

Nintendo Engineering Series @ PRGE 2026

- 10:15 - 10:45: NES Basics
- 10:45 - 11:00: Q&A / Break
- 11:00 - 11:45: Enhancement Chips
- 11:45 - 12:00: Q&A / Break
- 12:00 - 12:45: N64 Reverse Engineering
- 12:45 - 1:00: Q&A



I have been in the games industry since 2007, mostly focusing on graphics and optimization.

I founded Sleepy Bits Games in 2019 with my wife Emily.

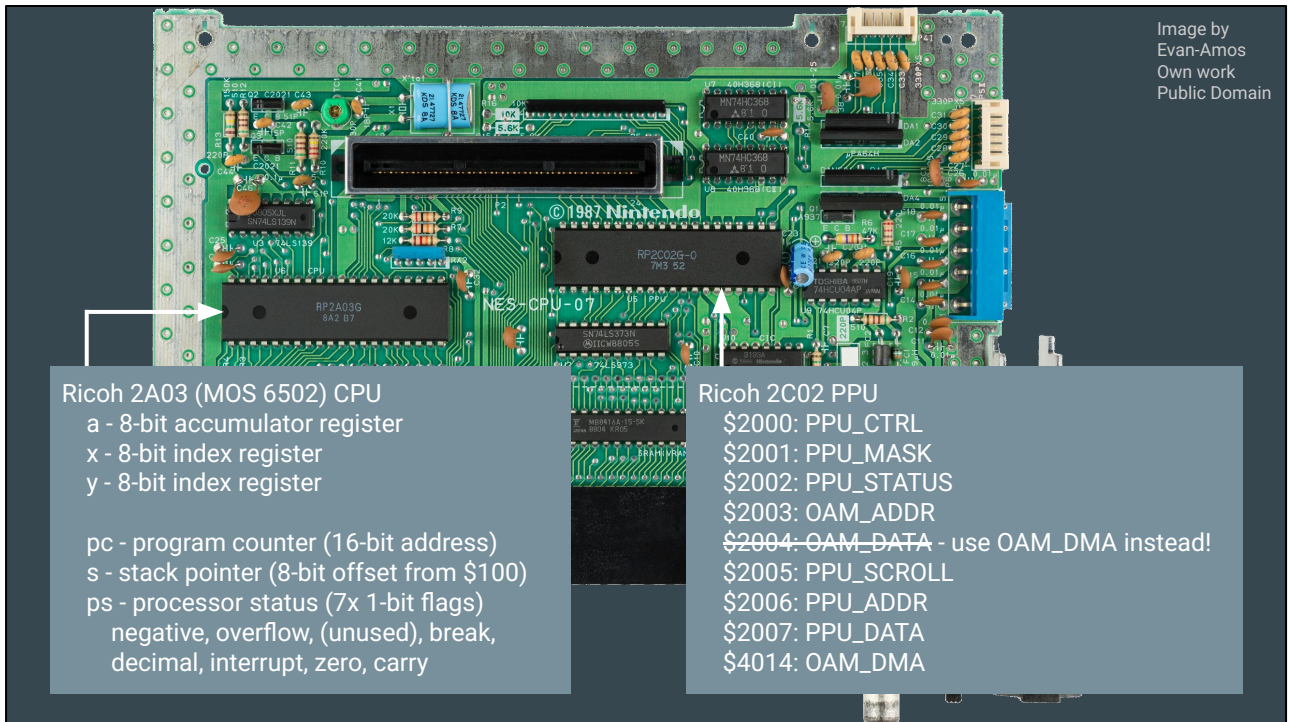
Part 1: NES Basics

Everything in this presentation is available online!

- There's a lot going on this weekend, so don't feel pressured to stay!
- No need to take notes or try to remember every detail.
- All you need is this link:

<https://sleepybits.com/prge2026.html>



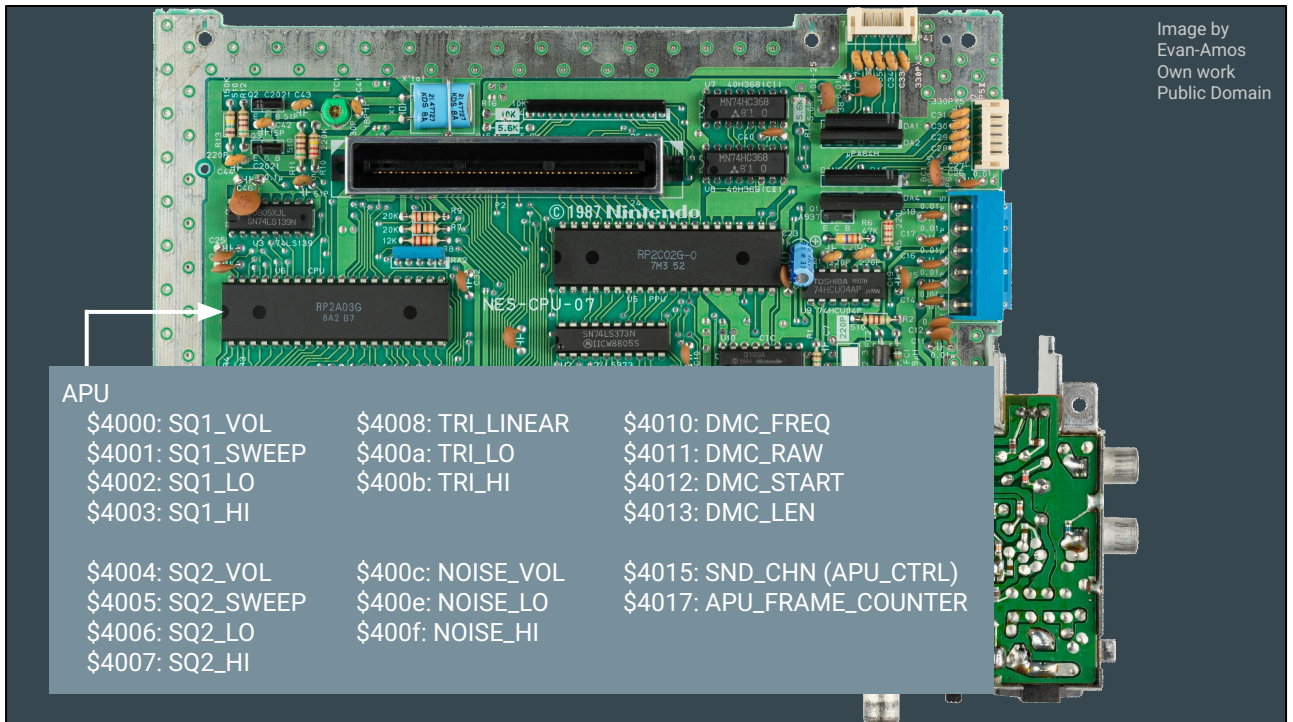


Here are the registers, or how we'll interact with, the NES's CPU and PPU.

Central Processing Unit

Picture Processing Unit (Graphics)

OAM_DATA can corrupt your sprite data! Use OAM_DMA (Direct Memory Access) instead.

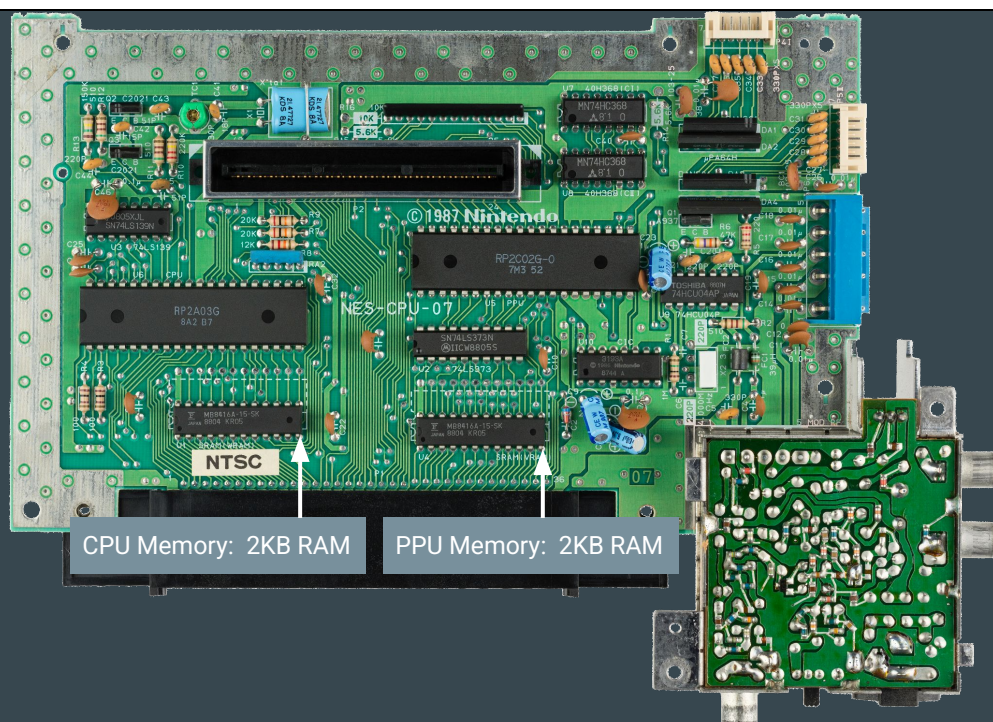


Audio Processing Unit

Wait a minute, isn't that the CPU? Ricoh removed some functionality from the MOS 6502 and replaced it with audio capabilities.

Five sound channels: two pulse or wave channels (beeps and bloops), one triangle wave channel (lower, sustained notes),

the noise channel (explosions, buzzers, percussion), and a Delta Modulation Channel for sample playback.



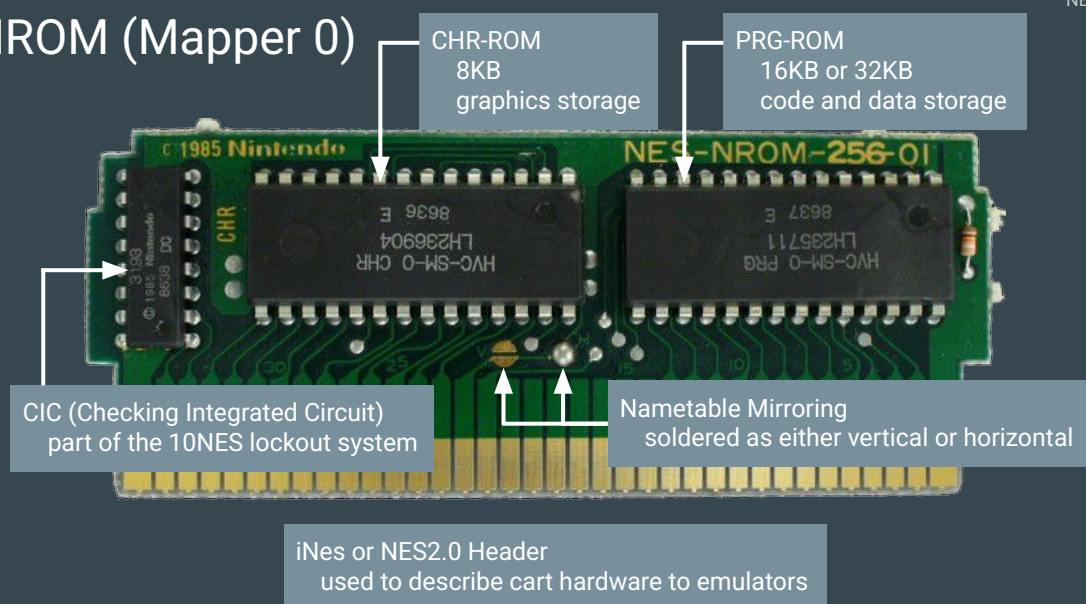
The CPU cannot directly access the PPU's memory.
It must issue commands via the PPU to transfer data.



Controller:
\$4016: JOY_STROBE (write)
\$4016: JOY1 (read)
\$4017: JOY2 (read)

To read button state, the controller must first be “strobed” to fill the buffer.
The buffer is then read one button at a time by the CPU.
Accessories like the zapper and multitap work in the same way, but we won’t be covering them today.

NROM (Mapper 0)



This configuration is used by Balloon Fight, Donkey Kong, Duck Hunt, Excitebike, Ice Climbers, Super Mario Bros. and many more. To encapsulate this information for emulators, the iNes or NES2.0 header is used. This has even been used in official products, like Animal Crossing and the NES Classic!

