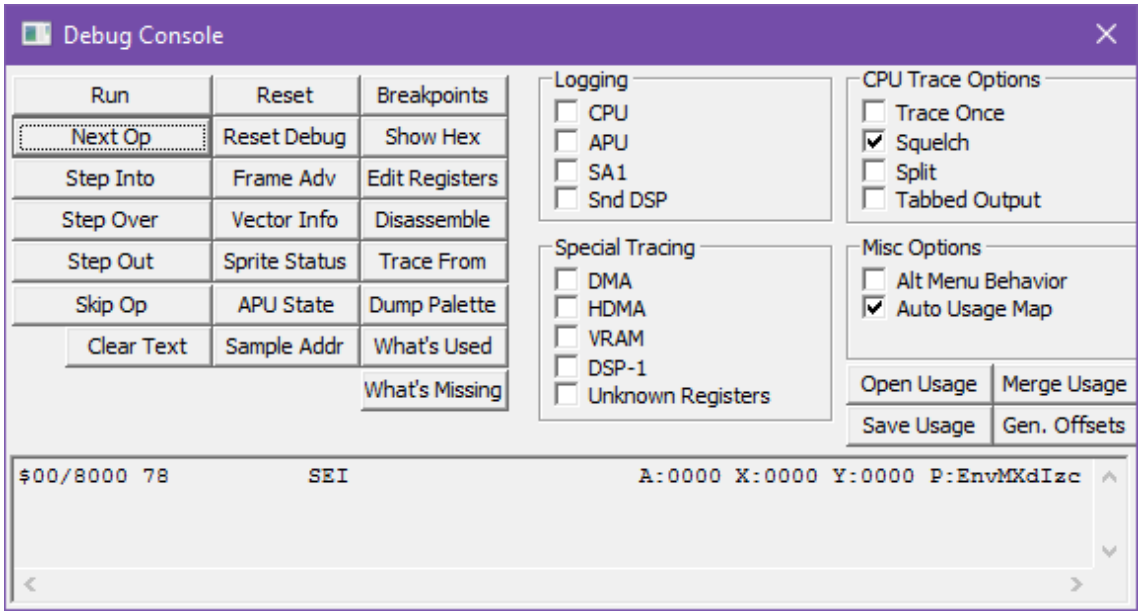
Você verá exemplos dessas 4 operações mais à frente no tutorial.

Capítulo 5: Manipulando as mecânicas do jogo

Para um começo suave com o aprendizado de ASM, você precisará de um certo emulador: “Geiser’s snes9x debugger”. Procure no Google e você encontrará websites como Zophar, Romhacking.net e até a SMWCentral. Sinta-se livre para baixar a última versão que encontrar. Este emulador lhe permitirá brincar com o funcionamento interno do SMW sem precisar escrever uma única linha de código. É importante que você realmente siga os passos mencionados. Não leia isso e pense “ah, ok, eu entendi isso” porque você não irá aprender nada.

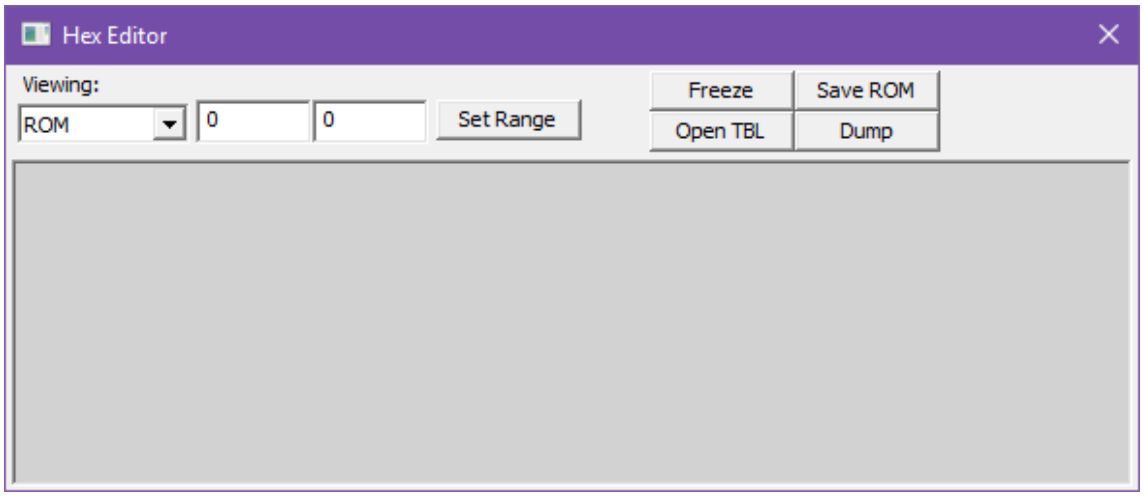
Notavelmente o emulador é desatualizado, mas ainda é excelente para aprender ASM. Também não consome muito da CPU e é muito fácil de usar. É apenas uma versão modificada do SNES9X (avá).

Após rodar este emulador, você irá dar de cara com duas janelas: o emulador padrão, e a seguinte janela chamada Debug Console:

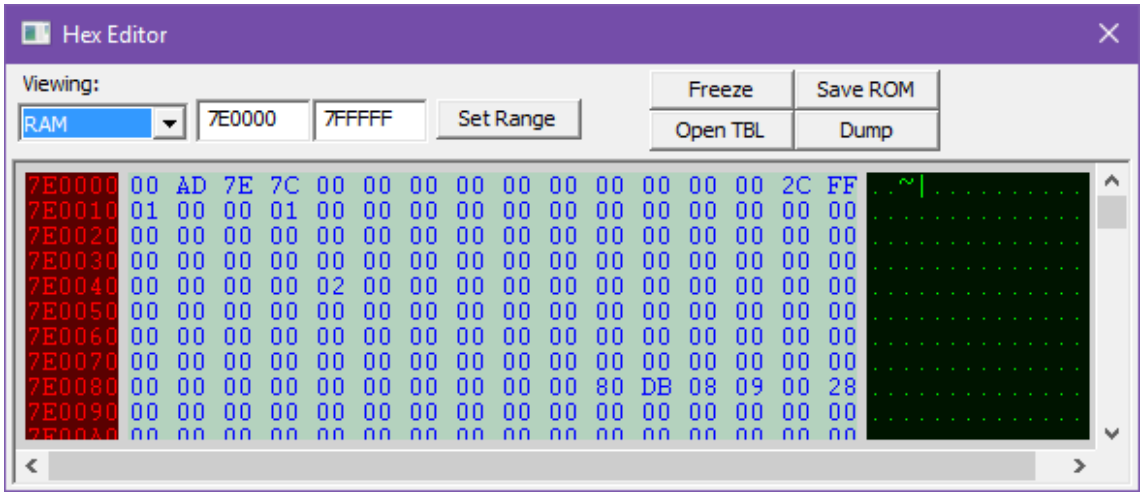


O Debug Console permite *depurar* ROMs. Depurar é basicamente o processo de encontrar e consertar bugs de software. Eu recomendo você não se apegar a esse emulador, porque no capítulo sobre depuração vamos utilizar um depurador claramente superior.

Carregue a ROM como você faria em qualquer emulador, então certifique-se de pressionar o botão **Run** se o jogo não tiver iniciado ainda. Agora pressione o botão **Show Hex**. Você verá um editor Hexadecimal (ou Hex Editor):



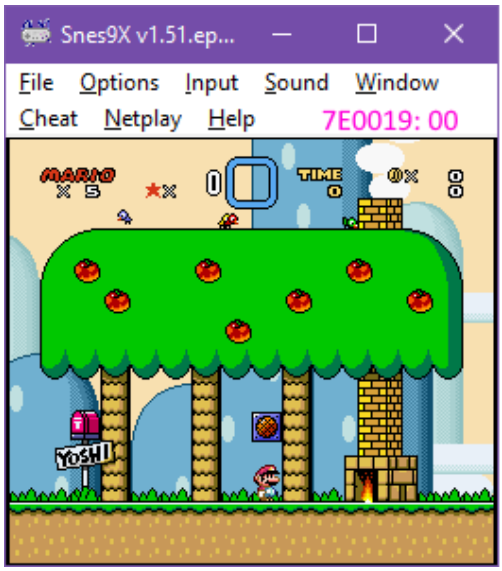
Nesta janela, clique em ROM e selecione RAM. Agora você verá a janela sendo preenchida com muitos números. Todos esses números são Hexadecimais:

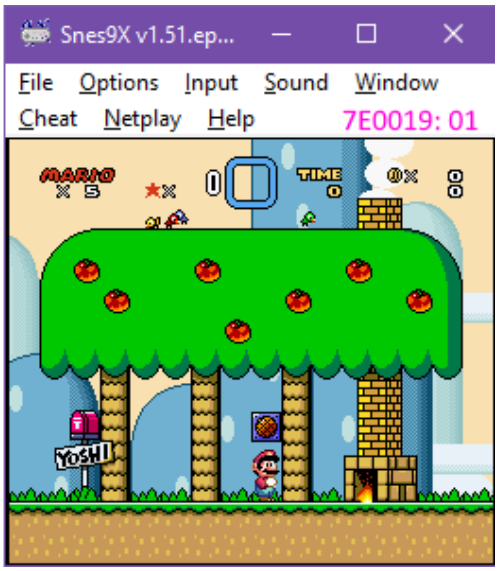


O que você vê aqui é a RAM do Super Mario World, e ela irá mudar constantemente enquanto o jogo roda. Nós estamos falando sobre milhares de mudanças em menos de um segundo. Isso acontece porque Super Mario World roda código, e o código utiliza RAM para armazenar e ler suas variáveis.

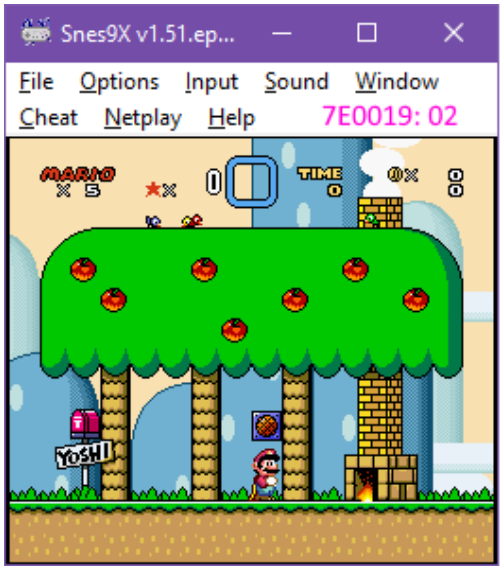
A fim de aprender a manipular o básico do jogo, nós iremos brincar um pouco com a RAM. Primeiro, entre em qualquer level. Para não ser interrompido pelo brilhante design da Nintendo de colocar um inimigo na primeira tela, evite entrar na Yoshi's Island 1 porque suas mãos estarão ocupadas com o debugger console ao invés da gameplay em si.

