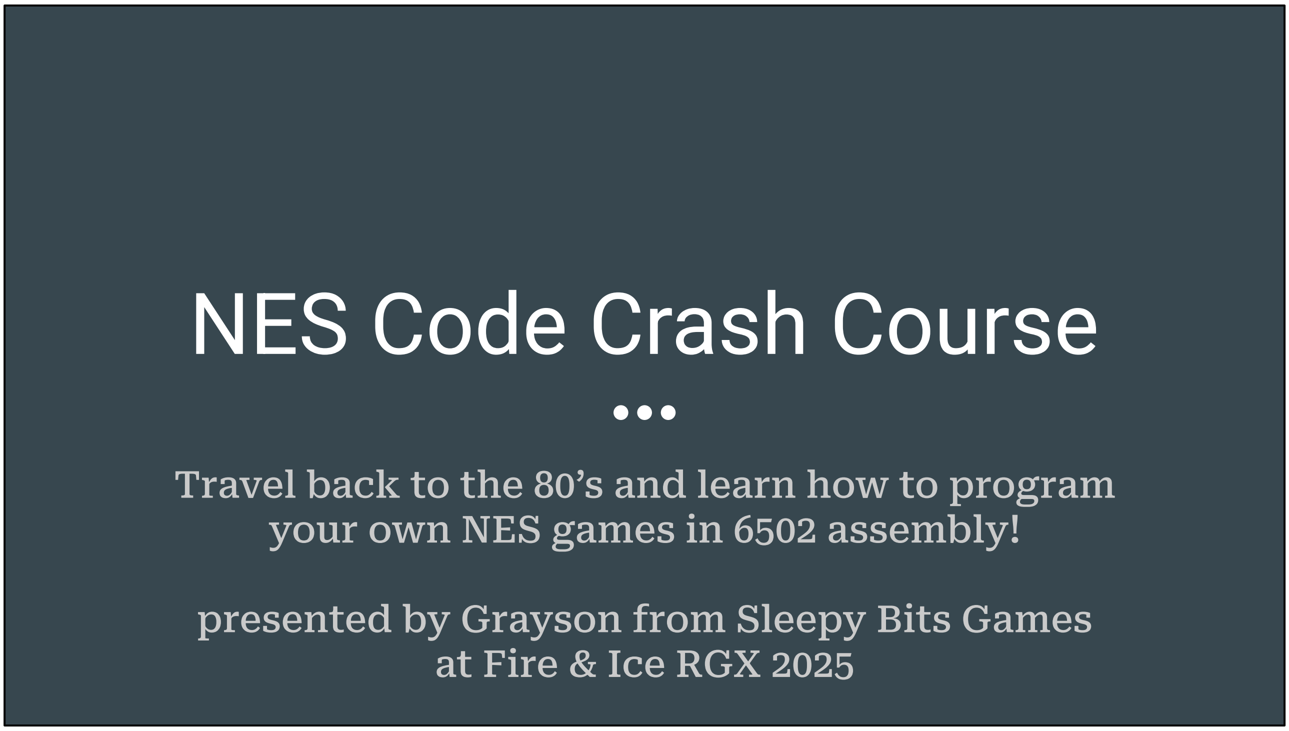


NES Code Crash Course

Travel back to the 80’s and learn how to program your own NES games in 6502 assembly!

presented by Grayson from Sleepy Bits Games at Fire & Ice RGX 2025

I have been in the games industry since 2007, mostly focusing on graphics and optimization.
I have dedicated the past 2.5 years to the NES specifically.
There are a tremendous number of resources online, some of which I will highlight towards the end of this talk.