Program Flow

- reset vector
 - starting point for code execution
 - program counter jumps here during power-on and reset
 - responsible for game initialization
- game loop
 - typically an infinite loop after initialization
 - usually waits for a graphics update before iterating
 - responsible for per-frame game logic updates
 - runs while the PPU is drawing the previous frame's graphics

Program Flow

- NMI vector
 - interrupt signaled at the beginning of Vertical Blank
 - typically graphics are only updated during VBlank
 - runs while the TV prepares for the next frame
- IRQ vector
 - entry point for all other interrupts
 - brk instructions
 - APU DMC playback
 - advanced game cartridge hardware

Typical Memory Map

- Zero Page: \$00 - \$ff
 - fast-access variables
- Stack: \$0100 - \$01ff
 - program flow
- OAM Shadow: \$0200 - \$02ff
 - next sprite update
 - copied via DMA (Direct Memory Access) during VBlank
- Game Specific: \$0300 - \$07ff

OAM - Object Attribute Memory

Case Study: Super Mario Bros.



Palettes

- The NES can produce 54 total unique colors



"Smooth (FBX)" palette definition and image by FirebrandX

The PPU also has emphasis bits for red, green, and blue, to offer full-screen tinting, as well as grayscale.

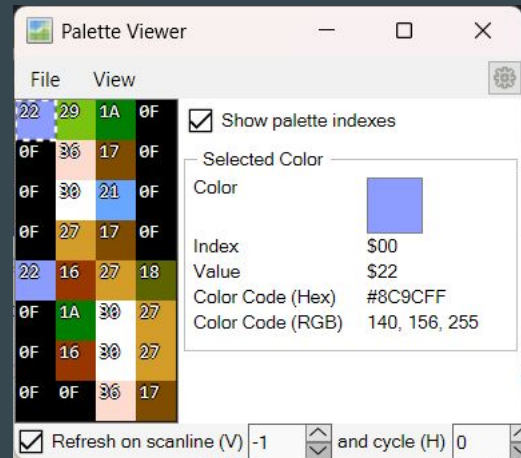
As a side-note, the NES does not define RGB values for its colors!

A ton of work has gone into identifying and replicating its color output for emulators and modern devices.

Embrace the unpredictability!

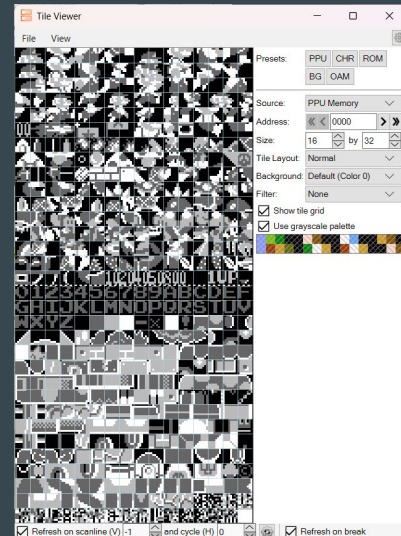
Palette Usage

- Backgrounds
 - 4 palettes of 3 colors + a shared background color
- Sprites
 - 4 palettes of 3 colors + transparency



Pattern Tables

- 1KB array of tiles
- stored as packed indices into a palette
- 2 pattern tables are available to store graphics on NROM

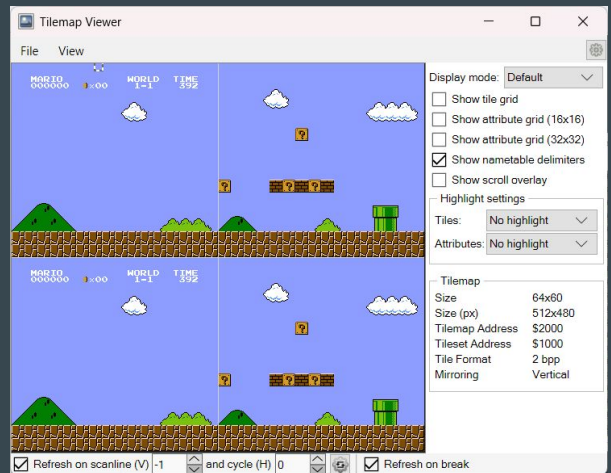


Tile Viewer inside Mesen

Backgrounds and sprites can reference the same or separate pattern tables.

Nametables

- 960 byte array of tile indices
- references the tiles in the background pattern table
- used to layout backgrounds into 32 x 30 screens
- 2 nametables are available to support scrolling on NROM

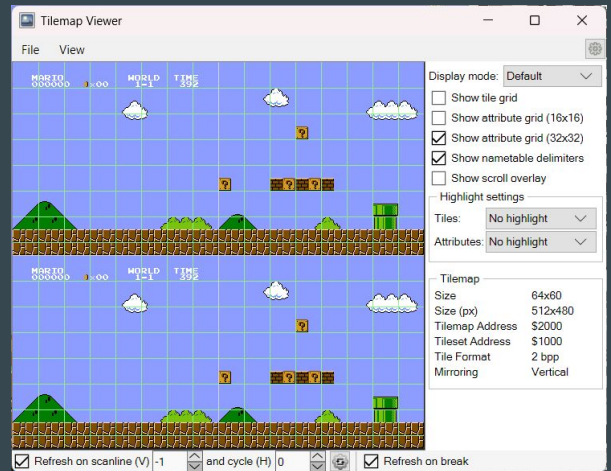


Tilemap Viewer inside Mesen

Vertical mirroring is typically used for horizontal scrolling, such as Super Mario Bros. Horizontal mirroring is typically used for vertical scrolling, such as Ice Climbers. More advanced mappers can support changing the mirroring during gameplay, such as Metroid.

Attribute Tables

- 64 byte array of packed palette indices in an 8 x 8 grid
- used to control a group of 4 x 4 tiles for backgrounds
- palettes can only be specified in groups of 2 x 2 tiles



Tilemap Viewer inside Mesen

This is the cause of those strange screen-edge colors while scrolling in so many NES games!

Sprite OAM (Object Attribute Memory)

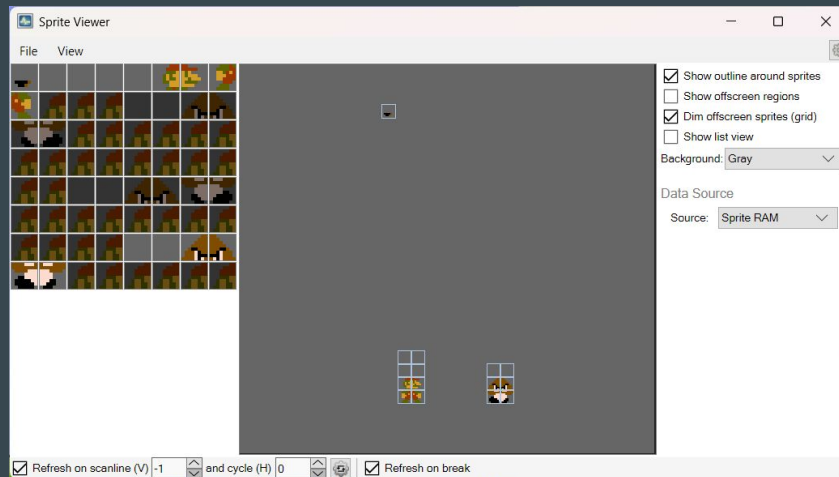
- 256 byte array of data for 64 sprites
 - y position
 - starting from the top of the screen
 - offset by one pixel!
 - tile index
 - references the sprite pattern table
 - attributes
 - palette index
 - priority
 - horizontal and vertical flipping
 - x position
 - starting from the left of the screen

It should be noted that it is impossible to place a sprite beyond the left edge of the screen, or above the second scanline of the screen!

Priority controls whether the sprite appears above or below the background graphics layer.

If more than 8 sprites appear on a single scanline, only the first 8 defined in memory will be drawn. This is why sprite flickering is utilized often on the NES!

Sprite OAM (Object Attribute Memory)



Sprite Viewer inside Mesen

Audio Channel Example: Super Mario Bros. 3



Super Mario Bros. 3 audio exported from Mesen

Let's discuss!

Part 2:
Enhancement Chips:
How Nintendo Cartridges
Leveled Up

Everything in this presentation is available online!

- There's a lot going on this weekend, so don't feel pressured to stay!
- No need to take notes or try to remember every detail.
- All you need is this link:

<https://sleepybits.com/prge2026.html>



What is an enhancement chip?



Resident Evil 2 disc image from psxdatacenter.com
PS2 GS image by Yaca2671 - Own work, Creative Commons

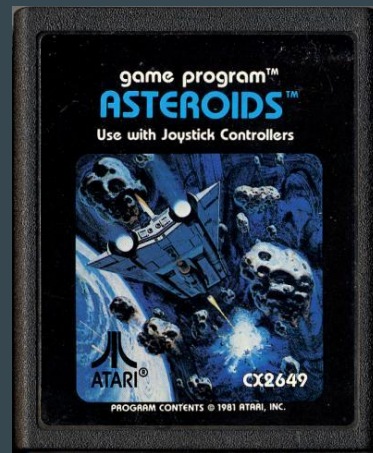
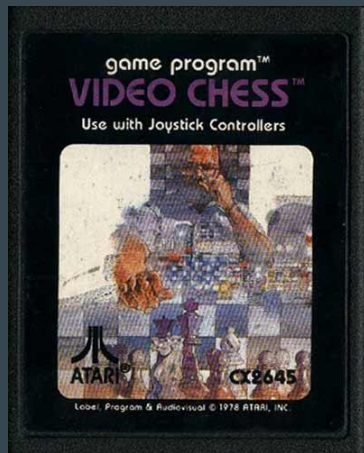
Enhancement chips take advantage of hardware design decisions in cartridge-based consoles in creative ways.

They can provide access to more code, data, and graphics, faster processing, or entirely new functionality.

No longer an option with disc-based consoles nor modern GPU architecture.

For example, no one considered soldering a PS2 graphics chip on Resident Evil 2.

Who was the first to do this?



Cartridge images from atariage.com

The Atari 2600 originally offered 4KB of storage for games. Understandably, it was a challenge to fit a chess program in such a small space. They developed what is called "bank switching" for the logic of Video Chess on the 2600, but ultimately managed to fit the game into 4KB. Later, the port of Asteroids from the arcade to the 2600 was the first cartridge to ship with bank switching.

How does a memory mapper work?



NES console image by Evan-Amos - Own work, Public Domain
NES cartridge image from nescartdb.com

When the PPU, or Picture Processing Unit, of the NES is preparing to draw a scanline of graphics, it must request these from the cartridge.
Super Mario Bros does not use bank switching, so requesting tile ID \$65 always results in a fireball.

How does a memory mapper work?



NES console image by Evan-Amos - Own work, Public Domain
NES cartridge image from nescartdb.com

Super Mario Bros 3, on the other hand, uses bank switching for PRG-ROM (code/data) as well as CHR-ROM (graphics).

First the game logic selects the current bank or section of the graphics data, and then the PPU request happens normally.

In this case, requesting tile ID \$53 results in a fireball, but it could be something completely different if another bank was selected.

The main thing to realize here is that the console and the PPU are unaware that anything more complicated is happening.

NES NROM Capabilities

- PRG-ROM
- CHR-ROM
- Soldered Nametable Configuration
- Background Attribute Limitations
- Sprite 0



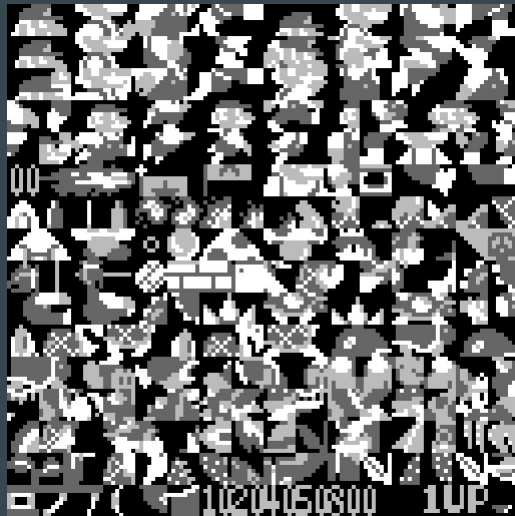
NROM PCB image from nescartdb.com
Sprite Zero Sugar image from coca-cola.com

If you've seen any of our other NES tech talks, you might recall that we've focused on the NROM, or simplest configuration for an NES cartridge.

16 or 32KB for PRG-ROM, 8KB for CHR-ROM. Pre-set nametable configuration. Background attributes, which control the colors of background tiles, can only be applied to every 2x2 set of tiles.

The CPU can actively ask "is Sprite 0 being rendered currently?" for effects like HUDs.