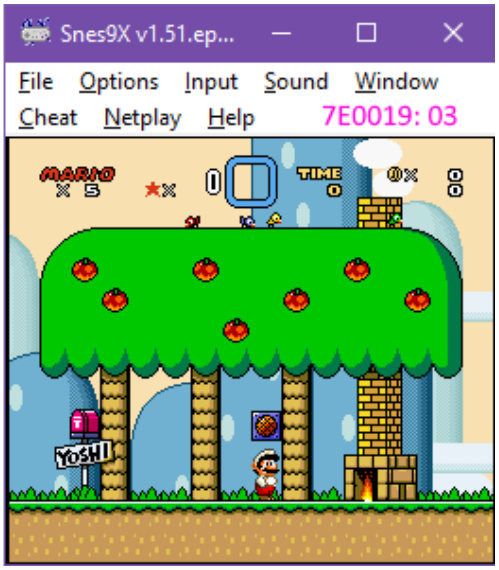
Na primeira caixa de input que diz "7E0000", substitua-o por "7E0019" e clique em Set Range. O texto vermelho no topo agora mostrará 7E0019. O primeiro número a direita do vermelho deverá estar como 00. Clique nele e substitua pelo número 01 (não esqueça do 0). O que aconteceu? Clique na janela do emulador para voltar o foco no jogo e veja você mesmo:



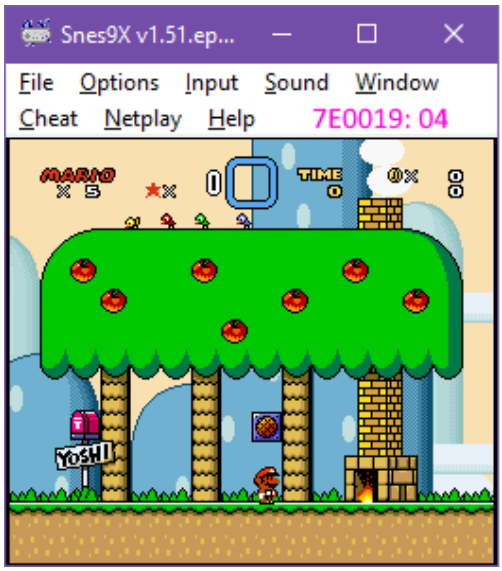


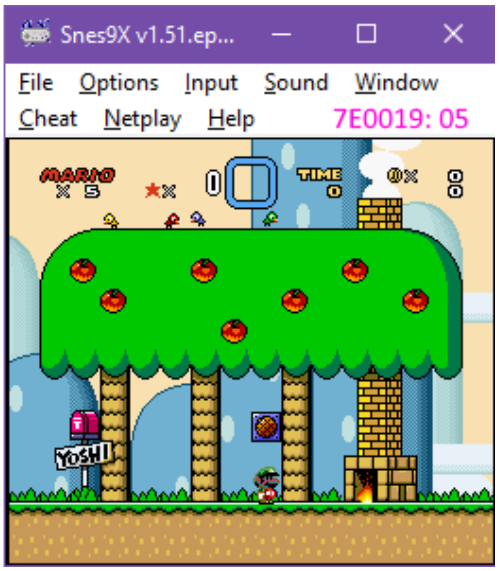
Certo. Mario se tornou Super Mario. A mudança na RAM é imediatamente visível na gameplay. Mario cresceu. Também, por trás das cenas, o jogo notou a mudança no powerup status e começou a exibir gráficos apropriados e paletas. O jogo é programado dessa forma. Claro, você também pode definir o powerup status para 02 e 03:





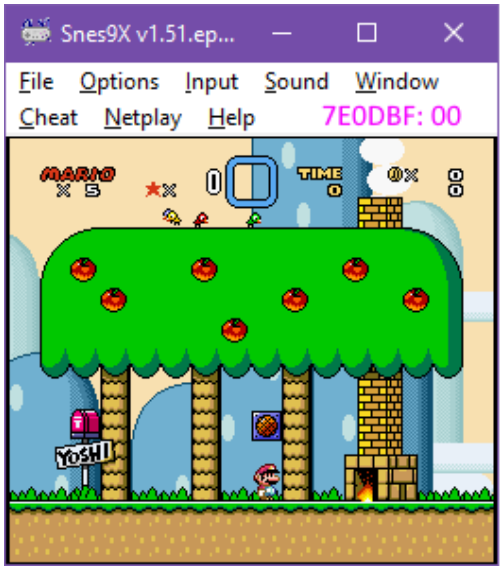
Mas... o que acontece depois do 03? Welp:

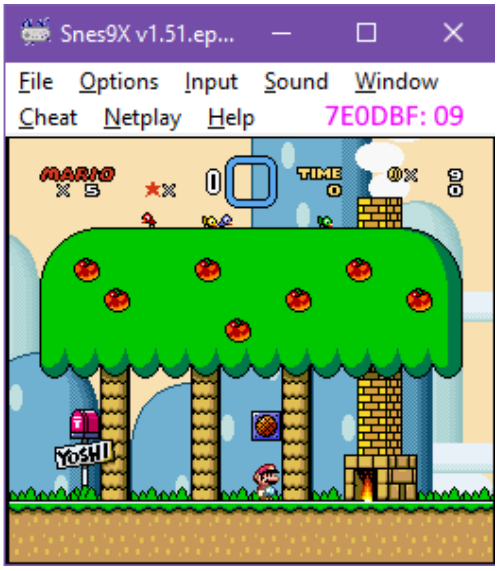




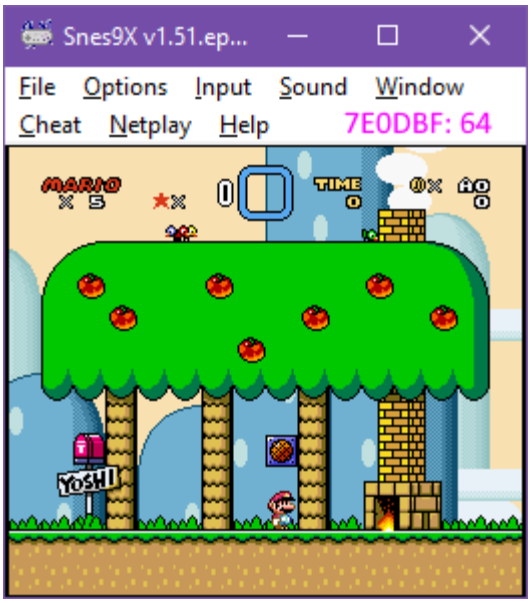
Woah! De repente, Mario parece todo estranho! O jogo não reconhece powerup status 04 e 05, e transformou Mario em um Halloween-y e Christmas-y porque começou a ler dados falsos para alterar a aparência do Mario, e quando você toca um powerup o jogo simplesmente crasha maioria das vezes. A lição aqui é que quando você escreve ASM, não acidentalmente escreva valores inválidos ao endereço RAM.

Bora brincar com outro endereço. Vamos manipular o contador de moedas. Substitua o 7E0019 por 7E0BDF, e coloque 09 na RAM. Você verá o contador de moedas repentinamente mudar de 0 para 9:





Incrível. Agora nós aprendemos como manipular o contador de moedas. Vamos para 100 moedas na tentativa de dar ao jogador 1 vida. Mude o valor de 09 para 64. 0x64 em Hexadecimal é 100. Você verá algo estranho acontecer:



Como você pode ver, o contador de moedas se tornou algo estranho. Está dizendo “A0” o que é uma quantidade estranha de moedas. Nós nunca vemos isso acontecer em Super Mario World durante uma gameplay normal.

Então como nós conseguimos 100 moedas e também uma vida com isso? Como mencionado anteriormente no tutorial, existe um gerenciador de moedas no SMW: RAM $7E13CC.

Mude o endereço RAM que editamos de 64 para 63. Você verá o contador de moedas mostrar “99”. Então, vá para a RAM $7E13CC dentro do Hex Editor e mude o endereço para 01. Como pode ver, Mario ganhará uma vida, e o contador reseta. No entanto, o endereço que editamos também reseta imediatamente. Mude para 01, e ele reseta de novo enquanto adiciona 1 moeda ao contador. Vamos altera-lo para 02 desta vez. O jogo adiciona 2 moedas ao contador! O jogo normalmente nunca adiciona múltiplas moedas de uma vez ao contador, mas é capaz disso de qualquer forma. Apenas por diversão, mude o valor para FF agora – o maior número possível em hexadecimal. Veja como o contador repentinamente aumenta incrivelmente rápido, enquanto o valor que você inseriu na RAM claramente diminui em número. É assim que o gerenciador de moedas funciona. “Então para que é usado o 7E0BDF?” Você pode utiliza-lo para comparar valores de moedas (por exemplo checar se o jogador tem 43 moedas). Isso é bom para ler, ao invés de escrever.

Finalmente, vá para o endereço 7E1DFB e mude o valor para 03. De repente, a música de água toca.

Se você quiser, fique a vontade para brincar com outros endereços também para se acostumar como o jogo manipula certos elementos, como as vidas extras, a posição X do jogador dentro do level, a velocidade do jogador, pause/despause, etc. Apenas digite Ctrl+F no RAM map da SMWC e procure por qualquer coisa a qual você queira brincar. Você precisa aprender como o jogo funciona por trás das câmeras desse jeito a fim de escrever códigos e entender quais são os resultados. Você também precisa de alguma prática navegando e procurando no RAM map.

Capítulo 6: Introduzindo Codificação

Para criar códigos em ASM, você tem que entender como o processador lê todo o código e o executa, ASM é executado pelo processador do Super Nintendo, e a fim de entendermos como o processador executa os opcodes do ASM, você tem que entender o que os opcodes realmente fazem. Para isso, consulte meu outro tutorial de ASM para detalhes mais aprofundados sobre os opcodes. Aqui eu vou focar nos opcodes e em especial suas aplicações práticas em SMW.

Carregamento, armazenamento e o acumulador

ASM é escrito por opcodes com parâmetros. Um opcode é um acrônimo de 3 letras. Um opcode pode não ter nenhum parâmetro ou um parâmetro o qual é 1, 2 ou 3 bytes em tamanho. Vamos introduzir você ao primeiro opcode: **LDA**

```
LDA #$02
```

Como explicado acima, um opcode é um acrônimo de 3 letras. Neste caso, este opcode tem um parâmetro de valor imediato.

O que LDA faz é carregar o valor no acumulador. Este valor pode ser tanto imediato (um número com prefixo #$) ou um valor de um endereço (um número com prefixo $). Nesse caso, o que este código faz é carregar o valor #$02 no registro do **acumulador** (ou "A"). É por isso que é chamado de LDA; LDA significa LoaD into Accumulator.

O que o acumulador é, então? O acumulador é um registro do 65c816 (mas para simplificar vamos chamar o 65c816 de SNES a partir de agora). Ele é capaz de memorizar um número para ser usado mais tarde. Também é capaz de ser modificado por matemática, como aumentando ou diminuindo seu valor por 1, 6, 14, etc. Com "uso mais tarde", imagine coisas como armazenar o valor na RAM, comparar o valor com a RAM, e assim por diante.

-POINT OF ADVICE-

Basicamente, imagine o acumulador como você lendo e memorizando um número. Olhe para o tempo. Que horas são? Memorize isso. Vamos supor que seja 14:00. Viu, você leu um número de algum lugar e o memorizou, e você pode fazer diversas coisas com esse número como... dizer a um amigo que horas são!

O segundo opcode que você vai aprender é o STA. STA significa "Store Accumulator", e o que ele faz é armazenar o valor dentro do acumulador em algum lugar. Exemplo:

```
STA $7E0019
```

O que este código faz é armazenar qualquer que seja o número dentro do A, para o endereço especificado após STA. Com este opcode você **escreve** para endereços. Este é o seu opcode primário para manipular endereços, e portanto, o jogo.

Existe uma pequena variação do STA chamada STZ – STore Zero – define um valor para 00 imediatamente. Ele não precisa do A, ele apenas modifica o A. Simplesmente escreve 00 para um endereço.

Finalmente, nós temos outro opcode importante o qual é sempre usado para finalizar uma função (uma “sub-rotina”) que você escreveu. Na verdade, ele possui duas variações:

```
RTS
```

```
RTL
```

Ambos servem para o mesmo propósito, apenas são usados em diferentes contextos que você não precisa saber por enquanto. Se um patch ou ferramenta lhe diz para terminar uma sub-rotina com RTS, então você o usa. O mesmo com RTL. RTS e RTL significam respectivamente “ReTurn from Subroutine” e “ReTurn from subroutine Long”. Uma regra geral para seguir é “se uma sub-rotina é chamada com um JSR, termine com RTS. Se chamada com um JSL, termine com RTL”. Falaremos sobre sub-rotinas mais tarde neste capítulo.