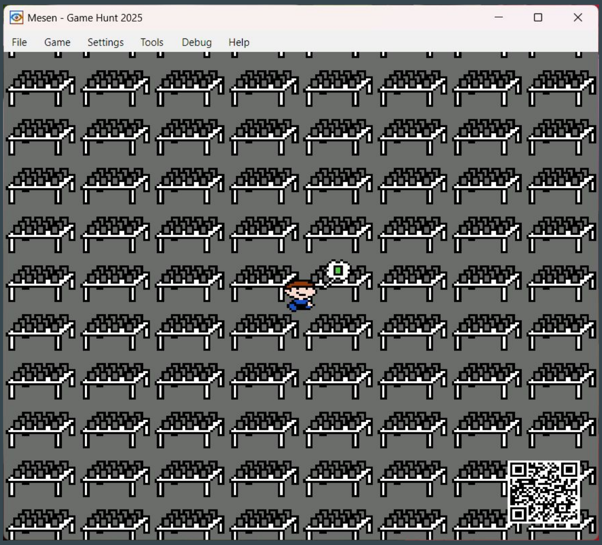
Everything in this presentation is open source!

- Feel free to use this project to get a running start on your NES project!
- No need to take notes or try to remember every detail.

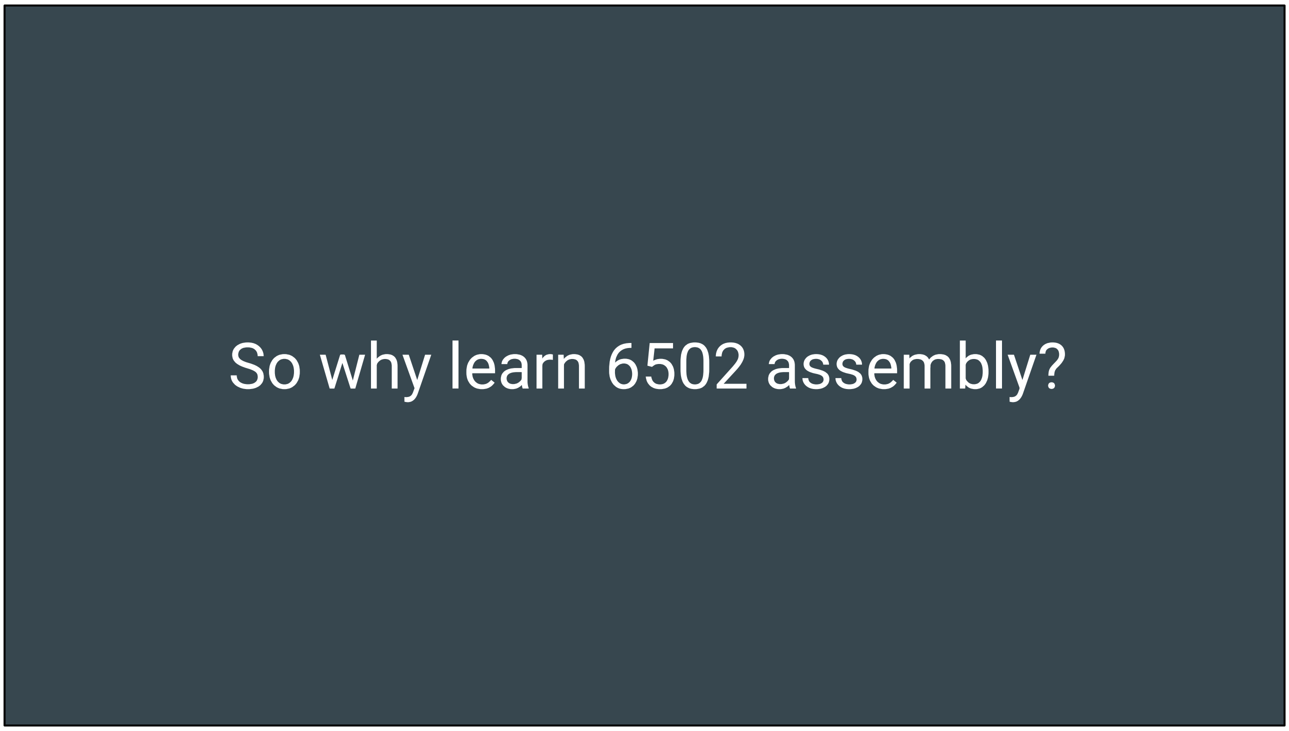
- All you need is this repository:

  https://github.com/grendell/gamehunt2025

Example Project: Game Hunt 2025



Let’s jump to the emulator for a quick demonstration.