NROM CHR-ROM Example

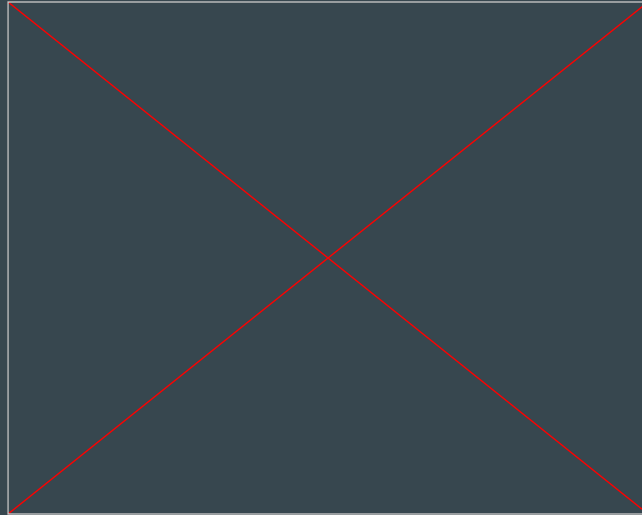


Practically, what does all of that mean?

These are all the graphics for Super Mario Bros without color information. These are always available, no matter what the game is doing.

The 4KB of graphics on the left are for sprites, while the 4KB of graphics on the right are for backgrounds.

NROM Nametable Configuration Example



Super Mario Bros uses what is called the “vertical mirroring” configuration for nametables.

This just means that we have two screens of graphics storage, laid out horizontally, to support horizontal scrolling.

If we scrolled up or down, the graphics of these two screens would be “mirrored” above or below.

NROM Sprite Zero Hit Example



Super Mario Bros also keeps track of whether or not the HUD has been completely drawn by repeatedly asking the PPU about Sprite 0.

The shadow under the coin in the HUD marks when HUD drawing is finished and it is safe to apply the horizontal scrolling for gameplay.

Without this, Mario would leave the HUD behind just like the rest of the level as he runs to the right.

NES MMC1 Capabilities

- Bank Switching (PRG-ROM/CHR-ROM)
- PRG-RAM
- NVPRG-RAM
- CHR-RAM
- Mapper-Controlled Nametable Configuration



MMC1 PCB image from nescartdb.com

MMC1 was how Nintendo brought many features of the Famicom Disk System to cartridge-based games.

It featured larger amounts of PRG and CHR storage with bank switching.

It introduced optional RAM integrated into the cartridge for more memory and/or writable storage that persists after the console has been powered down.

It also allows the nametable configuration to be changed during gameplay instead of permanently set.

MMC1 PRG-RAM Example



MMC1 PCB image from nescartdb.com

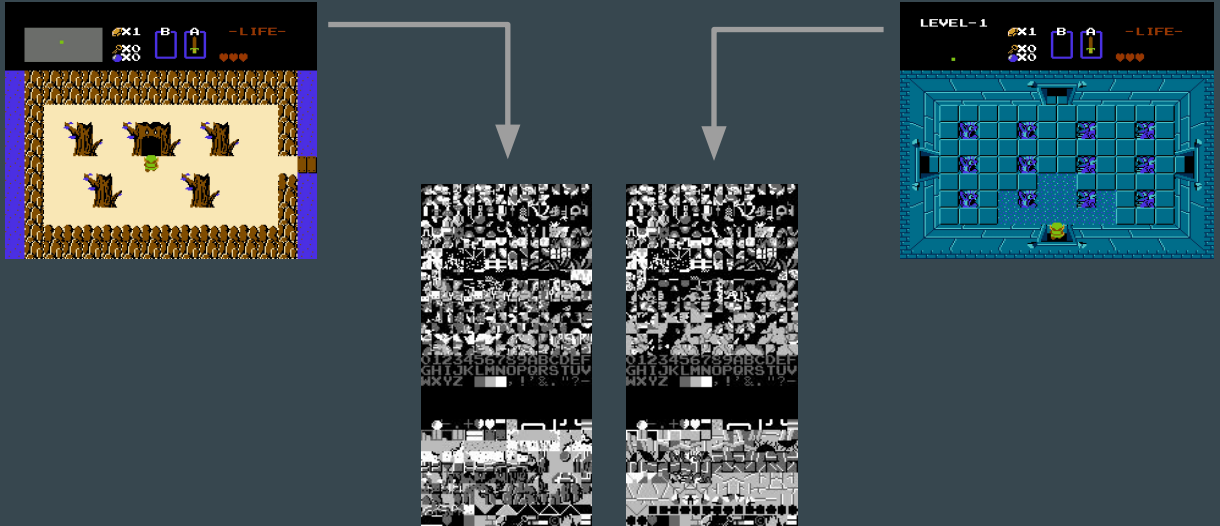


The Legend of Zelda provides 8KB of PRG-RAM (the chip in the top-right) along with a battery (top-left) to maintain save data.

Revision 1 of the game includes this message about holding down the Reset button when powering off the console to protect your save data.

This ensures that no errant writes happen to the save data as the CPU powers down. Later revisions of the MMC1 chip provide a way to enable/disable PRG-RAM to minimize this risk.

MMC1 CHR-RAM Example



All graphics in The Legend of Zelda live in the PRG-ROM as data before being copied into CHR-RAM during gameplay.

This is similar to how graphics data is stored and loaded on modern consoles.

The main benefit to this approach is that it allows per-tile, or even per-pixel updates, to the graphics data based on gameplay.

The main drawback is that these updates are slow and must be done either while graphics are disabled or between video frames.

Traversing between the overworld and the first dungeon results in these changes to the available graphics.

MMC1 Nametable Configuration Example



Here is a very simple example of nametable configuration changing during gameplay in Metroid.

When Samus enters the door, the background scrolls horizontally to reveal the next room.

It then scrolls vertically as she falls to the bottom of the room.

Finally, we scroll back up to the door and then horizontally again through the door to return to the first room.

MMC1 Initialization Code Part 1

```
; reset mapper state
;      xxxxxx - unused
;      1      - reset command
lda #%10000000
sta $8000
```

Communication with enhancement chips typically happens via specific memory-mapped registers.

Addresses \$8000 - \$fffa normally represent read-only PRG-ROM, so writes within that range can be intercepted by the additional cartridge hardware.

Any write with the highest bit set will be interpreted as a reset command, which initializes the MMC1 to a known state.

MMC1 Initialization Code Part 2

```
;          10 - vertical mirroring
;          11 - PRG-ROM switchable at $8000, fixed at $c000
;          1  - 4KB CHR-ROM banks
;          xxx - unused
lda #%00011110

ldx #5
: sta $8000
  lsr
  dex
  bne :-
```

Here we configure the MMC1 with vertical nametable mirroring as well as basic rules for bank switching.

There are many considerations when deciding how to configure your banks, which are out of the scope of this talk, but these examples can be used as a starting place.

We then pass this information to the MMC1 one bit at a time. Since this configuration is always five bits, it takes five writes to do so.

MMC1 Initialization Code Part 3

```
; switch CHR-ROM $0000 to bank 6 (offset: 6 * 4KB = $6000)
;      00110 - bank 6
;      xxx   - unused
lda #%00000110

ldx #5
: sta $a000
  lsr
  dex
  bne :-
```

Finally, we perform one example bank swap to CHR bank 6.
We do this by writing the configuration to a different location, one bit at a time.

NES MMC2 Capabilities

- PPU-based CHR bank switching



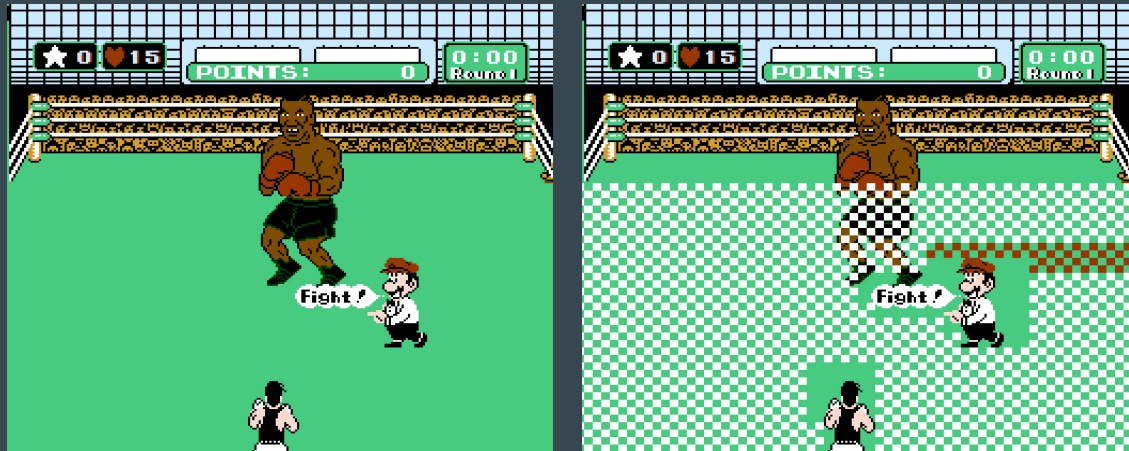
MMC2 PCB image from nescartdb.com

MMC2 allows CHR bank switching to be handled entirely by the PPU and cartridge during the frame.

Recall that so far, we only have the Sprite 0 test to know when we reach a certain point in the video frame.

If we can depend on the PPU and cartridge to handle this, we free up the CPU to perform a considerable amount of work.

MMC2 CHR Bank Switching Example



Punch Out! performs a mid-screen CHR bank switch to first draw the crowd with one bank of graphics, then the referee and Little Mac with another.

Any time a tile ID \$fd or \$fe is requested by the PPU, the cartridge can trigger a bank switch.

Tile ID \$fd has been changed to a checkerboard in the right screenshot so you can visualize when this happens in the frame.

MMC2 Sprite Usage



Unfortunately, Punch Out! also needs to know when to modify the background scroll during the video frame to position Little Mac during gameplay.

Since the CPU is unaware of when the bank switching is happening, it still relies on the Sprite 0 test to time this.

You can see sprite ID 0 well hidden on the right edge of the screen by Mike's elbow.

Punch Out! was the only game to use the MMC2, likely due to the limited advantages it provided.

MMC2 Initialization Code

```
; set up vertical mirroring
;          0 - vertical mirroring
;          xxxxxxx - unused
lda #%00000000
sta $f000

; switch CHR-ROM $0000 to bank 6 after tile $FD is encountered
;          00110 - bank 6 (offset: 6 * 4KB = $6000)
;          xxx   - unused
lda #%00000110
sta $b000
```

One advantage the MMC2 has over the MMC1 is that it no longer receives information one bit at a time, so operations only require one write. Here is how to set up the MMC2 in a similar configuration to our previous MMC1 initialization.

NES MMC3 Capabilities

- Fine-Grain Bank Switching
- Scanline-Based Interrupts
- PRG-RAM for Nametables



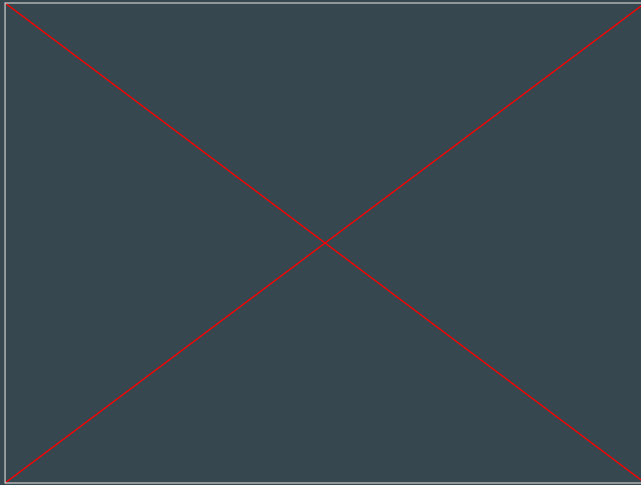
MMC3 PCB image from nescartdb.com

MMC3 drops the ability to bank swap based on PPU tile requests, but allows for more fine-grain control of bank size and location.

It also provides scheduled CPU interrupts at specific scanlines, so testing for Sprite 0 hits are typically no longer necessary.

Finally, MMC3 supports hard-wired nametable layouts for single-screen or four-screen layouts. We'll see an example of this later.

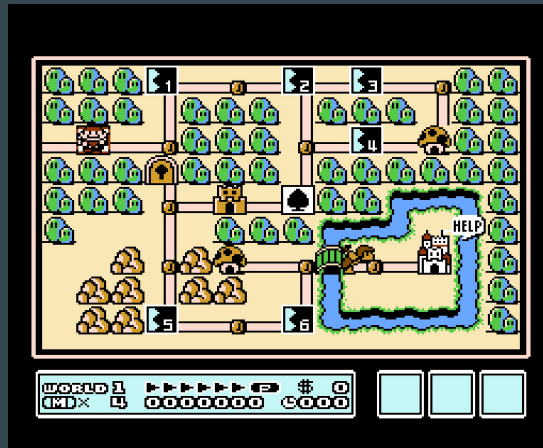
MMC3 CHR Bank Switching Example



On the World 1 map screen of Super Mario Bros 3, the dancing bushes represent too many background tiles to update per frame.