Então o que aprendemos até então é que com LDA podemos carregar um valor no A, com STA armazenamos A em um endereço, e com RTS ou RTL finalizamos a função. Combinando esses 3 opcodes é possível escrever um código para alterar o powerup do Mario:

```
LDA #$01
STA $7E0019
RTS
```

Se você lembra daquele exemplo do RAM Map, sabemos que o valor #$01 de $7E0019 significa “Mario Grande”. Este código carrega 01 no A, e armazena para a RAM $7E0019. Então a rotina termina com um RTS.

Comparando valores, branches e labels

ASM também é capaz de operações condicionais. Você pode fazer com que se uma condição for encontrada, pode ignorar um tanto de código. Há também dois tipos de opcodes para fazer isso. O primeiro é o CMP:

```
CMP #$02
```

CMP (CoMPare), compara o conteúdo dentro do A com o parâmetro que ele tem. Suponha que A tem o valor 01. Compare 01 com 02, eles não coincidem.

E agora? Existem opcodes que podem ignorar um código dependendo se uma comparação coincide ou não, eles são chamados de **branches**. Exemplo:

```
BNE skip
```

BNE (Branch if Not Equal) faz exatamente o que seu nome diz. Se uma comparação é incompatível, o branch será **"tomado"**; o opcode pula para a **label** chamada "skip". Também existe um BEQ (Branch if EQual. Este código dá branch se a comparação É compatível.

Mas o que é uma label? Uma label é basicamente qualquer palavra usada para marcar uma localização dentro do código. Você verá um exemplo daqui a pouco.

Então vamos combinar nosso conhecimento. Sabemos como carregar e armazenar valores, e também como ignorar certo código dependendo de comparações e seus resultados. Sabendo disso, podemos facilmente escrever um código que muda a música do level quando Mario estiver com o powerup de fogo:

```
LDA $7E0019
CMP #$03
BNE skip
LDA #$05
STA $7E1DFB
skip:
RTS
```

Para explicar o que este código faz: o powerup status é carregado no acumulador, então é checado se ele é #$03 (fire Mario). Se não, pula para o RTS, finalizando a rotina. Caso contrário, reproduz a música #$05 e também finaliza a rotina.

Fazendo Matemática

O SNES é capaz de operações básicas de + e -, que funciona editando os valores dentro do acumulador ou um endereço.

Para aumentar A por 1, você pode usar o seguinte opcode:

```
INC A
```

INC significa INCrease. Seu parâmetro é A. O que ele faz é aumentar o número atual dentro do acumulador por 1. Se o acumulador tiver o número #$45, agora ele é #$46.

E se você quiser aumentar em A mais de 1? Escrevendo "INC A" #$25 vezes para aumentar A por #$25 parece meio desajeitado. Felizmente existe um opcode que pode lidar com adições maiores:

```
CLC
ADC #$25
```

ADC significa "ADd with Carry" e adiciona o que estiver em A com o parâmetro de ADC.

O que este código faz é adicionar #$25 ao acumulador. Lembre-se que estamos trabalhando com números hexadecimais. Se o acumulador tem o número #$45, depois dessa operação ele será #$6A – 106 em decimal.

No entanto, como você pode ver, o opcode é precedido por “CLC”. CLC significa ‘CLear Carry”. Porque ADC é “add **with** carry”, você quer o carry limpo 99% das vezes, caso contrário você terminará com 6B invés de 6A – o carry contabiliza como +1 neste caso.

Há também opcodes que podem diminuir A. Para diminuir A por 1, você pode simplesmente usar:

```
DEC A
```

...que significa DECrease A. Se o acumulador era #$45, agora é #$44.

Você também pode diminuir A com números maiores.

```
SEC
SBC #$25
```

SBC significa “SuBstract with Carry” e subtrai o que estiver em A com o parâmetro de SBC. SEC é basicamente contrapartida de SBC usado para subtrações.

ADC e SBC podem também modificar A quando esses opcodes tem um endereço fornecido como um parâmetro:

```
SEC
SBC $7E0019
```

Nesse caso, o valor do acumulador é subtraído com o valor **dentro** de $7E0019. Ele não modifica 7E0019. SBC modifica A. O mesmo acontece com ADC.

INC e DEC são capazes de aumentar/diminuir endereços diretamente sem afetar o acumulador.

```
INC $0019
```

Isso aumenta 7E0019 por 1, mas algo está errado...

Woah! O 7E de 7E0019 está faltando. Como? Bom, INC $7E0019 não existe. INC com 6 números como parâmetro não existe. O mesmo com DEC. Portanto, o parâmetro é encurtado para $0019. Mas sobre o que é isso?

Encurtando endereços

De fato, é possível usar uma notação abreviada para endereços, mas eles têm uma certa condição. Você pode mudar essas condições manualmente usando códigos mais avançados, mas por enquanto vamos focar nas condições básicas que ocorrem 99% das vezes.

Você pode fazer um endereço de 4 dígitos invés de 6 se:

1. O endereço está no bank 7E (o “bank” sendo os primeiros 2 dígitos de um endereço de 6 dígitos).

2. Os números finais sendo entre 0000 e 1FFF (hexadecimal)

Com isso, você pode por exemplo abreviar $7E1344 para $1344.

Você também pode fazer um endereço de 2 dígitos invés de 6 ou 4 se:

1. O endereço está no banco 7E
2. Os últimos quatro números estão entre 0000 e 00FF (hexadecimal)

Com isso, você pode por exemplo abreviar $7E0019 para $19, o que resulta no seguinte código:

```
STA $7E0019
STA $19
```

Ambos fazem exatamente a mesma coisa, exceto que o segundo código é mais curto! Esta técnica também lida com o INC $7E0019 sendo inexistente. Você pode simplesmente usar INC $19.

Indexação e operações de stack

Existem mais dois conceitos importantes em codificação, são eles “indexação” e “o stack”. Com indexação você lida com registros similares ao A, chamados X e Y. Com o stack você pode preservar valores dentro de A, X e Y em uma certa (predefinida) localização na RAM do SMW. Para mais informação sobre Indexação e o stack, por favor consulte meu outro tutorial (“Assembly for the SNES”), capítulos **18** e **21**.

Aplicando o que aprendemos até aqui

Vamos combinar maior parte dos opcodes que aprendemos para pegar alguma prática. Iremos escrever códigos que aumentam as Bonus Stars do Mario por 9 quando Mario tiver uma capa e faça ele pequeno novamente. O endereço que gerencia as Bonus Stars é $7E0F48. Também vamos encurtar os endereços ao mínimo ao longo do caminho:

```
LDA $19
CMP #$02        ;Checa pelo status de cape
BNE NoCape      ;Se não é cape, ignora o próximo bloco de código
LDA $0F48
CLC             ;Adiciona 09 às Bonus Stars
ADC #$09
STA $0F48
STZ $19         ;Define o powerup status para - 0, Small Mario
NoCape:
RTS             ;Finalizado
```

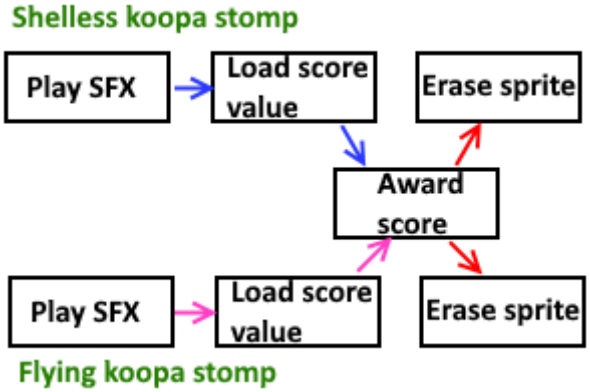
Note que eu repentinamente comecei a escrever texto dentro do meu código explicando coisas. Isto é chamado de **comentários**. Comentários são ignorados pelo assembler.

Comentários sempre começam com um “;”. Graças aos comentários, eu não tenho mais que explicar o que o código faz. (espero).

Mais uma coisa: Sub-rotinas

Anteriormente neste capítulo eu mencionei as sub-rotinas. Elas são blocos de código que podem ser chamadas durante outro bloco de código. Chamando sub-rotinas é bem conveniente, porque você não precisa escrever código duplicado.

Suponha a seguinte situação. Em Super Mario World, você obtém pontos ganhos quando esmaga um inimigo, certo? Você não precisa escrever a mesma rotina de ganho de pontos para cada inimigo, certo? É por isso que isso pode ser escrito como uma “sub-rotina”.



Como você pode ver, esta rotina está localizada fora do fluxo de código em ambas situações. A rotina está sendo “chamada”. Há dois opcodes para chamar sub-rotinas: JSL e JSR. Por enquanto você não necessariamente precisa saber a diferença, mas pelo bem da simplicidade, vamos usar JSL.

```
LDA #$03
STA $1DFC
LDA #$01
JSL AwardScore                ;A = 01
STZ $SpriteStatusRAMaddress   ;Por agora, vou usar nomes invés de
números
[...]
[...]

AwardScore:                   ;Quando o código chega aqui, A permanece
inalterado
STA $somescoreRAMaddress      ;É como chamar funções com parâmetros
RTL                           ;Portanto, armazenamos 01 neste endereço
```

O que este código faz é reproduzir um efeito sonoro (escrito para $1DFC), carrega o valor de pontuação 01 para o A, então chama uma sub-rotina que trata da atribuição de pontos ao jogador. Uma vez que a sub-rotina é feita, finaliza com RTL. Então o código pula de volta para aquele STZ.

RTL permite chamar a sub-rotina de qualquer lugar. Desse jeito, na sub-rotina, você não tem que especificar onde exatamente o código precisa voltar. Ele não precisa saber, se precisa voltar para a rotina do Shelless koopa, ou o flying koopa. Ele apenas retorna de onde ele veio.

Phew! Este foi um longo capítulo! Sabemos agora como escrever um código básico. Tem apenas um problema: Como inserimos código no Super Mario World? Não seria incrível se realmente começássemos a rodar código no SMW? SEU código?

De agora em diante, as coisas vão ficar **reais**. Deixe me introduzir a você...

Capítulo 7: Gopher Popcorn Stew (também conhecido como blocktool)

Gopher Popcorn Stew, ou GPS, é uma ferramenta de inserção de blocos criado por p4plus2. O porquê da ferramenta se chamar dessa forma permanece um mistério até o dia de hoje.

O terreno dentro dos levels é constituído de blocos. Chão, moedas, canos, todos esses são blocos (ou em termos de SMW, “map16 tiles”). O que o GPS faz é inserir o seu código para tiles específicos. Você pode por exemplo, executar um código quando o jogador toca um turnblock pela parte de cima. Isto é uma perfeita oportunidade para brincar com ASM.

Configurando o GPS

Esta seção será um mini tutorial sobre como fazer o GPS funcionar, mas para mais informações consulte o **readme.txt** da ferramenta.

Primeiro de tudo, tenha certeza de que sua ROM do SMW tenha sido editada pelo menos uma vez. Mova qualquer coisa e clique para salvar. É o suficiente para fazer o GPS funcionar. Coloque esta ROM e um atalho do Lunar Magic na pasta do GPS.

Segundo, crie um novo arquivo de texto (apenas dê um clique com o botão direito e selecione Documento de Texto no menu **Novo**). Nomeie-o “insert.bat”. Se você não consegue ver a extensão do arquivo, procure no google como mostrar extensões de arquivos.

Clique direito no arquivo e selecione **Edit**. Você agora pode editar o arquivo bat como qualquer arquivo de texto. Escreva o seguinte:

```
gps.exe smw.smc
@pause
```

Onde está “smw.smc” deve ser o nome da sua ROM. Eu recomendo alterar de “Super Mario World.smc” para “smw.smc” para evitar digitar muito quando utilizar várias ferramentas. Ctrl+s para salvar o arquivo, e saia. Você agora tem o seu insersor pronto.

Preparando um **arquivo ASM**

*Nota de tradução: Nas versões recentes do GPS existe um arquivo **template.asm**, porém esse passo é importante para você praticar ao invés de pegar tudo pronto).*

Navegue até a pasta “blocks”. Ela estará vazia. Crie um novo documento de texto novamente, mas agora nomeie-o “example.asm” Certifique-se de associar arquivos *.asm* com Notepad ou qualquer outro editor de texto da sua preferência. Abra o arquivo e cole o seguinte bloco de código nele:

```
db $42
JMP MarioBelow : JMP MarioAbove : JMP MarioSide
JMP SpriteV : JMP SpriteH : JMP MarioCape : JMP MarioFireball
JMP TopCorner : JMP BodyInside : JMP HeadInside

MarioBelow:
RTL
MarioAbove:
RTL
MarioSide:
RTL

TopCorner:
RTL
BodyInside:
RTL
HeadInside:
RTL

SpriteV:
RTL
SpriteH:
RTL

MarioCape:
RTL
MarioFireball:
RTL
```

Então salve. Este é o template básico para **custom blocks**. Eles sempre começam com db $42 por razões que você não precisa saber. Então é seguido por uma **jump table** (olhe para todos aqueles JMP’s). E por fim, há uma lista de labels com um RTL no fim.

Cada label executa sob certas condições. MarioBelow executa quando Mario toca o bloco por baixo. MarioAbove é o mesmo, mas por cima. MarioSide executa quando Mario toca o bloco por ambos os lados.

TopCorner executa quando Mario toca o canto superior direito ou esquerdo do bloco. BodyInside executa quando o corpo do Mario está dentro do bloco, e HeadInside executa quando a cabeça do Mario está dentro do bloco.

SpriteV e SpriteH executa quando qualquer sprite tocar o bloco por cima / baixo e esquerda / direita, respectivamente.

Por fim, MarioCape executa quando Mario acerta o bloco com a capa, e MarioFireball executa quando a bola de fogo do Mario acerta o bloco.

Contanto que essas condições sejam encontradas, o código executa a **cada quadro**. Isto significa que se o Mario ficar em cima de um bloco sólido, o código será executado constantemente, 60 vezes por segundo.

Seu primeiro custom block!

Vamos escrever um bloco no qual quando Mario andar sobre ele, mude o powerup status para 02, fazendo dele o Cape Mario!

Localize o seguinte bloco de código dentro do arquivo ASM:

```
MarioAbove:
RTL
```

Mude para:

```
MarioAbove:
LDA #$02
STA $19
RTL
```

Como aprendemos anteriormente no tutorial, isto altera a RAM $7E0019 para 02, a qual muda o Mario normal para o Cape Mario. Salve o ASM, mas não o feche ainda, porque iremos edita-lo novamente.

Volte para a pasta do GPS e abra o **list.txt**. Escreva:

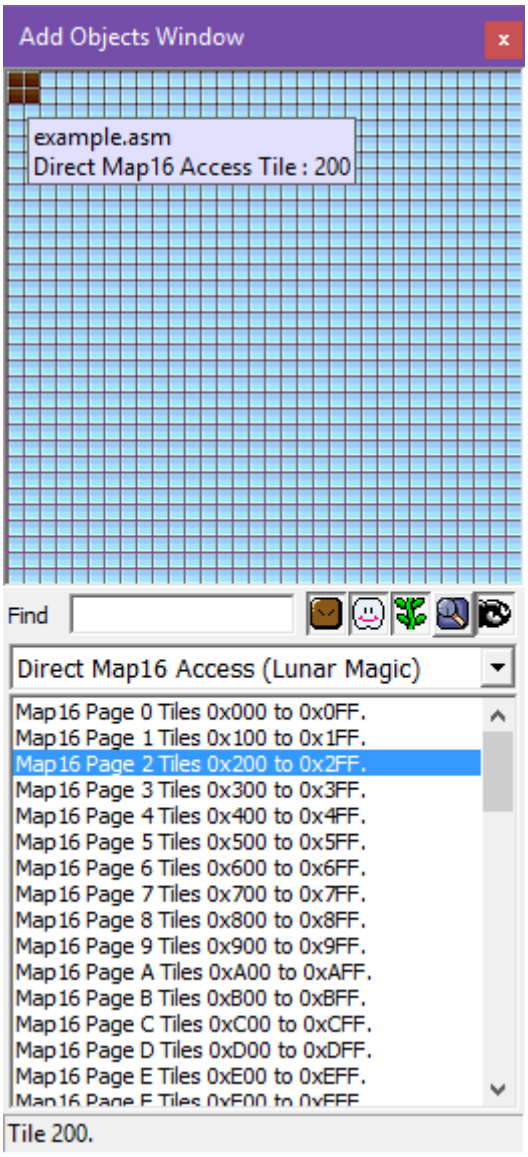
```
200 example.asm
```

Isso define onde o código que escrevemos deve ser inserido. Este código irá executar quando Mario tocar o map16 tile 200. 200 é em hexadecimal, a propósito, então é a numeração dos map16 tiles.

Estamos quase lá. Rode o arquivo .bat que fizemos antes (“inserter.bat”). Se disser que os blocos foram inseridos com sucesso, parabéns! Se não, tente encontrar o que deu errado durante essas etapas lendo o error log. Essas etapas são literalmente perfeitas.

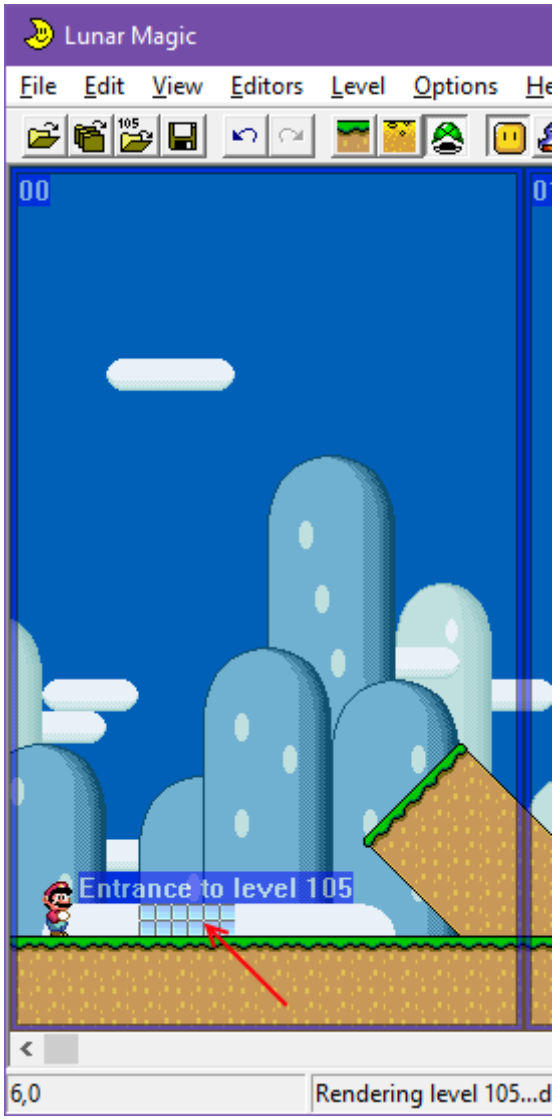
Testando o bloco

Nosso bloco foi inserido. Hora de testá-lo. Se você tiver o Lunar Magic aberto este tempo todo, simplesmente recarregue sua rom e vá para a tela de inserção de objetos e use o **Direct Map 16 Access**. Então desça até a página 2, nela tem os tiles 0x200 até 0x2FF. Se você passar o cursor por cima do tile 0x200, irá ver a seguinte tooltip:



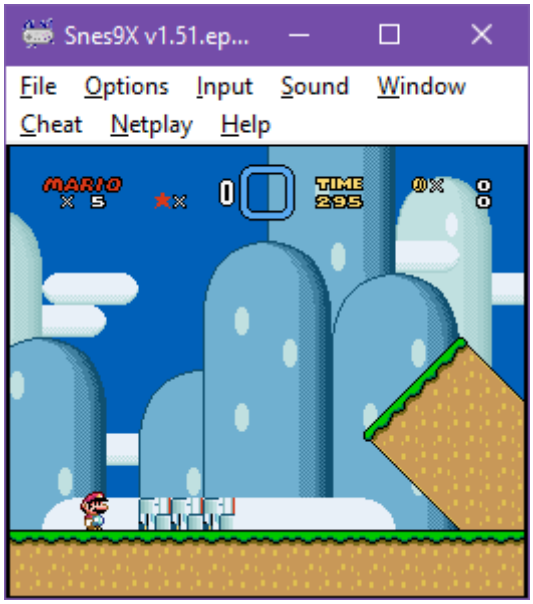
Se você não ver a tooltip, não se preocupe. Simplesmente reabra o Lunar Magic e tente novamente. Esta tooltip indica que o bloco agora tem o nosso código.

Insira o bloco no level

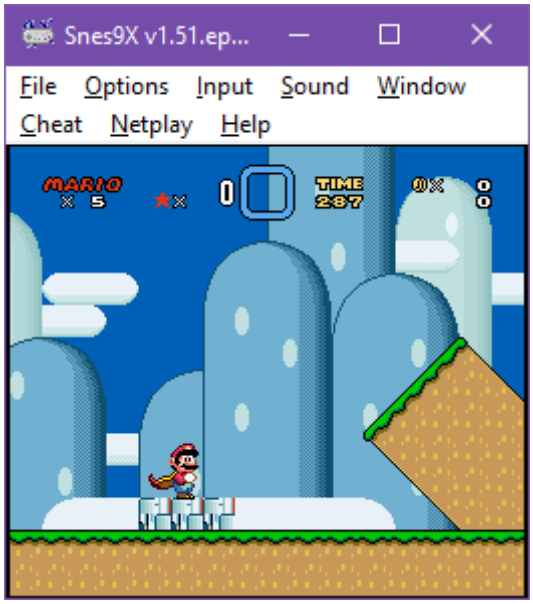


O bloco terá quadrados azuis por padrão. Está ok. Dê a ele gráficos se você quiser, mas isso não é relevante para este tutorial. Salve o level e rode a rom no seu emulador.

É assim que o level será exibido:



O bloco claramente tem gráficos diferentes, mas não deixe isso incomodá-lo, é apenas um glitch visual porque não demos gráficos customizados a ele. Agora, pule em cima do bloco. Você verá o bloco acionar nosso código:



É isso aí! É assim que você escreve custom blocks para Super Mario World!

Agora você deve estar pensando, por quê estamos escrevendo blocos? Blocos são as coisas mais fáceis de fazer para Super Mario World. ASM para SMW consiste de 4 grandes partes:

1. Custom Blocks
2. Custom Sprites
3. Custom Patches
4. “UberASM”

Custom blocks são as coisas mais fáceis de escrever enquanto custom patches são as mais difíceis.

Patches modificam o núcleo principal do jogo, fazendo ele rodar diferente. “UberASM” é um patch por exemplo, descrito abaixo.

Para sprites, eles são entidades com certo comportamento e interação. A dificuldade para “codar” (termo informal mais utilizado da palavra “codificar”) depende da complexidade do comportamento que o sprite deve ter.