This animation is accomplished by switching the graphics bank every so often so all the bushes update in unison.

MMC3 Scanline IRQ Example



The HUD in Super Mario Bros 3 is controlled by a scheduled interrupt on scanline 192 (of 240), towards the bottom of the screen.

HUDs on the bottom of the screen became more common around this time, since the CPU could be running gameplay logic instead of testing for Sprite 0.

Just as a note, due to the complexity of updates during a video frame, a small visual artifact can be seen during gameplay at this location.

Displaced Gamers has a great YouTube video covering the internal PPU registers that cause this.

MMC3 Initialization Code Part 1

```
; set up vertical mirroring
;          0 - vertical mirroring
;          xxxxxxx - unused
lda #%00000000
sta $a000
```

This is similar to the MMC2 mirroring command, but it is written to a different location.

MMC3 Initialization Code Part 2

```
; switch CHR-ROM $0000 to bank 6
;      000 - update 2KB CHR-ROM bank at $0000
;      xxx - unused
;      1   - PRG-ROM switchable at $8000, fixed at $c000
;      0   - 2x 2KB CHR-ROM banks at $0000
lda #%01000000
sta $8000

;      00000110 - bank 6 (offset: 6 * 1KB = $1800)
lda #%00000110
sta $8001
```

Notice the increased configurability of the MMC3 banks.

This state information must be provided for each bank switch, and is typically tracked in CPU variables by the game.

NES MMC5 Capabilities

- Custom Nametable Configurations
- Fine-Grain Background Attributes
- Vertical Split-Screen
- 8-bit Multiplication!



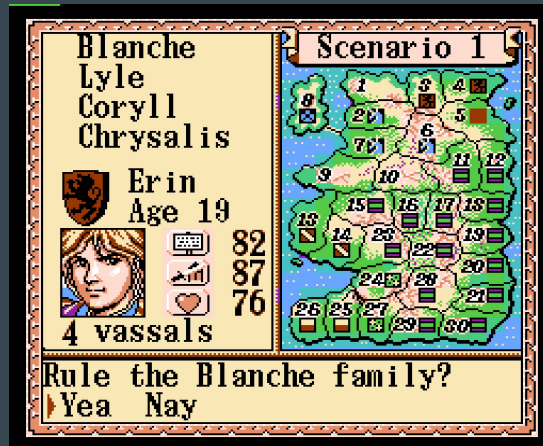
MMC5 PCB image from nescartdb.com

The MMC5 allows full control over nametable layouts, as well as optional support for additional nametable RAM on the cartridge.

It can also supply additional RAM to specify background attributes (colors) per tile. Very few titles took advantage of vertical split screen, which enables independent scrolling per screen segment.

Finally, we have access to 8-bit (0-255) multiplication in hardware! Previously we would need to depend on binary shifts or slow software implementations.

MMC5 Background Attributes Example



In Gemfire, you can see the map terrain and flags frequently combine to break the traditional 16x16 background color rules on the NES.

MMC5 Initialization Code Part 1

```
; set up 2KB CHR-ROM banks
;          10 - 2KB CHR-ROM banks
;          xxxxxx - unused
lda #%00000010
sta $5101

; disable internal extended RAM usage for nametable
;          11 - $5c00 - $5fff read only,
;          $2000 - $2fff not available for nametable or attributes
;          xxxxxx - unused
lda #%00000011
sta $5104
```

In this example, we configure the CHR-ROM banks.
Then we disable all cartridge-based RAM for nametables.

MMC5 Initialization Code Part 2

```
; set up vertical mirroring
;      00 - nametable at $2000 mapped to page 0
;      01 - nametable at $2400 mapped to page 1
;      00 - nametable at $2800 mapped to page 0
;      01 - nametable at $2c00 mapped to page 1
lda #%01000100
sta $5105

; switch CHR-ROM $0000 to bank 6
;      00000110 - bank 6 (offset: 6 * 2KB = $3000)
lda #%00000110
sta $5121
```

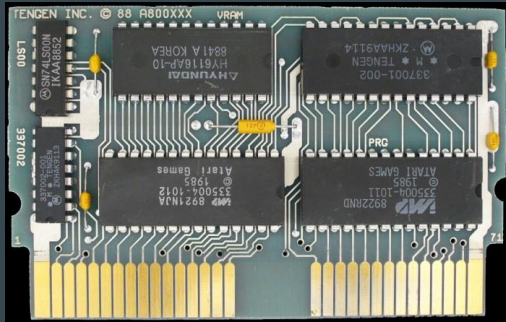
Because of the increased configurability of nametables on the MMC5, we have to explicitly define the RAM used for each.

Finally, we switch to CHR bank 6.

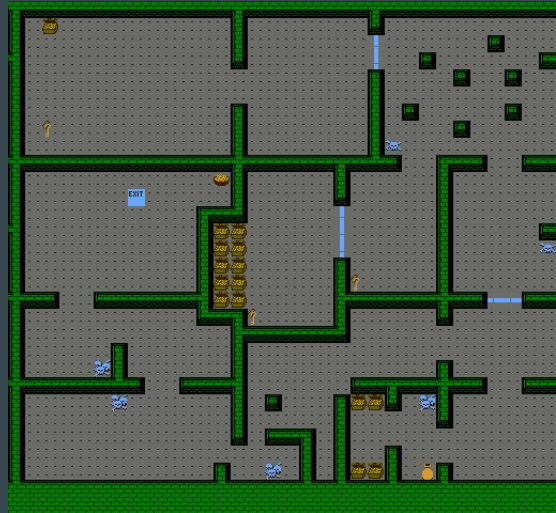
One last thing to note here is that MMC5 takes into consideration the current CHR bank size when selecting a bank.

Different mappers have different behaviors for this, so be careful when laying out your graphics banks!

Other Unique Enhancement Chips: MIMIC-1

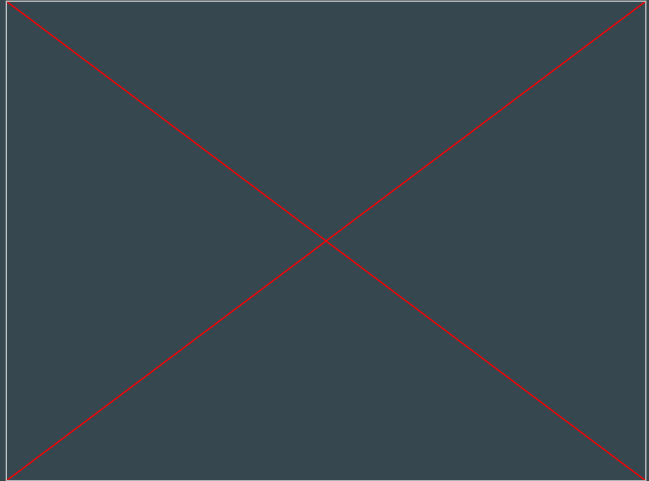
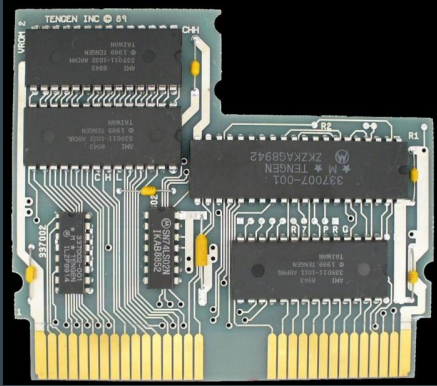


MIMIC-1 PCB image from nescartdb.com



As you may know, Tengen was Atari's line of unlicensed NES cartridges. When porting Gauntlet to the NES, Atari used the MIMIC-1, a stripped down MMC3 clone, to lay out entire levels across four nametables. When Gauntlet was later released as a licensed NES cartridge, they switched to the MMC3!

Other Unique Enhancement Chips: Sunsoft-4



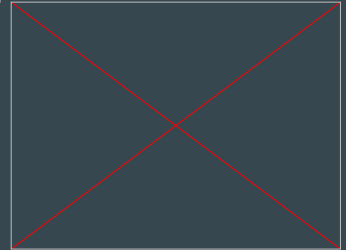
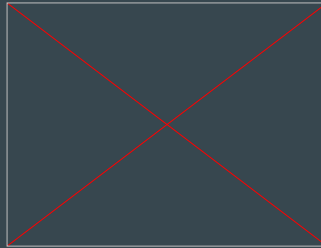
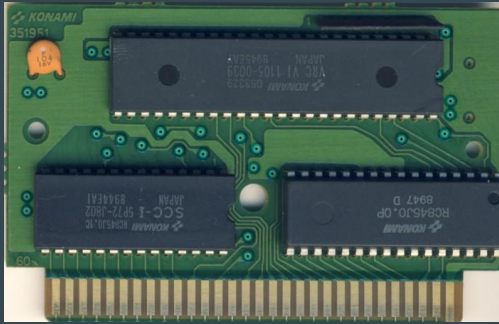
Sunsoft-4 PCB image from nescartdb.com

After Burner is another example of a Tengen cartridge using an enhancement chip, the Sunsoft-4.

This cartridge supports using segments of the ROM as a static nametable.

These static layouts are swapped based on time and the roll of the plane to enable fullscreen updates that would otherwise be too slow for single-frame updates.

Other Unique Enhancement Chips: VRC-6



VRC-6 PCB image from nescartdb.com

One advantage the Famicom console has over the NES is the ability to receive additional audio channels from the cartridge hardware.

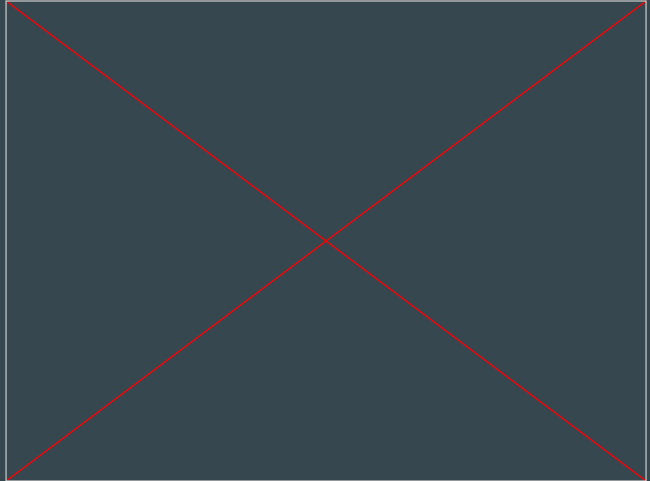
In Japan, Castlevania III, or Akumajō Densetsu, used Konami's VRC-6 enhancement chip to access two additional pulse channels and a sawtooth channel.

Let's compare a few audio tracks from each to get an idea of what that means practically.

Other Unique Enhancement Chips: VRC-7



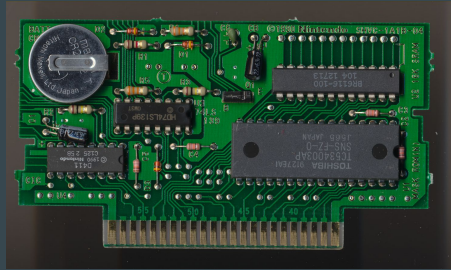
VRC-7 PCB image from nescartdb.com



Lagrange Point is a really impressive RPG that features the VRC-7. The VRC-7 included six FM synthesis channels to create complex audio playback. This allows the game to sound more like a Genesis game than an NES game!

Onwards, to the
fourth console generation!

SNES Baseline Example: F-Zero

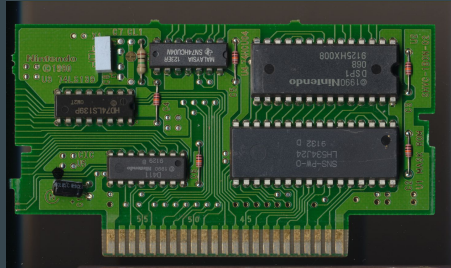


PCB image from archive.org/details/near-snes-scans-png
Thank you for everything, Near. ❤️

F-Zero absolutely blew my mind upon release, and it continues to impress when you realize it only used the base SNES hardware.

The Mode 7 perspective effect, as well as the placement and keyframe-based rotation of the other vehicles are all done by the console itself.

SNES DSP Example: Pilotwings

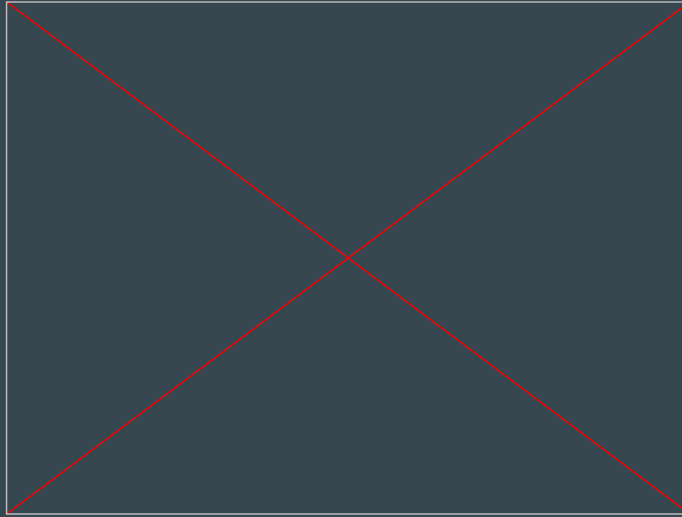


PCB image from archive.org/details/near-snes-scans-png

Pilotwings, another launch title, needed to perform quite a bit more math for player control and object placement.