Blocos são apenas... blocos dentro do level, fixos ali, esperando para serem tocados pelo jogador ou entidades de sprites. Eles não têm nenhum movimento, então o código pode ser incrivelmente fácil de escrever. Isso permitirá uma fácil prática de ASM.

Por fim, UberASM é um patch feito por p4plus2 que permite executar códigos dependendo do estado atual do jogo, seja um certo level ou certo game mode. É um patch muito extenso e completo, permitindo códigos bem flexíveis.

---

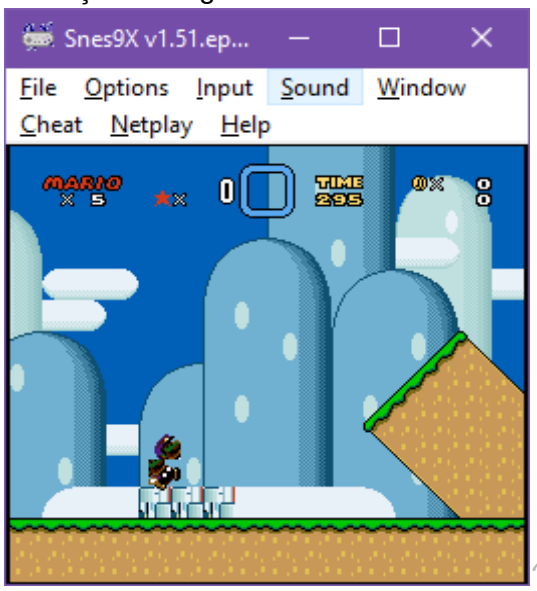
*Nota de tradução: Nessa época ainda não existia **UberASM Tool**. Atualmente não há necessidade de utilizar este modelo de patch.*

Voltando ao *coding*, eu quero que você faça mais um bloco. Edite o arquivo ASM que inserimos anteriormente. Substitua nosso código com o seguinte:

```
MarioAbove:
INC $19
RTL
```

O que este código faz é **aumentar** o powerup status. Salve o ASM e rode o arquivo .bat. Não há necessidade de deletar e reinserir os blocos no level. O código é automaticamente atualizado para aqueles blocos. Agora, rode a ROM novamente.

Entre no level e pule no bloco. Você verá algo muito estranho acontecendo. Mário começará a bugar loucamente:



“Me mate” ~ INC $19 Mario

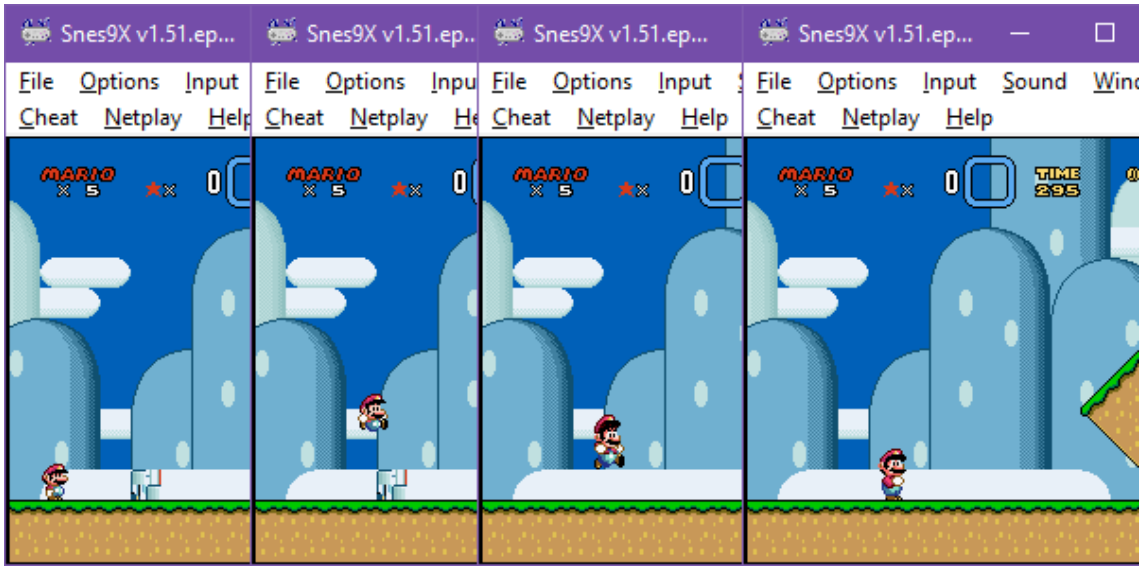
Isso é por que o código executa **a cada frame**, não apenas uma vez. É muito importante ter isso em mente. O que nosso código faz agora é aumentar o powerup status constantemente. Isto é, o powerup status será aumentado 60 vezes por segundo. Como resultado, ele está passando rapidamente por estados inválidos de powerup porque o endereço RAM passou loooonge do valor #$03. Uma vez que chega no valor #$FF, ele volta ao #$00.

Se você quer criar um bloco que aumenta o tamanho do Mario apenas uma vez, terá que mudar o bloco para um tile em branco (blank tile) uma vez que ele é tocado, então o código nunca será executado novamente. Porque o map16 tile (0x200) muda para um blank tile (0x25), e nós não temos código no blank tile, nosso código basicamente não existirá no level mais.

Substitua nosso MarioAbove novamente, mas com o seguinte código dessa vez:

```
MarioAbove:
INC $19             ;Aumenta o powerup status
LDA #$02            ;\ Bloco a ser gerado: blank tile
STA $9C             ;/ Cheque a RAM Map para o 7E009C.
JSL $00BEB0         ; Rotina de Block Change
RTL
```

Aqui está o resultado:



Como você pode ver, quando o Mario pisa no bloco, ele desaparece e Mario se torna o Super Mario. Se Mario pisar enquanto tivesse com a Cape, ele se tornaria o Fire Mario. Se fosse Fire Mario se tornaria um Mario bugado.

Se você não quer que aquilo aconteça, precisará adicionar um check extra:

```
MarioAbove:
LDA $19
CMP #$03
BEQ +             ;Se for Fire Mario, ignora o código e vai para +
INC $19           ;Aumenta o powerup status
LDA #$02          ;\ Bloco a ser gerado: blank tile
STA $9C           ;/ Cheque a RAM Map 7E009C.
JSL $00BEB0       ; Rotina de Block change
+
RTL
```

O bloco não fará nada se você já estiver como Fire Mario. Se você quiser que o bloco desapareça de qualquer forma, basta mover o + para depois do INC invés do JSL.

Capítulo 8: Patches

Patches são na verdade uma das coisas mais difíceis relacionadas a ASM que você precisa escrever. O processo de escrever um patch consiste em múltiplas partes:

- Encontrar código relevante que você queira “sequestrar” (hijack)
- Fazer o hijacking do código sem quebrar nada
- Escrever o patch
- Certificar-se que não cause nenhum crash ou efeitos colaterais (testando)

É claro, você também precisa configurar um ambiente de patching e tudo para começar.

Dando início

Crie uma pasta separada e coloque sua ROM e um atalho do Lunar Magic. Depois faça o download do **asar** pela SMWCentral:
https://www.smwcentral.net/?p=section&a=details&id=25953

Depois de baixá-lo, coloque asar.exe na mesma pasta. Abra a sua ROM com o Lunar Magic, edite-a uma vez e salve no intuito de expandi-la para 1MB. Por fim renomeie sua ROM para “smw.smc” e será fácil de usar o asar. E é isso! É assim que você configura seu ambiente de patching para este tutorial.

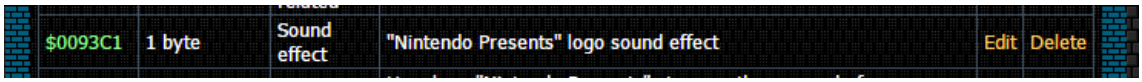
Encontrando códigos relevantes para hijack

“Hijacking” significa repor parte do código original com uma **call** para a sua nova rotina customizada. Em consideração a este tutorial, vamos criar um patch bem simples: Sempre que o Mario morre, define suas moedas para 0.

Nós sabemos o que queremos fazer agora, mas por onde começamos? Você tem duas opções:
- Consultar o **SMW ROM** map:
    - https://www.smwcentral.net/?p=memorymap&game=smw&region=rom
- Consultar o **all.log**:
    - https://www.smwcentral.net/?p=section&a=details&id=21822

ROM Map

O “ROM map” é basicamente um mapa da SMW ROM **original, não modificada**. Isso lhe mostra o que cada endereço faz. Aqui uma imagem de parte do ROM map como exemplo:



Como você pode ver, isso diz o **endereço da SNES ROM** (NÃO PC hex), seu tamanho, seu tipo e sua descrição. Indo à ROM map da SMWCentral e procurando por palavras-chave especificas, você pode encontrar pistas sobre onde certos recursos estão localizados na ROM.

“all.log”

All.log é uma **desmontagem completa (disassembly)** da ROM do Super Mario World. O link acima é uma versão bem comentada. As imagens abaixo são da versão original a qual foi removida do site.

All.log tem um formato simples. Eis aqui algumas linhas de código para melhor entendimento do formato:

```
11508  DATA_00F6CB:                    .db $00,$00,$20,$00
11509
11510  DATA_00F6CF:                    .db $D0,$00,$00,$00,$20,$00,$D0,$00
11511                                   .db $01,$00,$FF,$FF
11512
11513  CODE_00F6DB:        8B          PHB
11514  CODE_00F6DC:        4B          PHK
11515  CODE_00F6DD:        AB          PLB
11516  CODE_00F6DE:        C2 20       REP #$20                ; Accum (16 bit)
11517  CODE_00F6E0:        AD 2A 14    LDA.W $142A
```

Existem dois principais tipos de labels que você precisa saber sobre. DATA_xxxxxx que denota tabelas de dados raw, e CODE_xxxxxx que denota o código de fato. O “xxxxxx” é o endereço ROM daquela instrução.

Entre CODE_xxxxxx e o opcode, você verá um ou mais números hexadecimais. Essa é a versão hex do opcode e seus parâmetros. Graças àqueles números, você saberá quantos bytes compõem uma instrução. **Isto é uma informação importante de saber a fim de fazer patches que não acabem crashando a ROM.**

Encontrando um hijack

Temos então duas versões de documentação: o ROM map e o all.log. Se queremos a rotina de morte, precisamos procurar por coisas relacionadas a morte. Tanto no SMW ROM map ou no all.log você pode começar procurando (ctrl+f) por palavras-chave como “death” ou “kill” (como em “kill the player”). Fazendo isso no ROM map nos leva a “death pose”, mas aquilo não é nada do que precisamos. Pesquisando mais chegamos na ROM **$00F606**.

ROM $00F606 é documentada como “Death SubRoutine (JSL to it to kill Mario)”. Esta é informação que estamos procurando. Sabendo o endereço, podemos ver essa sub-rotina em all.log. Ctrl+f “CODE_00F606”. Mas espere, isso não deu nenhum resultado! Isso é porque all.log na verdade dá labels adequadas para algumas rotinas. Se você não encontrar certo endereço em all.log, tente procurar por endereços adjacentes, por exemplo 00F607, 00F608, etc. ou as palavras-chave relevantes até você ter sorte.

```
11419  KillMario:          A9 90        LDA.B #$90               ; \ Mario Y speed = #$90
11420  CODE_00F608:        85 7D        STA RAM_MarioSpeedY      ; /
11421  CODE_00F60A:        A9 09        LDA.B #$09               ; \
11422  CODE_00F60C:        8D FB 1D     STA.W $1DFB              ; / Change music
11423  CODE_00F60F:        A9 FF        LDA.B #$FF
11424  CODE_00F611:        8D DA 0D     STA.W $0DDA
11425  CODE_00F614:        A9 09        LDA.B #$09               ; \ Animation sequence = Kill Mario
11426  CODE_00F616:        85 71        STA RAM_MarioAnimation   ; /
11427  CODE_00F618:        9C 0D 14     STZ.W RAM_IsSpinJump     ; Spin jump flag = 0
11428  CODE_00F61B:        A9 30        LDA.B #$30
11429  CODE_00F61D:        8D 96 14     STA.W $1496              ; Set hurt frame timer
11430  CODE_00F620:        85 9D        STA RAM_SpritesLocked    ; set lock sprite timer
11431  CODE_00F622:        9C 07 14     STZ.W $1407              ; Cape status = 0
11432  CODE_00F625:        9C 8A 18     STZ.W $188A
11433  Return00F628:       6B           RTL                      ; Return
```

Então localizamos o código que controla a rotina de morte do jogador. A questão é, qual parte do código vamos fazer o hijack? Neste caso, pode ser qualquer parte do código, mas geralmente uma boa prática é fazê-lo onde é um simples LDA e STA, livre de qualquer operação de comparação (como CMP, BEQ, BCC, etc), e de stack também. CODE_00F60A parece perfeito. Localizamos nosso hijack!