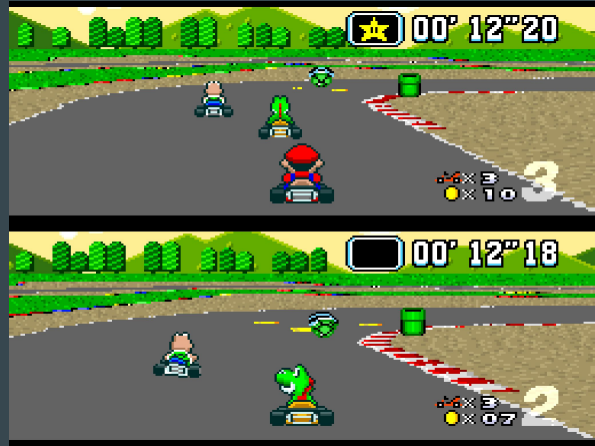
It accomplished this by leveraging the DSP-1 chip (top right), which could perform these operations while the console CPU did other work.

SNES DSP-1 vs DSP-1B Example: Pilotwings



Two revisions of the DSP-1 chip were later released, the latter of which made a “small” change to the implementation of calculations. A later run of Pilotwings cartridges included the DSP-1B chip without accounting for this change, which can actually be seen in the game’s attract mode.

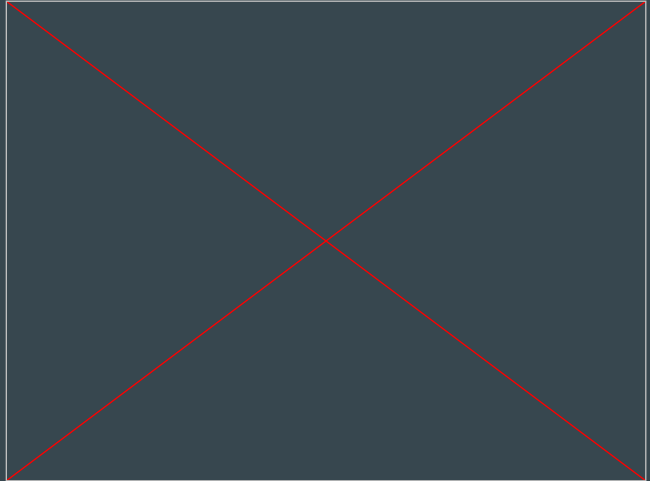
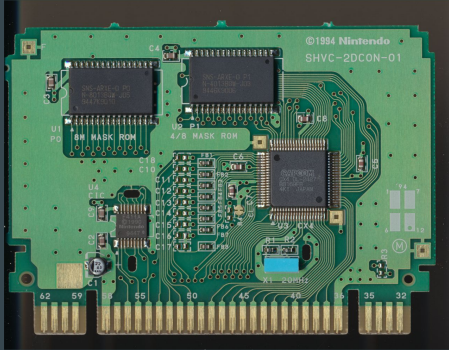
SNES DSP Example: Super Mario Kart



PCB image from archive.org/details/near-snes-scans-png

Similarly to Pilotwings, Super Mario Kart leveraged the DSP-1 to ease the burden of object placement and keyframe-based rotation, especially in split-screen.

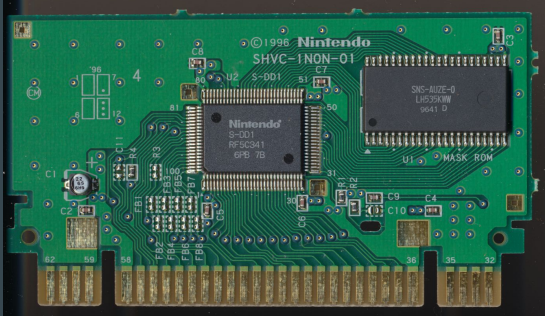
SNES CX4 Example: Mega Man X2



PCB image from archive.org/details/near-snes-scans-png

Capcom created the CX4 chip to produce very futuristic wireframe models with true scaling and rotation, as seen in Mega Man X2 and X3.

SNES S-DD1 Example: Street Fighter Alpha 2



PCB image from archive.org/details/near-snes-scans-png

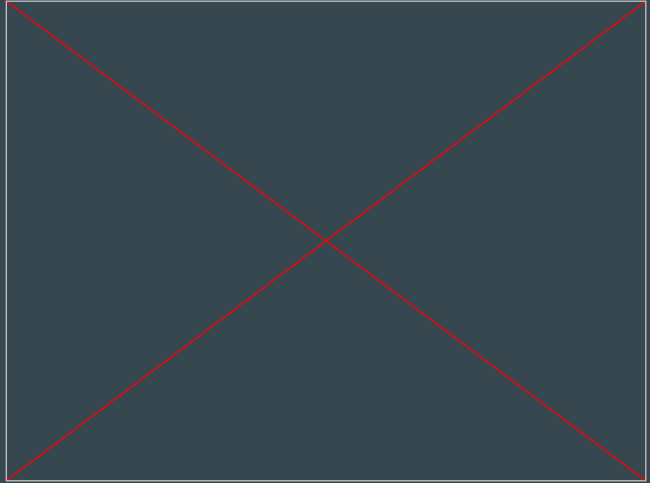
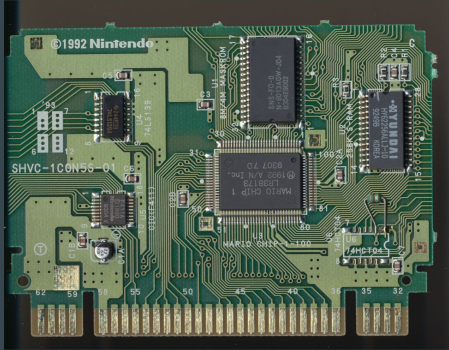
The S-DD1 provides image decompression in order to get the most out of the SNES's storage limit of 4MB.

It is based on the concept run-length encoding, which takes advantage of lines of similar pixels typical for sprites.

Street Fighter Alpha 2 used this to fit all the animations required for all of its characters, at the expense of big delays before fights can start.

Keep in mind, this is a 450MB game on the Sega Saturn!

SNES Super FX Example: Star Fox



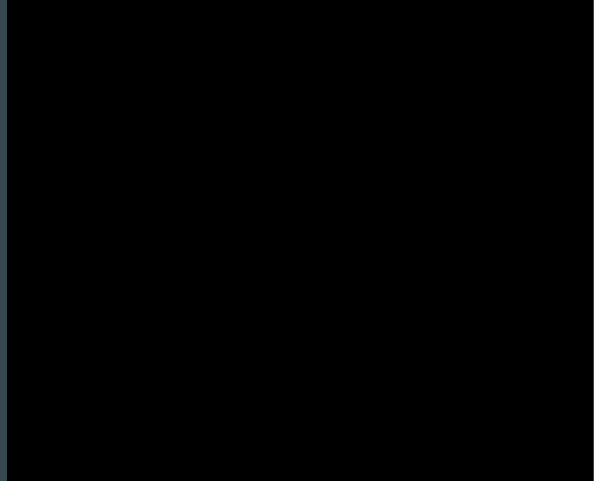
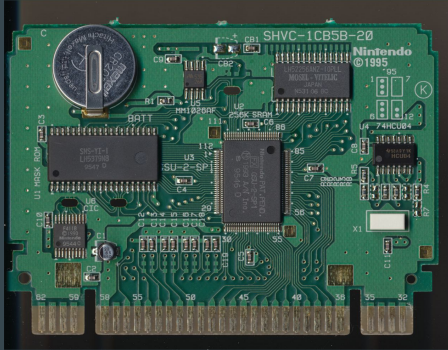
PCB image from archive.org/details/near-snes-scans-png

Perhaps the most famous enhancement chip, the Super FX, developed by the technical wizards at Argonaut Games, is what powered the polygons of Star Fox and others.

The graphics are rendered completely on cart and then transferred to the SNES for presentation.

This process took three video frames, which is why the game runs at 20 FPS.

SNES Super FX 2 Example: Yoshi's Island



PCB image from archive.org/details/near-snes-scans-png

The next iteration, the Super FX 2, runs as twice the speed of the Super FX and can access a larger range of RAM and ROM.

Yoshi's Island 2 uses the Super FX 2 not for polygons, but to scale and rotate very large sprites to great effect.

Here you can see the title screen manipulating a significant number of sprites as we float around the island.

SNES Super FX 3 Example: Doom (LRG, 2026)



PCB image from raspberrypi.com

Limited Run Games' Doom for SNES uses a Raspberry Pi RP2350B running at 150Mhz to emulate the Super FX chip and interface..

SNES SA1 Summary



PCB image from archive.org/details/near-snes-scans-png

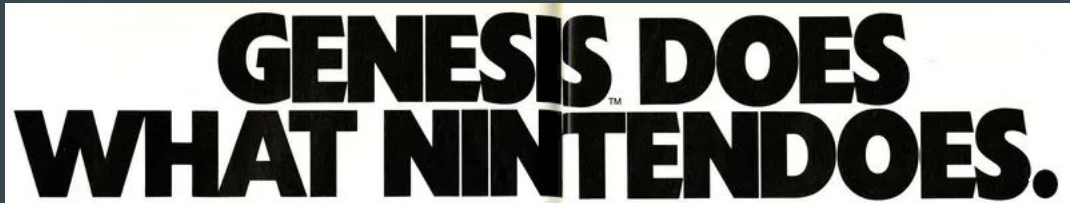


The SA1 featured a CPU very similar to the SNES's main CPU, but it was more than three times faster.

These processors would communicate via interrupts and work in parallel to speed up all sorts of general processing tasks.

The SA1 was used by Super Mario RPG, Kirby Super Star, and Kirby's Dream Land 3, among others.

It is also a favorite among SNES rom hackers who want to find ways of reducing slowdown in other games.

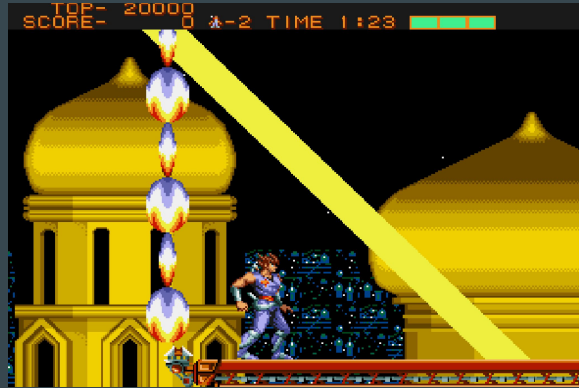
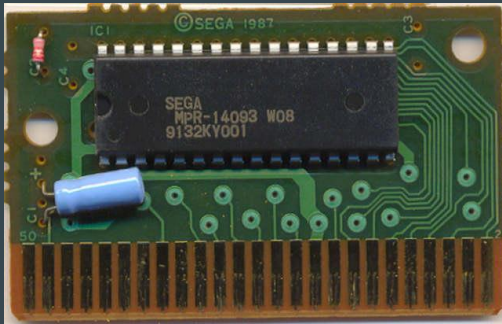


**GENESIS DOES
WHAT NINTENDOS.**

Magazine ad image from <https://sega.fandom.com>
Modified and used under Fair Use.

I know the title of this talk only includes the word “Nintendo,” but we have to shout out a huge effort by Sega as well.

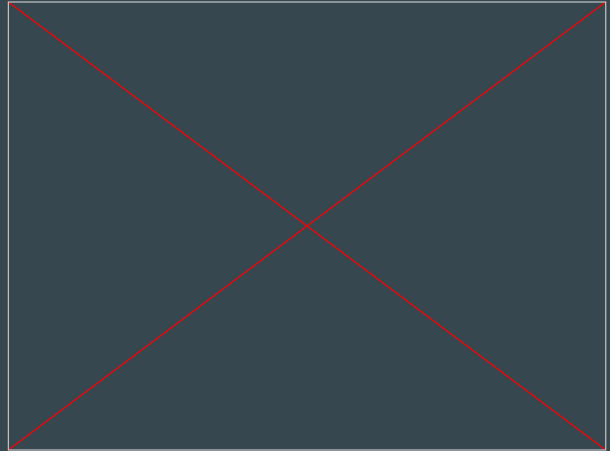
Genesis Baseline Example: Strider



PCB image from <https://gendev.spritesmind.net/mirrors/www.nepereny.com>

Here is an example of baseline Genesis cartridge hardware, in this case Strider.

Genesis SVP Example: Virtua Racing



PCB image from <https://gendev.spritesmind.net/mirrors/www.nepereny.com>

This is the cartridge hardware of Virtua Racing, featuring the Sega Virtua Processor (center).

This was Sega's answer to the Super FX chip. Let's take a look at the results.

Due to high cartridge prices and other hardware initiatives like the 32X, this was the only game to ever release with the SVP.

Let's discuss!

Part 3:
Intro to N64 Reverse
Engineering & Randomizers

Everything in this presentation is available online!

- There's a lot going on this weekend, so don't feel pressured to stay!
- No need to take notes or try to remember every detail.
- All you need is this link:

<https://sleepybits.com/prge2026.html>



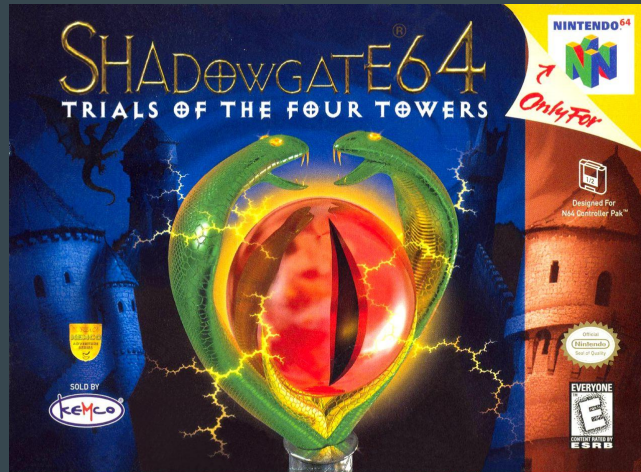
What is a randomizer?

- Randomizers are a great way to breathe new life into your favorite video games.
- Pokémon, Mario 64, Ocarina of Time, Resident Evil, Dark Souls, and many, many other games have amazing randomizers.
- Resident Evil 2 for the N64 actually had a built-in randomizer in 1999!



Randomizers are a great way to breathe new life into your favorite video games by shuffling items, enemies, and/or doors while maintaining gameplay logic so that the game can still be completed.

Introducing Shadowgate 64: Trials of the Four Towers



Shadowgate 64 cover art from thecoverproject.net

Shadowgate 64 was the 1999 sequel to Shadowgate, the 1987 classic, and the 1993 Beyond Shadowgate for the TurboGrafx CD, which no one played. Confusingly, there's also a Beyond Shadowgate from 2024, which is a sequel to the 1987 game.

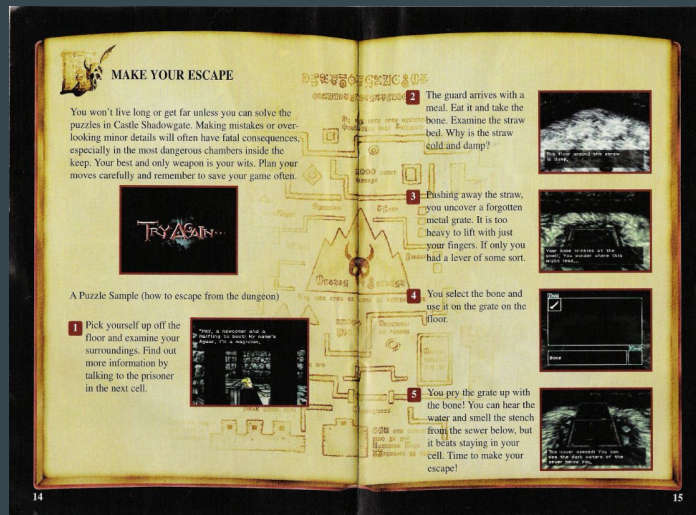
If you're interested in getting into the franchise, the original is available on most vintage computers, NES, Gameboy Color, and even the Playdate.

A remake was released for PC and modern consoles in 2014, but today we're going to be discussing the N64 exclusive, Shadowgate 64.

It is a 3D, first person puzzle and adventure game with a fantasy setting.

The events of the original are mentioned throughout the game, but prior knowledge isn't strictly required.

Spoiler alert!



Manual scan from archive.org/details/N64Manuals

With a game like Shadowgate, spoilers can especially compromise the experience, so we'll try to keep it as light as possible.

That said, even the manual for Shadowgate 64 ruined the first few puzzles!

How does the game track item status?

	1	in inventory
	3	not in inventory, placed in world
	4	not in inventory
	22	in inventory, equipped

The game maintains two separate lists in memory. One is a fixed-sized array that stores a “status” for each item at its ID’s position. This status is updated when an item is picked up, used, placed in the world, or equipped.

How does the game track item status?



The other is a variable-sized array that stores the ID's of the current inventory, based on pick-up order.

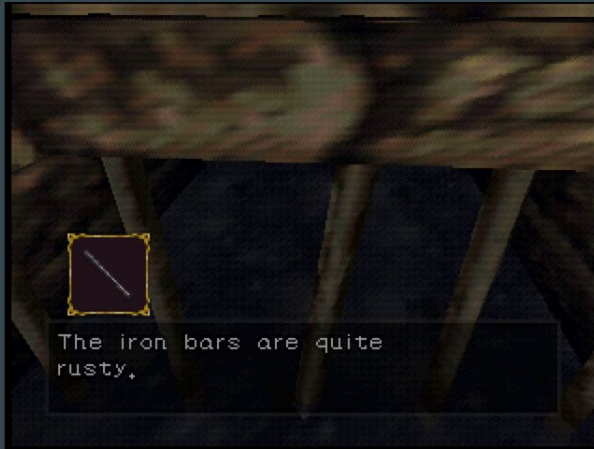
Items in this list are added to the first position when picked up and removed when used.

Look-Up Table Concept

	1	→		3
	3	→		4
	4	→		22
	22	→		1

The core concept of this randomizer is a look-up table that maps the original ID's to shuffled ID's.

Player Experience Examples

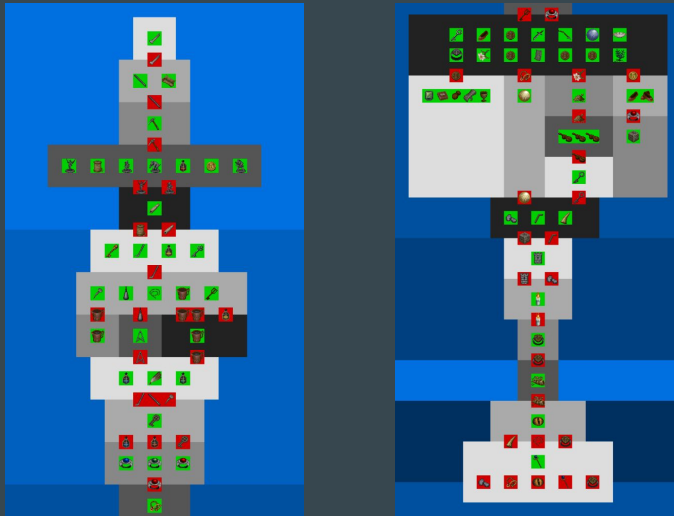


This randomizer doesn't affect the 3D game world. Doing so would require much more knowledge of the game data.

Instead, the icons are now our source of truth while playing.

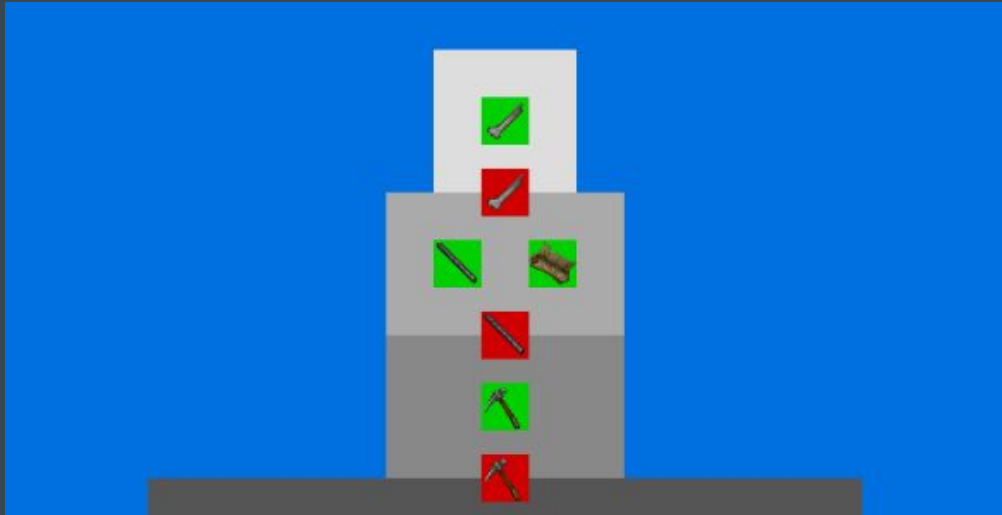
Some items will appear in their original locations, like the Iron Bar on the left, and others will be completely different, like the Dragon Flute on the right.

Game Design: Dependency Graph



In order to visualize the game logic, we can create a dependency graph. Here is every item available in Shadowgate 64, along with which items gate progress. When there are multiple paths available to the player, the gameplay becomes more complex, and the dependency graph becomes wider in those areas.

Game Design: Dependency Graph



Fortunately, this is the dependency graph for escaping the dungeon and sewers at the beginning of the game.

Typically you want your game design to remain “narrow” while introducing players to your game.

However, these narrow segments also limit our ability to randomize the experience.

Code: Dependency Graph

```
typedef enum {  
    MAP                = 0x2a,  
    BONE               = 0x2b,  
    STAR_CREST        = 0x2c,  
    IRON_BAR           = 0x2d,  
}
```

```
#define ITEMS_DUNGEON BONE  
#define ITEMS_SEWERS_1 MAP, IRON_BAR
```

```
{  
    BONE,  
    {  
        ITEMS_DUNGEON,  
    },  
},
```

```
{  
    MAP,  
    {  
        EVERYTHING,  
    },  
},
```

```
{  
    IRON_BAR,  
    {  
        ITEMS_DUNGEON,  
        ITEMS_SEWERS_1,  
    },  
},
```

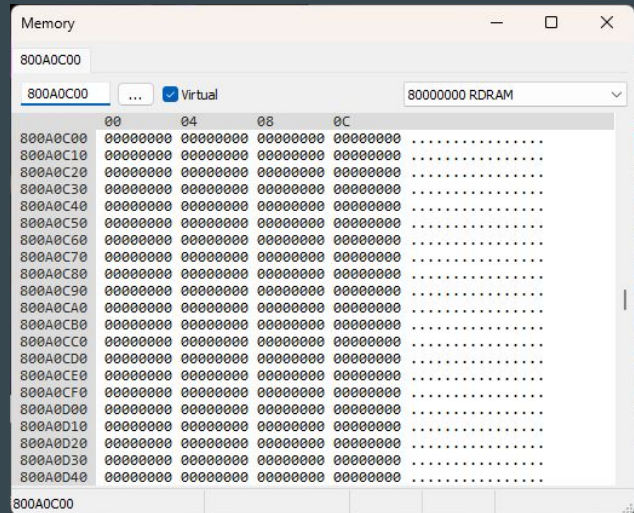
Here are some examples of how I represent the items, segments, and possible destinations for items when shuffling.

Initially, the player only has access to the Bone, so the Bone MUST stay in its original location.

The Map, while useful, is not required to beat the game, so this item may be shuffled to any location.

The Iron Bar is required to exit the first part of the sewers, so it must be placed in that area.