Realmente fazendo o hijacking

...sem quebrar nada. Isso é muito importante.

Lembra como eu mencionei que existe um código para chamar sub-rotinas? Agora é a hora de usá-lo: **JSL. JSL é sempre 4 bytes de comprimento, não importa onde aponta.** Você sabe que algo é incrivelmente importante quando eu uso a cor... roxa?

A fim de fazer hijacks que não quebram nada, você precisa fazer uma “contagem de byte”. All.log é bem conveniente para isso, porque você pode ver quantos bytes uma instrução usa. Você pode ver que aquele LDA #$09 usa 2 bytes, enquanto STA $1DFB usa 3 bytes. Nosso JSL usa 4 bytes, então **haverá 1 byte restante.** Isso não está nos impedindo!

Escrevendo o patch

Agora temos toda informação necessária e podemos começar escrevendo nosso patch. De imediato vou introduzir novos conceitos também. Aqui está parte do patch – os barebones:

```
ORG $00F60A                     ; O endereço a ser hijacked
autoclean JSL Hijack            ; Pula para o nosso código
NOP                             ; Substitui o byte restante com um
                                ; opcode que faz literalmente nada

freecode

Hijack:
RTL                             ; Sempre use um RTL depois do JSL
```

ORG não é um opcode. É mais uma instrução para o próprio assembler que significa “origem”. Ele basicamente diz ao assembler “coloque o código nesse endereço ROM”.

O código **sobrescreve** o código existente, ele não é “inserido”.

Nota de tradução: Por isso é necessário backup da ROM cada vez que um patch é inserido, não há como reverter o processo de outra forma.

Por isso também usamos o opcode NOP. Se o byte restante não fosse tratado, isso faria o SNES ler instruções falsas quando chegasse ali.

“autoclean” também é uma instrução para o assembler e para ser bem honesto eu não tenho certeza do que isso faz exatamente, mas eu ouvi que é necessário limpar tudo abaixo do “freecode” caso o patch seja inserido múltiplas vezes depois de fazer edições.

“freecode” diz ao assembler para procurar por espaço livre na ROM a fim de colocar o seu código lá.

O que este código faz é basicamente repor parte do código original com uma chamada (call) para o seu código customizado localizado em algum lugar na ROM. Este patch pode realmente ser inserido sem problemas, e eu aconselho a fazer isso agora mesmo.

Crie um novo documento de texto e nomeie-o “patch.asm”. Copie o código acima e cole nele. Salve o arquivo e rode asar.exe. Siga as instruções. Escreva o nome do patch, Enter, nome da ROM, Enter novamente (Certifique-se de digitar a extensão do arquivo e preste atenção na capitalização, letras maiúsculas são levadas em consideração). Após mostrar que o patch foi inserido sem problemas, podemos testar nosso pequeno patch. Rode o jogo e se mate. Você irá notar alguma coisa.

A música de morte não toca.

A proposta do código era “Mudar a música” de acordo com o all.log. Considerando como o sobrescrevemos, ele não existe mais. Isso conta como **quebrado** apesar de quão trivial pode parecer. Nosso patch não está completo se não **restaurarmos o código original**. Você basicamente faz isso colocando o código original substituído na sua nova rotina:

```
ORG $00F60A                     ; Endereço a ser hijacked
autoclean JSL Hijack            ; Pule para o Codigo Custom
NOP                             ; Sobrescreve o byte restante com um
                                ; opcode que não faz nada


freecode

Hijack:
LDA #$09
STA $1DFB
RTL                             ; Sempre use RTL depois do JSL
```

Insira o patch na sua ROM e se mate novamente. Você notará a música de morte tocando novamente. Tudo está certo. Agora para o nosso último passo – remover as moedas após a morte.

Visite o SMWCentral RAM map, e procure por “coin count”. Você vai dar de cara com o $7E0DBF. Para remover todas as moedas do jogador, você precisa definir esse endereço para 00. Inclua isto em nosso patch:

```
ORG $00F60A                    ; Endereço a ser hijacked
autoclean JSL Hijack           ; Pula para o custom code
NOP                            ; Sobrescreve o byte restante com um
                               ; opcode que não faz nada

freecode

Hijack:
STZ $0DBF                      ; Armazena #$00 para $0DBF
LDA #$09
STA $1DFB
RTL                            ; Use RTL depois do JSL
```

Insira o patch novamente e teste. Pegue algumas moedas e se mate novamente. Note que no momento que você morrer, seu contador de moedas se torna 0. Nós atingimos nossa meta agora vamos deixar a meta aberta, e dobrarmos a meta xD – nosso patch está pronto!

-POINT OF ADVICE-

Quando fizer o hijacking, fique de olho nas larguras dos registros A, X e Y. Se você fizer hijack em um código que roda em modo 16-bit A por exemplo, e você começa a escrever código com parâmetros 8-bit (ex. LDA #$03), o jogo irá crashar. Defina a largura do registro para 8-bit no começo, então antes de retornar, defina de volta para modo 16-bit.

Prevenindo crash e testes

Em certo sentido, ao longo de todo o processo nós já medimos esforços em prevenir um crash. Encontramos localizações adequadas de hijacks. Certificamo-nos de lidar com bytes restantes com NOP. Não tem como este patch causar crash em qualquer circunstância. No entanto, você também deve testar efeitos colaterais não intencionais.

E se, por exemplo, você morre como **Luigi**? Ou morre caindo em um buraco? O primeiro você terá certeza sobre, porque o endereço RAM que armazenamos é sobre o **atual jogador** então isso naturalmente inclui Luigi. Mas a morte no abismo é uma morte diferente que não tem animação e etc. Você poderia testar isso.

Apenas um pequeno spoiler. As moedas ainda resetam a 0, porque tanto morrendo por um inimigo ou um abismo, a música de morte sempre toca, então nossa call para o código será sempre executada.

Testar é um passo importante quando você está criando um patch. Pense fora da caixa quando testar. Pense em cenários muito improváveis ainda relevantes e tente.

E é isso! É assim que patches bem simples são escritos. No caso de travamentos e não saber o que os causa, você terá que usar...

Capítulo 9: Debugging e crashes

É aqui onde ASM fica realmente divertido e desafiador. Como você está aprendendo ASM, muitas vezes irá lidar com crashes ou efeitos colaterais. Mais cedo no tutorial nós usamos o snes9x debugger. Um decente e incompleto debugger. Um melhor debugger é o “bsnes plus”. Se você procurar “bsnes-plus” no Google encontrará alguns resultados. O download pela romhacking.net é uma boa opção. Você pode também visitar este link: https://github.com/devinacker/bsnes-plus/releases

“Debugging” é o processo de encontrar e corrigir bugs e erros no código. Se Super Mario World crasha em um certo ponto do jogo depois de aplicar um patch, debugging pode ser usado para encontrar qual patch crasha, onde e por quê. A fim de fazer o debug, você precisa de um conhecimento decente sobre ASM, então este capitulo essencialmente contém um grande salto de dificuldade comparado aos anteriores.

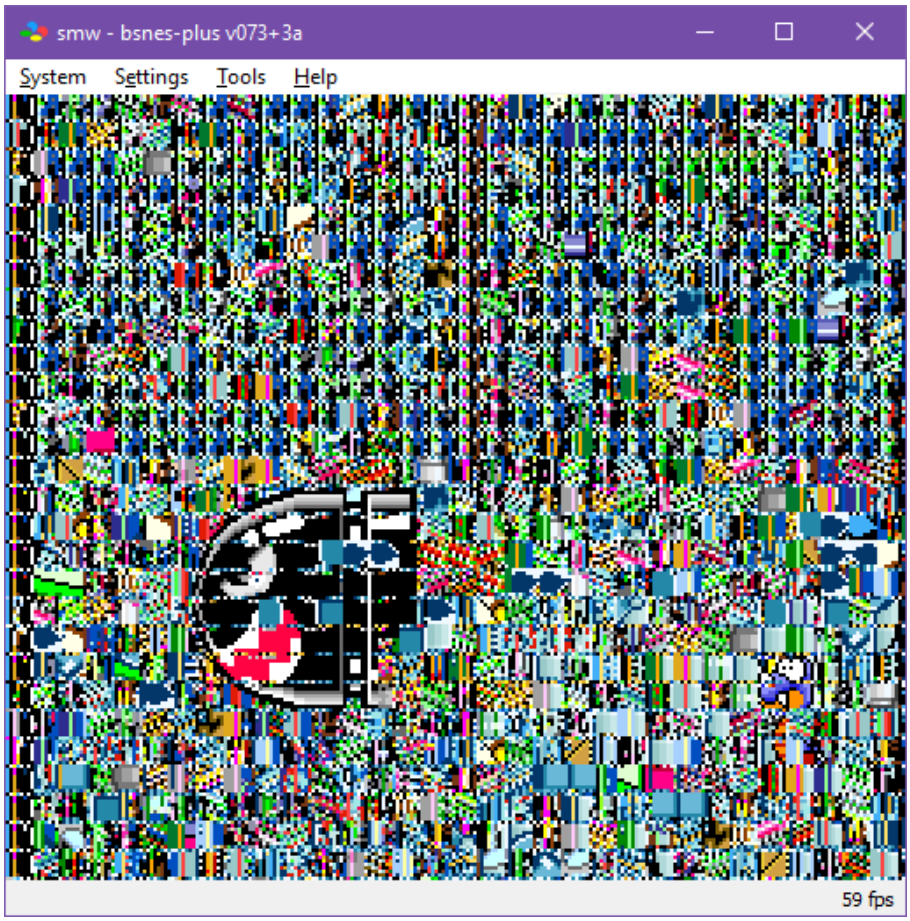
Para começar, aqui está um patch de exemplo que é uma versão levemente modificada do seu do capitulo anterior. Após a morte, o contador de moedas reseta. No entanto, se você tiver qualquer moeda, a cor do background se tornaria vermelho-sangue também (claramente visível em Yoshi’s Island 1). Se você morre com 0 moedas, nada acontece.

```
ORG $00F60A                     ; O endereço a ser hijacked
autoclean JSL Hijack            ; Pula para o nosso código
NOP

freecode

Hijack:
LDA $0DBF                       ; Se zero moedas, pula o próximo bloco
BEQ +                           ; de código.
STZ $0DBF                       ; Zero
REP #$20                        ; 16-bit A para escrever nossa cor
LDA #$001F                      ; 1F é vermelho puro
STA $0701                       ; (Formato: xBBB BBGG GGGR RRRR)
+
LDA #$09
STA $1DFB
RTL                             ;Você sabe o que isso faz, cmon
```

Se você aplicar este patch e morrer sem moedas, nada acontece. Mas se você morrer com moedas, o jogo crasha:



Para acessar o debugger no bsnes-plus, vá em Tools > Debugger ....
Conheça o debug console:

Onde é o crash?

A melhor forma de traçar a origem de um crash é habilitando a primeira opção no bloco Trace, **preferencialmente quando o crash está para ocorrer**. O debugger regista até mesmo códigos recorrentes, fazendo o arquivo de registro facilmente alcançar milhares de MB no tamanho do arquivo em apenas poucos segundos. Será incrivelmente prolixo.

Os registros são armazenados na mesma pasta da ROM. É um arquivo de texto nomeado mais ou menos como "*nomedarom*-trace.log" (ex. smw-trace.log). Este registro contém todas as instruções executadas a partir do momento em que você habilitou o registro da CPU até o ponto do crash e talvez até além. Abra-o no notepad++ ou qualquer coisa que não seja o notepad padrão – Notepad padrão é muito lento quando se trata de mexer com milhares de MB's.

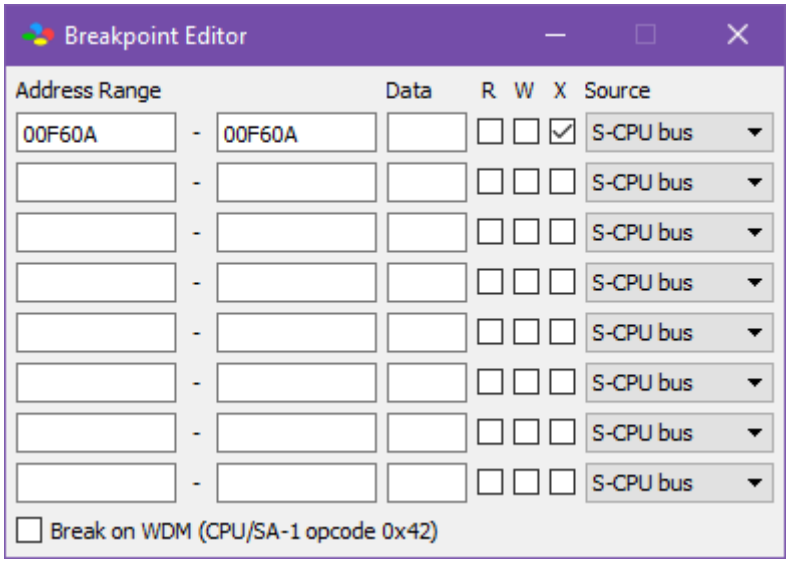
Role para baixo até encontrar alguma coisa suspeita. As vezes também ajuda digitar Ctrl+F e procurar por "BRK", porque 95% das vezes um BRK é envolvido em um crash. E certamente, quando você morre enquanto possui moedas...

```
88692  00f60a jsl $908715   [908715] A:0190 X:0000 Y:0000 S:01e6 D:0000 DB:01 NvMXdizC V:  84 H:324 F:18
88693  908715 lda $0dbf     [010dbf] A:0190 X:0000 Y:0000 S:01e3 D:0000 DB:01 NvMXdizC V:  84 H:339 F:18
88694  908718 beq $8722     [908722] A:0101 X:0000 Y:0000 S:01e3 D:0000 DB:01 nvMXdizC V:  85 H:  7 F:18
88695  90871a rep #$20               A:0101 X:0000 Y:0000 S:01e3 D:0000 DB:01 nvMXdizC V:  85 H: 11 F:18
88696  90871c lda #$001f             A:0101 X:0000 Y:0000 S:01e3 D:0000 DB:01 nvmXdizC V:  85 H: 16 F:18
88697  90871f sta $0701     [010701] A:001f X:0000 Y:0000 S:01e3 D:0000 DB:01 nvmXdizC V:  85 H: 22 F:18
88698  908722 lda #$8d09             A:001f X:0000 Y:0000 S:01e3 D:0000 DB:01 nvmXdizC V:  85 H: 32 F:18
88699  908725 xce                    A:8d09 X:0000 Y:0000 S:01e3 D:0000 DB:01 NvmXdizC V:  85 H: 38 F:18
88700  908726 ora $006b,x   [01006b] A:8d09 X:0000 Y:0000 S:01e3 D:0000 DB:01 Nv1Bdizc V:  85 H: 42 F:18
88701  908729 brk #$00               A:8d69 X:0000 Y:0000 S:01e3 D:0000 DB:01 nv1Bdizc V:  85 H: 50 F:18
88702  0082c3 rti                    A:8d69 X:0000 Y:0000 S:01e0 D:0000 DB:01 nv1BdIzc V:  85 H: 64 F:18
88703  00872b bmi $878d     [00878d] A:8d69 X:0000 Y:0000 S:01e3 D:0000 DB:01 nv1Bdizc V:  85 H: 75 F:18
88704  00872d sta $04       [000004] A:8d69 X:0000 Y:0000 S:01e3 D:0000 DB:01 nv1Bdizc V:  85 H: 79 F:18
```

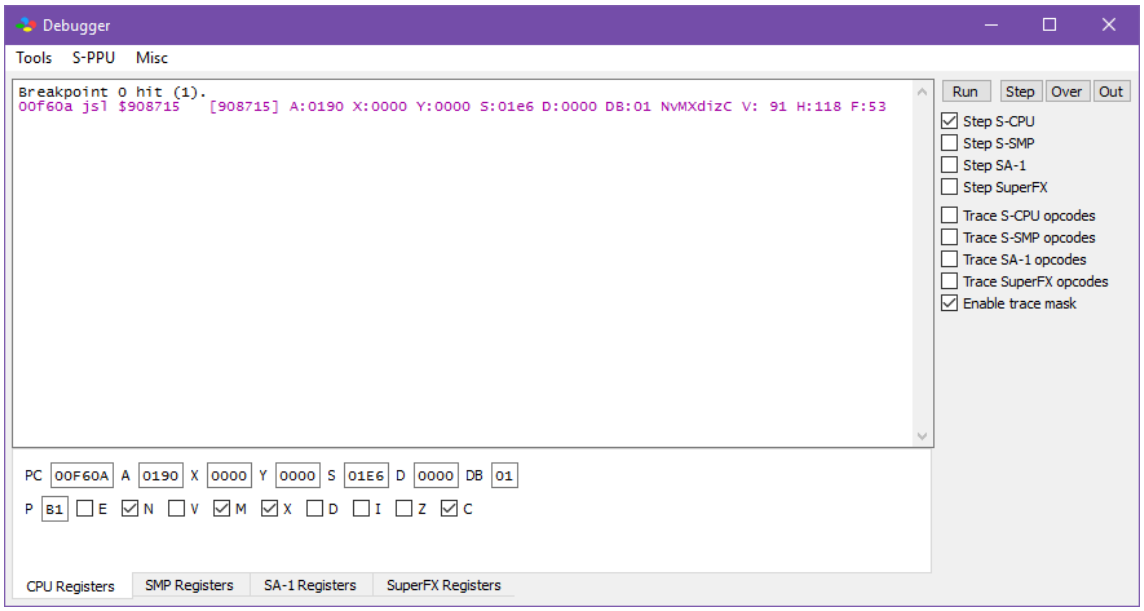
É de fato um BRK.

Breakpoints

Breakpoints definem endereços onde o Debugger deve ‘pausar’, então você pode inspecionar as seguintes instruções após aquele breakpoint peça por peça. Existem 3 tipos de breakpoints: Quebra ao **ler (R), executar (X)** ou **escrever (W)**. A seguinte janela é acessível em Tools > Breakpoint Editor:



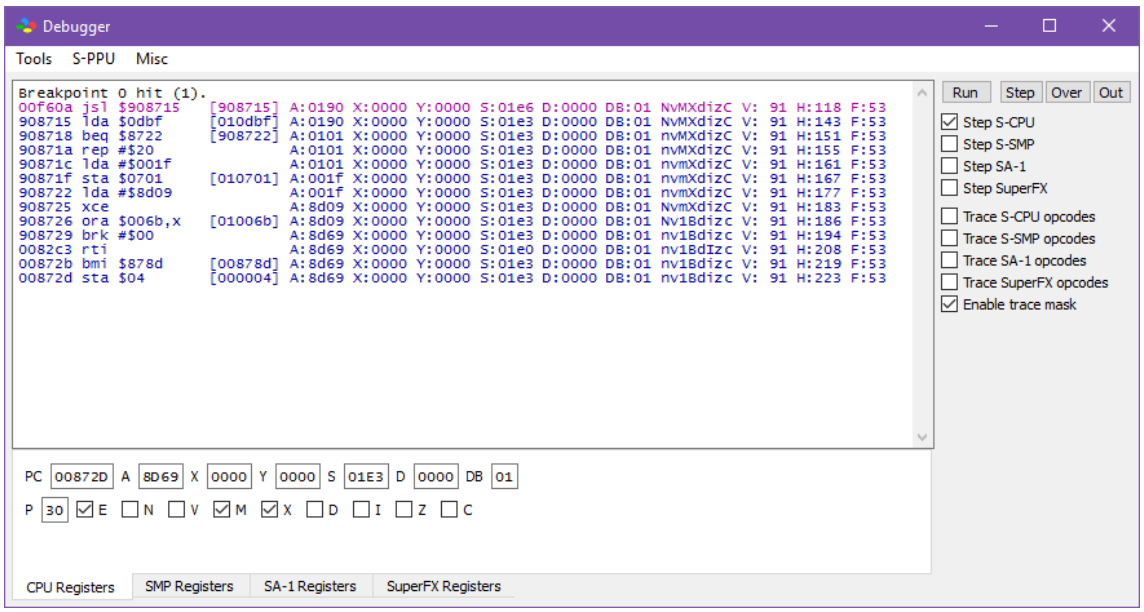
Na imagem acima, eu já inseri a localização do nosso hijack: o debugger irá quebrar em $00F60A. A fonte é S-CPU. É ali que nosso ASM sempre executa. Tanto esse, ou SA-1 (mas este tutorial não cobre SA-1). Defina o breakpoint, rode o jogo (você pode fechar esta janela, é salvo automaticamente) e morra. Veja o que acontece:



Uma linha de código automaticamente apareceu: um JSL. Esse é o nosso hijack!