Finding Available Space in the ROM

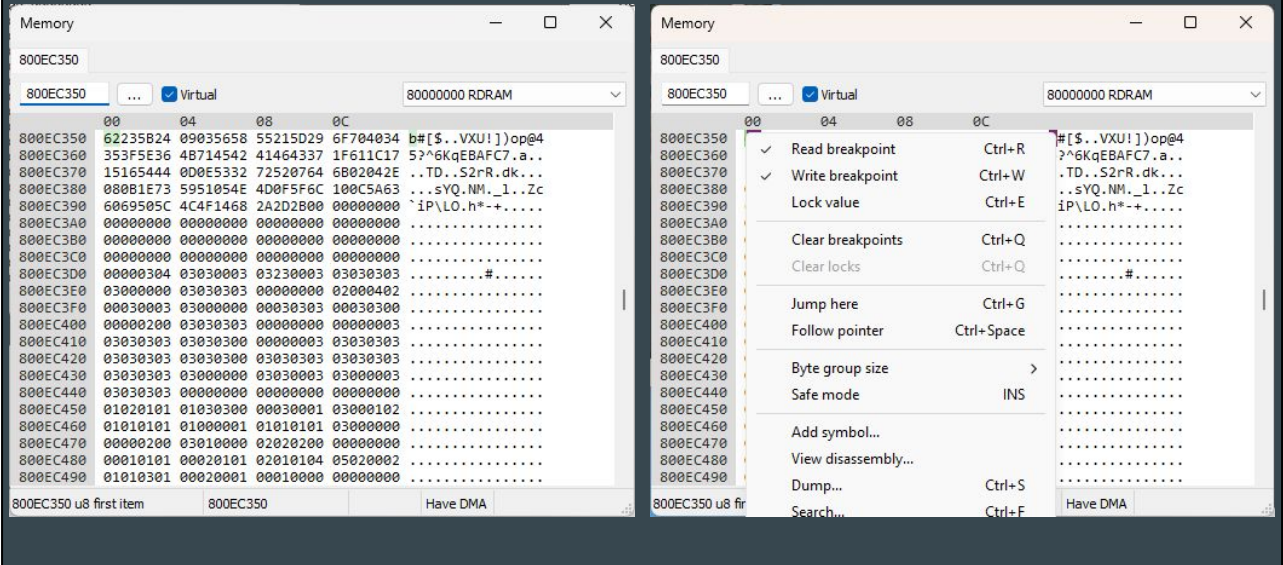


We need to find unused space in the ROM to store our new logic and look-up table. Generally, this can be found simply by visually scanning for an appropriate segment of 0's or FF's towards the end of the ROM.

Project64 is an N64 emulator with many useful tools that we will use extensively in this project.

Here is the memory view of the Project64 emulator, showing the segment we will target.

Finding Code That References the Inventory



We now need to locate existing code that uses the inventory in some way. Finding this memory location can be done manually, but with most games, I would suggest checking for existing Game Genie, GameShark, or Action Replay codes to give you a head start.

There were plenty of us snooping through game memory in the 90's...

Once you find these locations, Project64 can be used to set read and write breakpoints.

Finding Code That References the Inventory

The screenshot displays a debugger window with two main panes. The left pane, titled 'Commands', shows a list of assembly instructions with their addresses, commands, parameters, and symbols. The right pane shows the 'CPU General Purpose Registers' with their current values.

Commands Pane:

Address	Command	Parameters	Symbol
80032188	SB	A2, 0x0000 (V1)	
8003218C	LHU	V0, 0x081E (V1)	
80032190	ADDIU	V0, V0, 0x0001	
80032194	J	0x80032254	
80032198	SH	V0, 0x081E (V1)	
8003219C	LUI	A0, 0x800E	
800321A0	LHU	A0, 0x7A22 (A0)	
800321A4	ANDI	V1, A0, 0xFF00	
800321A8	BNE	V1, V0, 0x800321CC	
800321AC	ADDIU	V0, R0, 0x1400	
800321B0	ANDI	V1, A0, 0x00FF	
800321B4	ADDIU	V0, R0, 0x0008	
800321B8	LUI	AT, 0x800F	
800321BC	ADDU	AT, AT, V1	
800321C0	SB	V0, 0xC300 (AT)	
800321C4	J	0x80032258	
800321C8	ANDI	V0, S1, 0xC000	
800321CC	LHU	A0, 0x0000 (FP)	
800321D0	ANDI	V1, A0, 0xFF00	
800321D4	BNE	V1, V0, 0x800321F4	
800321D8	ADDIU	V0, R0, 0x1500	
800321DC	LUI	AT, 0x800F	
800321E0	SB	S7, 0xC46E (AT)	
800321E4	SB	R0, 0x005E (FP)	
800321E8	ANDI	V0, A0, 0x00FF	
800321EC	J	0x80032250	
800321F0	ORI	V0, V0, 0x0F00	
800321F4	BNE	V1, V0, 0x80032214	
800321F8	ADDIU	V0, R0, 0x1600	
800321FC	LUI	AT, 0x800F	
80032200	SB	R0, 0xC46E (AT)	
80032204	SB	R0, 0x005E (FP)	
80032208	ANDI	V0, A0, 0x00FF	
8003220C	J	0x80032250	
80032210	ORI	V0, V0, 0x0F00	
80032214	LHU	A0, 0x0000 (FP)	
80032218	ANDI	V1, A0, 0xFF00	
8003221C	BNE	V1, V0, 0x80032234	
80032220	ADDIU	V0, R0, 0x1700	

CPU General Purpose Registers:

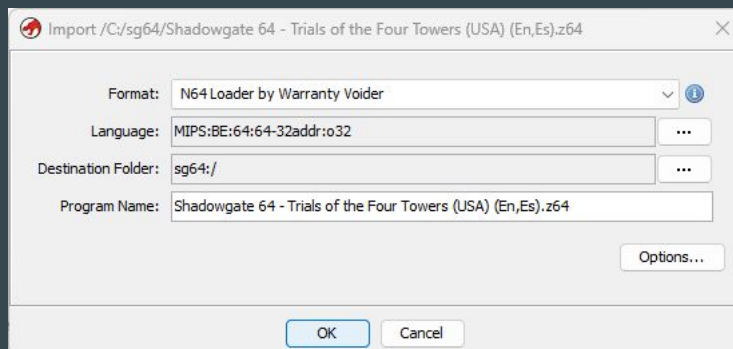
Register	Value	Register	Value
R0	00000000	S0	00000000
AT	FFFFFFFF	S1	00000000
V0	00000000	S2	00000000
V1	FFFFFFFF	S3	00000000
A0	00000000	S4	00000000
A1	FFFFFFFF	S5	00000000
A2	00000000	S6	00000000
A3	00000000	S7	00000000
T0	00000000	T8	00000000
T1	00000000	T9	FFFFFFFF
T2	00000000	K0	FFFFFFFF
T3	00000000	K1	00000000
T4	00000000	GP	00000000
T5	00000000	SP	FFFFFFFF
T6	00000000	FP	FFFFFFFF
T7	FFFFFFFF	RA	FFFFFFFF
HI	00000000	LO	00000000

Here is an example of a write breakpoint being triggered in Project64.

You can see the assembly instructions on the left, and the registers contents on the right.

Playing through enough of the game in the emulator will allow you to make a list of all of these code locations.

Finding Code That References the Inventory

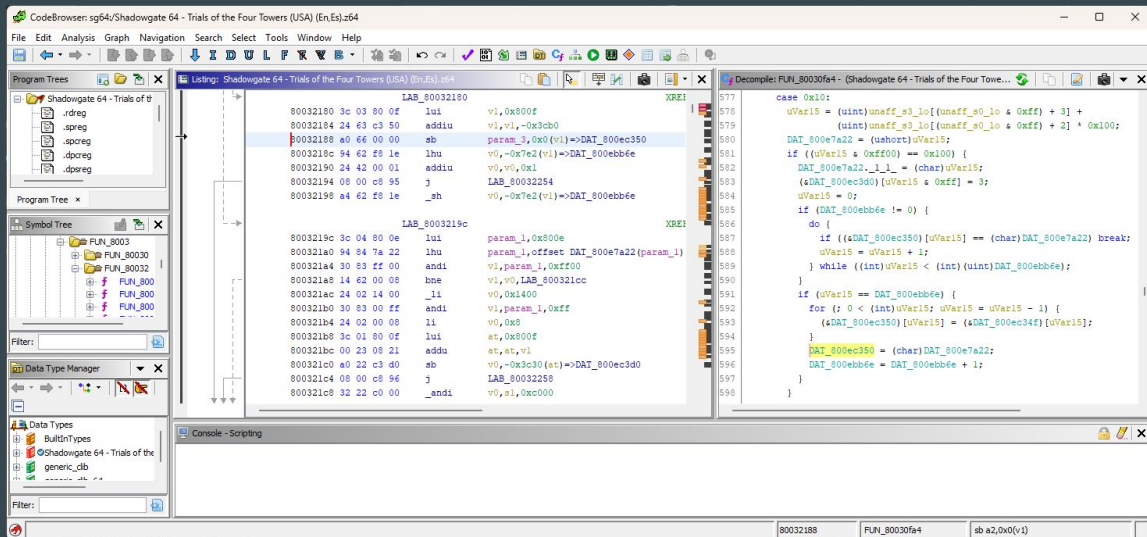


While it's entirely possible and often necessary to read the disassembled code, sometimes it's challenging to get the bigger picture from this output.

Ghidra is a software reverse engineering framework created and maintained by the NSA which provides a powerful, free way to decompile this code into C, which can be much easier to read and understand.

Warranty Voider on github has created many loaders for retro platforms, including the N64, which help with the initial setup and decompilation.

Finding Code That References the Inventory



Here you can see that same disassembly we saw in Project64 on the left, along with the decompiled C code on the right.

It's now much easier to tell that this code is responsible for moving each item ID down one position in the inventory array at 800ec350 (line 592), storing the new item ID at the front of that array (line 595), and incrementing the inventory size variable (line 596).

Cleaning Up Ghidra's Decompilation

```
void FUN_80030dd0(uint param_1) {
    uint uVar1;

    if ((param_1 & 0xff00) == 0) {
        *(undefined1 *)((param_1 & 0xff) + 0x800ec3d0) = 0xff;
    } else if ((param_1 & 0xff00) == 0x4000) {
        *(undefined1 *)((param_1 & 0xff) + 0x800ec3d0) = 0;
    } else if ((param_1 & 0x8000) == 0) {
        *(byte *)((param_1 & 0xff) + 0x800ec3d0) = (byte)((param_1 & 0xff00) >> 8) |
            *(byte *)((param_1 & 0xff) + 0x800ec3d0);
    } else {
        *(byte *)((param_1 & 0xff) + 0x800ec3d0) = *(byte *)((param_1 & 0xff) +
            0x800ec3d0) & (byte)(0xff00 - (param_1 & 0x7f00) >> 8);
    }
    // ...
}
```

That said, Ghidra's output isn't always the easiest to read.

Here is another example of some code we need to trace, and at first glance it's quite opaque.

Cleaning Up Ghidra's Decompilation

```
void updateItemStatus(uint16_t itemCode) {
    uint8_t itemId = itemCode & 0xffu;
    uint8_t itemAction = (itemCode & 0xff00u) >> 8;

    if (itemAction == SET_ALL) {
        inventoryStatus[itemId] = 0xffu;
    } else if (itemAction == CLEAR_ALL) {
        inventoryStatus[itemId] = 0u;
    } else if ((itemAction & STATUS_FLAG_CLEAR_ACTIONS) == 0u) {
        inventoryStatus[itemId] |= itemAction; // set bits in itemAction
    } else {
        // clear bits in itemAction
        inventoryStatus[itemId] &= ~(itemAction & 0x7fu);
    }
    // ...
}
```

However, I would encourage you to manually re-write these code segments as you explore and experiment.

Reverse Engineering is just guesswork until you start making progress!

Identifying All References

- Override the item icon displayed.
- Override the in-world item pick-up.
- Override the item status update.
 - Example: Jezibel gives you her "Pendant."
- Override the item status check.
 - Example: Jezibel will only release the "Family Diary" once you have received her "Pendant."

With all of these approaches, we can identify four code paths for referencing the inventory which will need to be adjusted for our randomizer.

The Best Laid Plans

- Override the item icon displayed.
 - **Exception: don't shuffle when handing items to Agaar.**
- Override the in-world item pick-up.
- Override the item status update.
 - Example: Jezibel gives you her "Pendant."
 - **Exception: don't shuffle when clearing item status.**
 - **Exception: don't shuffle when returning items to player.**
- Override the item status check.
 - Example: Jezibel will only release the "Family Diary" once you have received her "Pendant."
 - **Exception: don't shuffle "is equipped" checks.**
 - **Exception: don't shuffle when checking event triggers.**

However, you know what they say about the best laid plans.

Through experimentation, we also can identify cases where we DON'T want to apply our shuffling logic.

Updated Logic Based on Item Action

- Override the item icon displayed.
 - Exception: don't shuffle when handing items to Agaar.
- Override the in-world item pick-up.
- Override the item status update.
 - Example: Jezibel gives you her "Pendant."
 - **Exception: don't shuffle when clearing item status.**
 - Exception: don't shuffle when returning items to player.
- Override the item status check.
 - Example: Jezibel will only release the "Family Diary" once you have received her "Pendant."
 - **Exception: don't shuffle "is equipped" checks.**
 - Exception: don't shuffle when checking event triggers.

These exceptions are easily handled by checking what status is being applied to the item.

We Still Need a Good Way to Handle These

- Override the item icon displayed.
 - **Exception: don't shuffle when handing items to Agaar.**
- Override the in-world item pick-up.
- Override the item status update.
 - Example: Jezibel gives you her "Pendant."
 - Exception: don't shuffle when clearing item status.
 - **Exception: don't shuffle when returning items to player.**
- Override the item status check.
 - Example: Jezibel will only release the "Family Diary" once you have received her "Pendant."
 - Exception: don't shuffle "is equipped" checks.
 - **Exception: don't shuffle when checking event triggers.**

These, however... oh no. Does anyone have any ideas?

[Check the current room ID.]

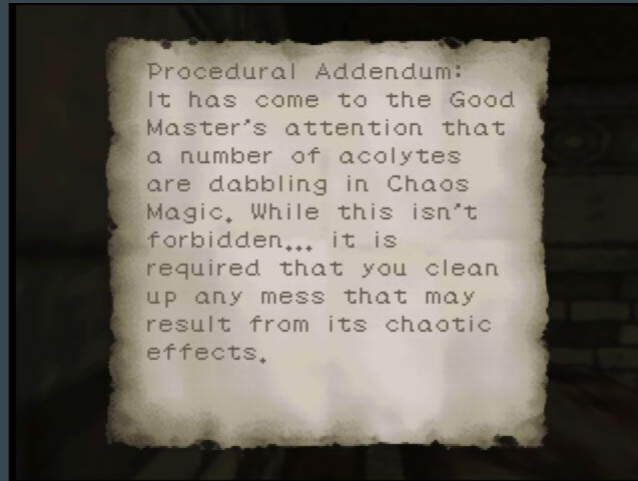
This was the solution that unlocked the rest of the logic, and was astutely suggested by my wife.

Strategy for Changing as Little as Possible

- For each of these references, jump to our borrowed code space.
- If our item action and room ID checks pass, override the item ID by looking up the new item in the shuffled look-up table.
- Minimize registers modified in new code for safety.
- Avoid modifying the program stack!

So here's the plan.

Strategy for Changing as Little as Possible



I think this note, which appears in-game, is a great summary.

Pop Quiz!

- Most N64 games execute 64-bit instructions.
 - True
 - False

Pop Quiz!

- Most N64 games execute 64-bit instructions.
 - True
 - **False**

While Nintendo marketing would have you believe otherwise, most games execute 32-bit instructions on the MIPS processor.

This is due to one simple fact: 64-bit instructions take twice as much space on the cart!

Because of this, we'll see examples of multiple 32-bit instructions being used to construct 64-bit addresses.

ROM Patch Example #1

```
// default item pickup path
replaceCommand(rom, 0x800320e8u, 0x0802832cu); // J 0x800a0cb0

replaceCommand(rom, 0x800a0cb0u, 0x3c01800au); // LUI AT, 0x800a
replaceCommand(rom, 0x800a0cb4u, 0x00260821u); // ADDU AT, AT, A2
replaceCommand(rom, 0x800a0cb8u, 0x90260C00u); // LBU A2, 0x0c00 (AT)
replaceCommand(rom, 0x800a0cbcu, 0x0800c83bu); // J 0x800320ec
replaceCommand(rom, 0x800a0cc0u, 0x30C200ffu); // ANDI V0, A2, 0x00ff
```

Here is a segment of the actual randomizer code.

On each line, we are overriding the value at a specific location with 4 bytes (32 bits) of new machine code (the result of compiling the human-readable code) as well as the assembly equivalent in a comment for convenience.

This code is hijacking execution at a specific instruction into our new code, then constructing an address inside our look-up table based on A2.

AT = 0x800a0c00 (look-up table) + A2 (item ID)

A2 = value at AT

V0 = A2, for future code

ROM Patch Example #2

```
// update item display
// prevent item look-up in room 0x01 (Dungeon)
replaceCommand(rom, 0x80030cc0u, 0x0c028320u); // JAL 0x800a0c80
replaceCommand(rom, 0x80030cc4u, 0x00000000u); // NOP

replaceCommand(rom, 0x800a0c80u, 0x3c01800eu); // LUI AT, 0x800e
replaceCommand(rom, 0x800a0c84u, 0x9421a726u); // LHU AT, 0xa726 (AT)
replaceCommand(rom, 0x800a0c88u, 0x24070001u); // ADDIU A3, R0, 0x0001
replaceCommand(rom, 0x800a0c8cu, 0x10270006u); // BEQ AT, A3, 0x800a0ca8
replaceCommand(rom, 0x800a0c90u, 0x30a700ffu); // ANDI A3, A1, 0x00ff
replaceCommand(rom, 0x800a0c94u, 0x30a5ff00u); // ANDI A1, A1, 0xff00
replaceCommand(rom, 0x800a0c98u, 0x3c01800au); // LUI AT, 0x800a
replaceCommand(rom, 0x800a0c9cu, 0x00270821u); // ADDU AT, AT, A3
replaceCommand(rom, 0x800a0ca0u, 0x90270c00u); // LBU A3, 0x0c00 (AT)
// ...
```

Here is another example, this time using Jump And Link instead of Jump, because the stack was already setup for a function call at this code location.

We first construct the address 0x800ea726, the location of the current room ID.

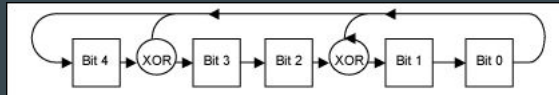
We then compare that room ID to 1. Since MIPS does not have an instruction to load an immediate byte, we add 1 with register R0, which always contains zero.

If these values are equal, we are in room ID 1, and we skip past the logic to apply our look-up table.

The rest of the logic is similar to example #1.

The Importance of Deterministic Seeds

- Linear Feedback Shift Registers to the rescue!
- 32-bit Input:
 - 0xdecafbad
- 8-bit Output:
 - 138
 - 69
 - 254
 - 127
 - 99



LFSR logic from analog.com

When a player creates a ROM with a particular seed value, it's important that that seed always result in the same shuffling.

This will enable things like easier debugging and sharing of particularly interesting experiences.

We will accomplish this by using a Linear Feedback Shift Register to generate all random values while shuffling.

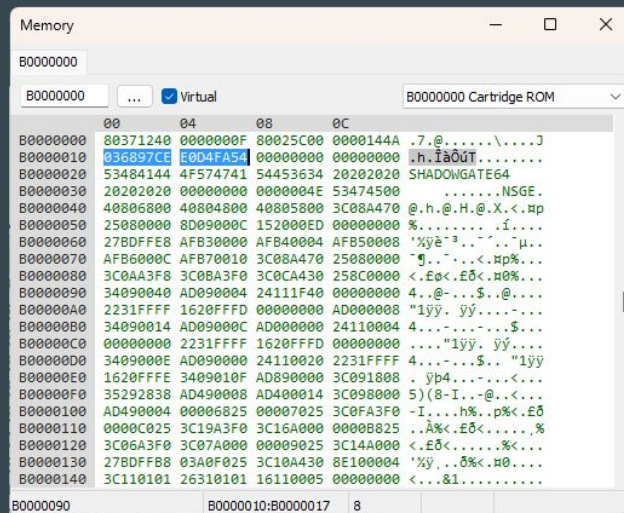
LFSRs use polynomial "taps" to modify new values based on old values while shifting.

LFSRs will always produce the same sequence of values for the same input seed, which is exactly what we want.

Limitations

- Certain items will never be shuffled, to avoid soft-locks or hangs.
- Certain items locations will never hold critical-path items, to avoid losing them via sequence breaks.

Additional Considerations: Header Checksums



Bytes 16 - 19 and 20 - 23 of every N64 game header contain two four-byte checksums based on the first 1MB of the cartridge.

Each game uses one of five different checksum algorithms, the most common of which is CIC-6102.

The CIC security chip confirms the checksums during Initial Program Load 3, the final stage of the console's built-in boot code, and will refuse to boot the game if there is a mismatch.

Most emulators will ignore this discrepancy.

When modifying a ROM, you should always update the checksums for maximum compatibility.

Pop Quiz!

- Accessing a game cartridge is much faster than accessing a CD.
 - True
 - False

Pop Quiz!

- Accessing a game cartridge is much faster than accessing a CD.
 - **True**
 - False

This is why load screens are much more prevalent on the Playstation compared to the N64.

Pop Quiz!

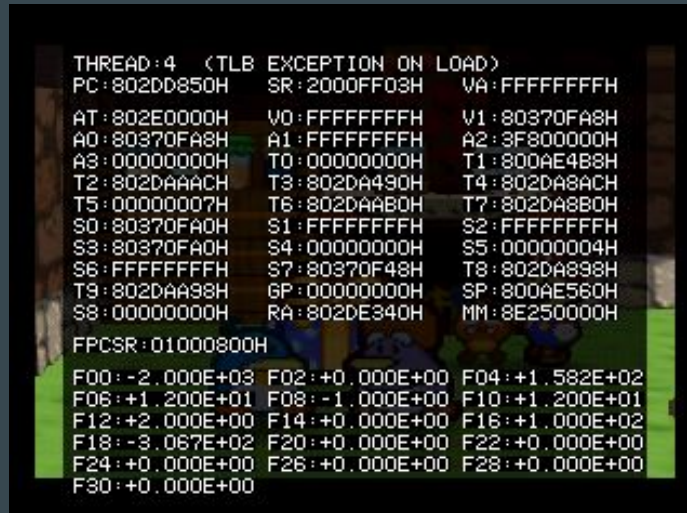
- Most N64 games execute code straight from the game cartridge.
 - True
 - False

Pop Quiz!

- Most N64 games execute code straight from the game cartridge.
 - True
 - **False**

It's actually still faster to read from RAM, so most N64 games will copy code into memory before executing it.

Add'l Considerations: Translation Lookaside Buffer



Paper Mario 64 crash handler screenshot by A gossip-loving Toad on mariowiki.com

Large copies from cartridge to RAM are typically handled by the Translation Lookaside Buffer system on the N64.

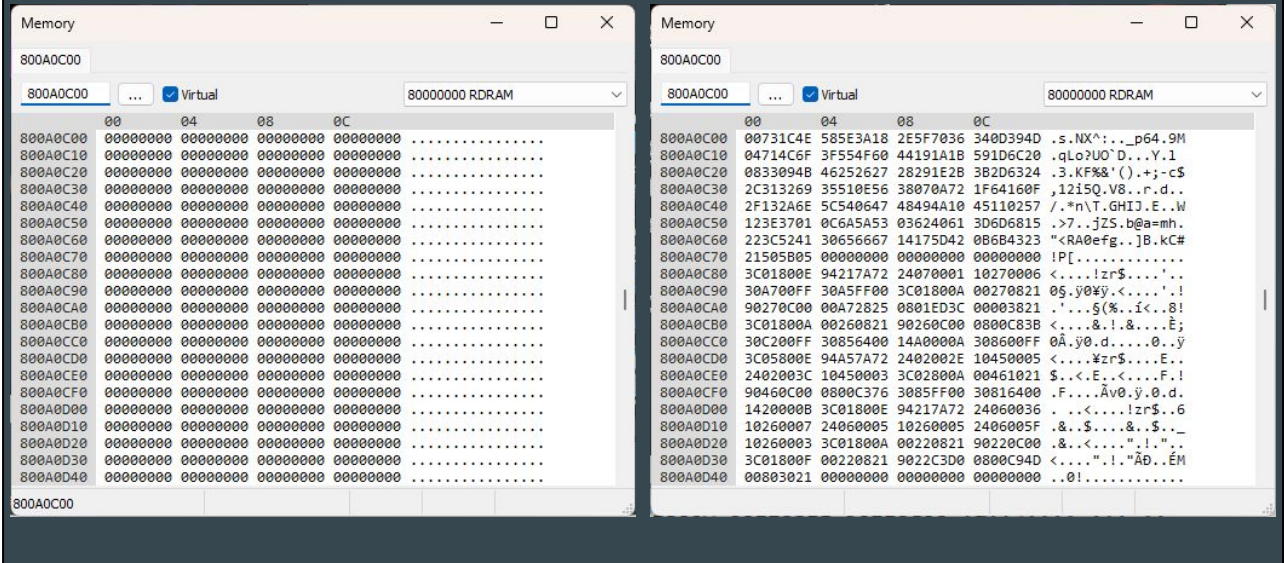
This system is responsible for mapping virtual memory operations to physical memory locations.

Here's the bad news: Shadowgate 64 only performs one TLB load at boot, which makes it easy to work with, but your favorite game might do this quite often.

Viper182 said it best: "TLB mapping is pretty evil in the eyes of ASM hackers. This is the reason you don't see 1337 ASM codes for Goldeneye and very few for Perfect Dark."

The screenshot is an example of a virtual memory request in Paper Mario 64 that cannot be accommodated by the TLB, so the game just crashes.

Putting It All Together



Putting all of this together, we're finally ready to fill in our borrowed ROM space!

Thank you!

Stay connected:

- feedback@sleepybitsgames.com
- <https://sleepybits.com/prge2026.html>
- <https://github.com/grendell>
- nesdwarves.com