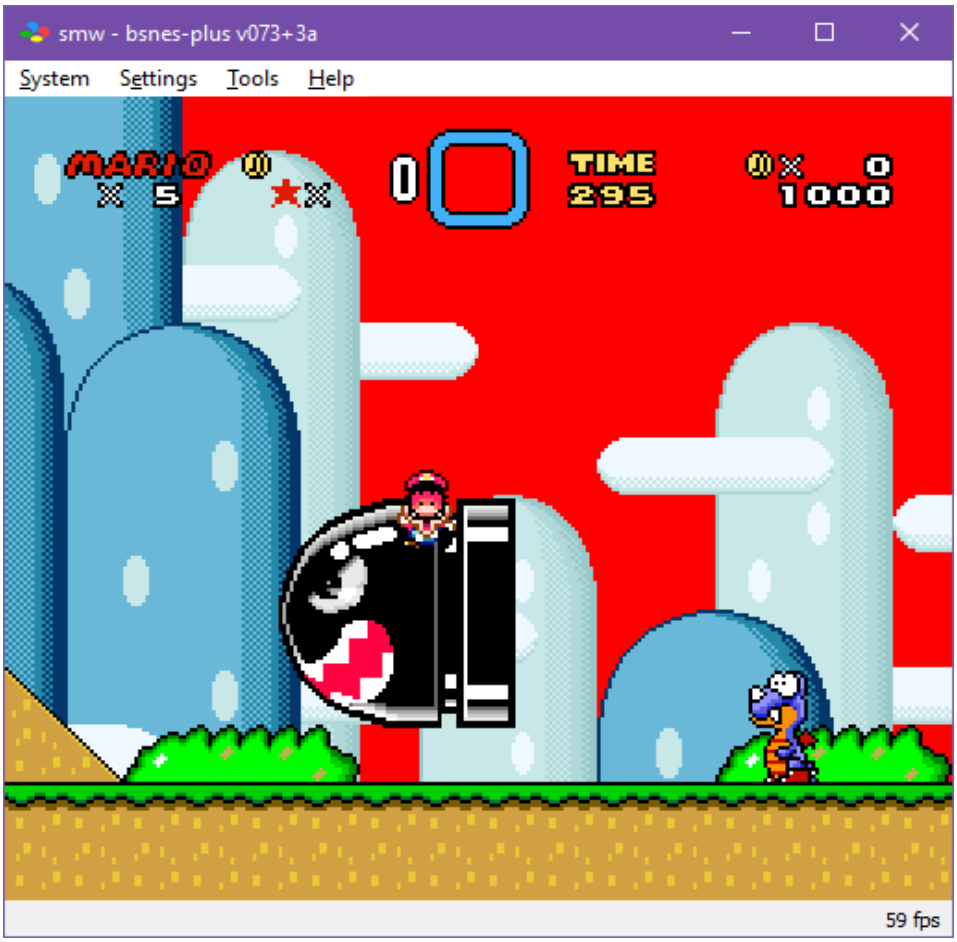
Se você clicar em “Step” (Step Into), executará o próximo opcode. “Over” (Step Over) é um botão similar, mas não faz o branch. Útil se você souber que o branch ou a sub-rotina não está com defeito – isso salva tempo. O código continua executando, você apenas não o vê no registrador. Por fim, “Out” (Step Out) pula para fora da sub-rotina

atual. Mesmo conceito acima, o código continua executando, mas não é registrado. Pressione “Step” algumas vezes para ver o que acontece:



Por quê crasha?

Olhando para o registro de CPU acima, a razão para o crash é logo visível. (para pessoas experientes) – A continua 16-bit quando supostamente deveria ser 8-bit novamente em certo ponto. Portanto, a CPU começou lendo instruções falsas que nunca escrevemos e resultou em um crash. Para consertar isso, um **SEP #$20** é necessário entre **STA $0701** e o sinal de **+**. O resultado:



Existem inúmeras razões pela qual um código pode crashar e o exemplo acima foi a mais simples. Códigos podem crashar nas seguintes situações:

- Tamanho incorreto do registro. 16-bit AXY invés de 8-bit e vice-versa
  - É notável por quase sempre incorrer um BRK no registro também como executar falsos opcodes que você nunca escreveu.

- Um RTS usado para um JSL. Da mesma forma, um RTL usado para um JSR.
  - É notável por executar instruções falsas ou BRK após um retorno no registro.

- Empurrando x quantidade de bytes, mas não puxando-os de volta após um retorno
  - (Isso porque os opcodes de retorno também usam o stack para encontrar o endereço de retorno, e valores não puxados no stack bagunçam com ele).
  - É notável por executar instruções falsas ou BRK após um retorno no registro.

- Um loop infinito
  - É notável por possuir um grande muro de códigos repetidos no registro. Isso também simplesmente congela o jogo, invés de crashar de um jeito espetacular.

Este capitulo cobriu as partes mais importantes do debugging – encontrando crashes e caminhando pelo código. Consertar crashes depende do seu conhecimento de ASM – Não existe uma “fórmula” para consertar crashes.

O debugger tem diversos outros recursos como registrar código SA-1, algumas ferramentas de visualização de PPU, um editor de memória, etc. e custaria um longo tempo explicar todas elas (Isso iria se tornar um completo readme para o debugger). Conforme seu conhecimento de ASM e SNES crescem, você será capaz de entender os recursos restantes.

Capítulo 10: Dúvidas Frequentes sobre ASM

Aprender ASM é uma questão de ler documentos e prática. Porém, contudo, todavia, algumas dúvidas podem surgir sendo elas não documentadas. Isso é especialmente verdade para hacking de SMW, não é como se existissem documentos que abrangem cada dúvida que as pessoas têm.

Este capitulo tem a intenção de preencher esse vazio. As seguintes questões (e respostas) são direto do tópico de ajuda com ASM da SMWCentral. Eu tomei a liberdade de passar por maioria das páginas procurando dúvidas que me fizessem pensar “ei, aquela é uma boa. Pessoas novas em ASM vão questionar sobre aquilo e não está sendo falado neste tutorial e nem no anterior”. Sem mais delongas...

Custom Sprites

Q: Eu quero usar o registro X em meus custom sprites, mas eu ouvi que não posso alterá-lo porque é o atual sprite index ou algo do tipo. E agora?

A: Você na verdade pode modificar o registro X, mas certifique-se de preservá-lo de antemão usando PHX e restaurando com PLX. Como alternativa você poderia também usar “LDX $15E9” que é o atual sprite index.

Custom Blocks

Q: Como eu posso fazer um bloco ter um custom “acts like” configurado programaticamente?

A: Use o seguinte fragmento de opcode onde quiser dentro do custom block:

```
LDA #$30    ; act like block 130 [Cement Block]
STA $1693   ;
LDY #$01    ; this is the high byte
```

Isto pode ser colocado em qualquer lugar que você quiser, enquanto Y permanece inalterado durante o resto da rotina (usando PHY/PLY deve ser útil em ambas situações).

Patch

Q: Como eu posso, por exemplo, fazer a vida de um Chargin’ Chuck variável invés de estática 3-hit baseado no level?

A: Pelo fato de que a comparação é feita na ROM (um CMP #$03), você precisará fazer um hijack na comparação a fim de fazê-lo ler a partir de um endereço RAM invés daquele valor imediato #$03. Para ajudá-lo com conteúdo baseado em levels, use UberASM.

Coding geral

Q: Quando eu tento escrever meu arquivo ASM em uma ferramenta de linha de comando, diz que o arquivo não existe mesmo o arquivo estando bem ali!

A: É provável que as extensões dos seus arquivos estejam escondidas por padrão, e tem uma grande chance do seu arquivo estar na verdade nomeado "filename.**asm.txt**". Procure um tutorial sobre como mostrar extensões de arquivos por padrão. Isso permitirá que você edite as extensões dos seus arquivos renomeando diretamente.

Q: Eu posso fazer múltiplas comparações com um único opcode, ex. "CMP #$03,#$04,#$54"?

A: Não, tal notação de opcode não existe, você tem que fazer as comparações uma por uma.

Q: E sobre múltiplos endereços. Posso fazer algo como isso para checar os múltiplos endereços?

```
LDA $0DB4
LDA $0DB5
LDA $0DBE
CMP #$04
BCC ...
```

A: Não, as instruções LDA vão apenas sobrescrever A repetidamente, no final das contas fazendo apenas CMP checar o valor final dos LDAs.

Q: "Flags" são como um true/false. Onde 0 é false e 1 é true?

A: Sim. Maioria das vezes é na verdade como "**0 = false**" e "**qualquer outra coisa = true**".

Q: ASM pode modificar a ROM?

A: Não, ROM é **read-only memory**, significa que não pode ser modificado por ASM.

Q: Existe alguma forma de fazer um valor hex ser negativo?

A: Sim, em casos onde valores negativos são aceitos (ex. valores de velocidade), valores negativos são entre #$80-#$FF (inclusivo). Maior o valor, menor o número negativo, #$FF seria -1.

Q: Qual a diferença entre RTS/RTL (exceto pela óbvia diferença da letra final)?

A: RTS é usado para retornar de sub-rotinas JSR, enquanto RTL é usado para retornar de sub-rotinas JSL.

Q: Quando eu olhei a lista de opcodes do SNES, encontrei um peculiar: STP. Aparentemente significa: “Stop the clock” e tudo que ele faz é congelar o jogo. Isso deveria ser... inútil?

A: Sim. Normalmente você vai querer ficar longe daquele opcode. Já que tocamos no assunto, fique longe de BRK #$xx, COP #$xx e WDM #$xx também.

Q: Lendo a descrição de BIT, diz que: “BIT realiza AND com o Acumulador e o operante, no entanto, as flags são afetadas, mas o resultado não é armazenado”. O que isso significa?

A: BIT age como um AND, mas os resultados não são armazenados no registro A. No entanto, as flags do processador são afetadas, (tais como a flag Zero).

Q: XBA significa Exchange B with A, mas eu nunca ouvi sobre o registro “B” antes?

A: Sim, registro B é simplesmente o nome do byte alto do acumulador. Este opcode, unicamente, significa que o baixo e o alto byte do acumulador estão trocados.

Q: Eu preciso de um timer em meu ASM para, por exemplo, aumentar as moedas do jogador em 1 a cada 5 segundos. Existe algum timer que eu possa configurar?

A: Sua melhor aposta é usar o frame counter ($7E0014). O frame counter aumenta a cada frame, e 1 segundo é igual a 60 frames. Considerando que o valor máximo de frame counter é $FF, significa que você pode trabalhar com 256 frames (se contar o frame 0 também).

Q: Como eu gero um random number (RNG)?

A: Você pode chamar a sub-rotina em $01ACF9 com um JSL, então leia os resultados da RAM $7E148D e $7E148E.

Q: Como eu posso checar se um certo evento do Overworld é ativado? O endereço da RAM está em um formato estranho!

A: Para salvar tempo, aqui vai um curto bloco de código que você pode utilizar:

```
!EventToCheck = #$34

CheckEvent:
LDA !EventToCheck
PHA
AND #$07
TAX
PLA
LSR
LSR
LSR
TAY
LDA $1F02,Y
AND MaskValues,x
BNE EventSet:
;coisas que ele fará se o evento não foi definido
RTS

EventSet:
;coisas que ele fará se o evento foi definido
RTS

MaskValues: db $80,$40,$20,$10,$08,$04,$02,$01
```

Q: Como eu posso desabilitar os controles do jogador me possibilitando escrever os movimentos do Mario programaticamente?

A: Escreva #$0B em $7E0071. Para habilitar os controles novamente use um STZ em $7E0071.

Q: Como que BEQ e BNE fazem o branch sem um CMP?

A: Isto é relevante para a Flag Zero.

```
LDA #$00
BEQ whatever ;BEQ fará o branch
LDA #$01 ;01 ou maior
BEQ whatever ;BEQ não fará o brach
```

Se você a qualquer momento tiver alguma pergunta não respondida no seu processo de aprendizado, sinta-se livre para contatar-me com uma sugestão para esse FAQ.

Capítulo 11: Considerações finais

No fim, este tutorial lhe ensinou as habilidades básicas para introduzi-lo em ASM hacking de Super Mario World. Se você ficou preso em alguma parte do processo de aprendizagem, nunca hesite em pedir ajuda. Não há nada de errado com isso. Todo mundo tem que iniciar de algum lugar, no fim das contas.

Este tutorial de ASM é uma versão “bem pequena” e especifica para Super Mario World. Eu havia escrito um longo e profundo tutorial de ASM chamado **“Assembly para o SNES”**, também disponível na SMWCentral. Aquele tutorial abrange quase todo tipo de opcode e explica o que acontece por trás das câmeras também enquanto certos opcodes estão rodando. Eu altamente recomendo a leitura assim que você entender o básico de ASM neste tutorial. É sempre empolgante aprender algo novo. O tanto de ASM falado neste tutorial é apenas a ponta do iceberg.

Também, meu profundo tutorial de ASM possui um capítulo com diversos links úteis de vários documentos e websites. Definitivamente dê uma olhada.

Se você tiver qualquer sugestão ou melhoria para este tutorial, tenha a liberdade de me contatar.

Fim do tutorial de ASM.

Autor: Ersanio
Tradução: Insanit
Revisão: YuriDensetsu