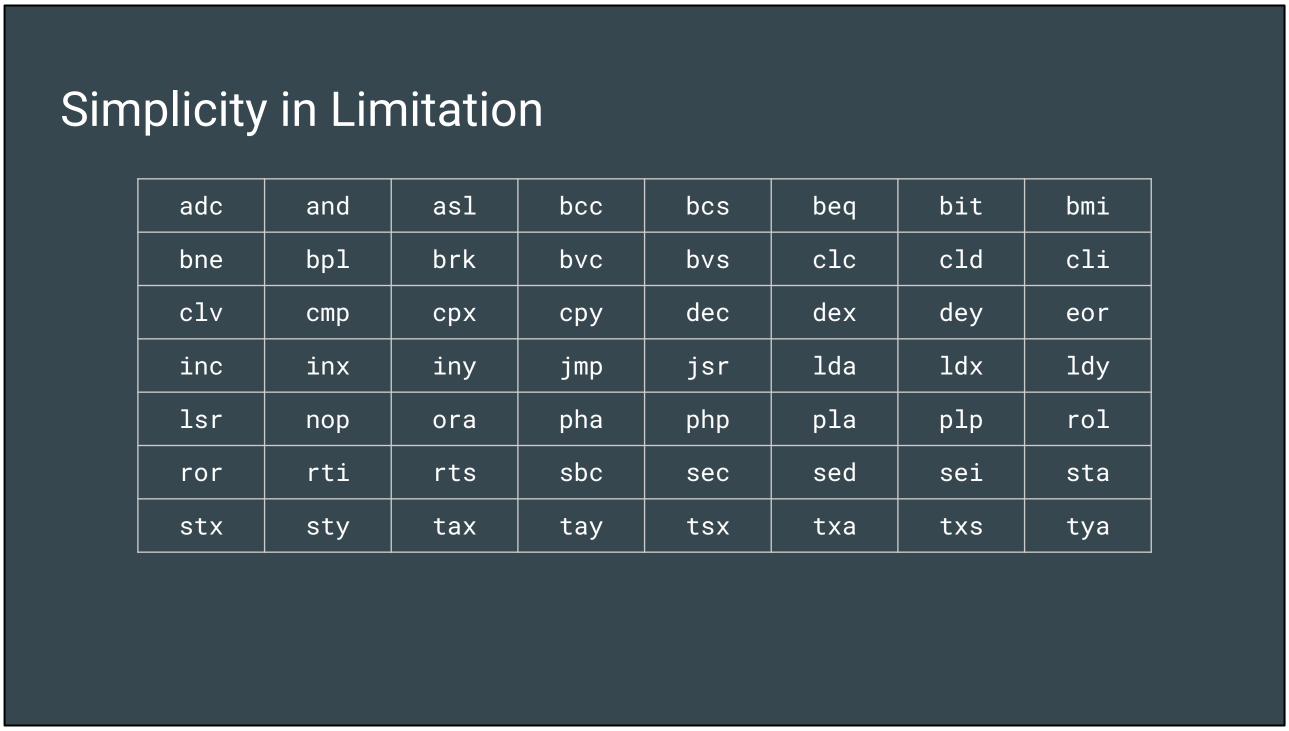
So why learn 6502 assembly?

It’s not the only way to make NES games in 2025.
Great alternatives exist.
It’s definitely not the easiest way.

Wide Range of Retro Hardware

Apple][image by Rama & Musée Bolo - Own work, CC
All other images by Evan-Amos - Own work, Public Domain

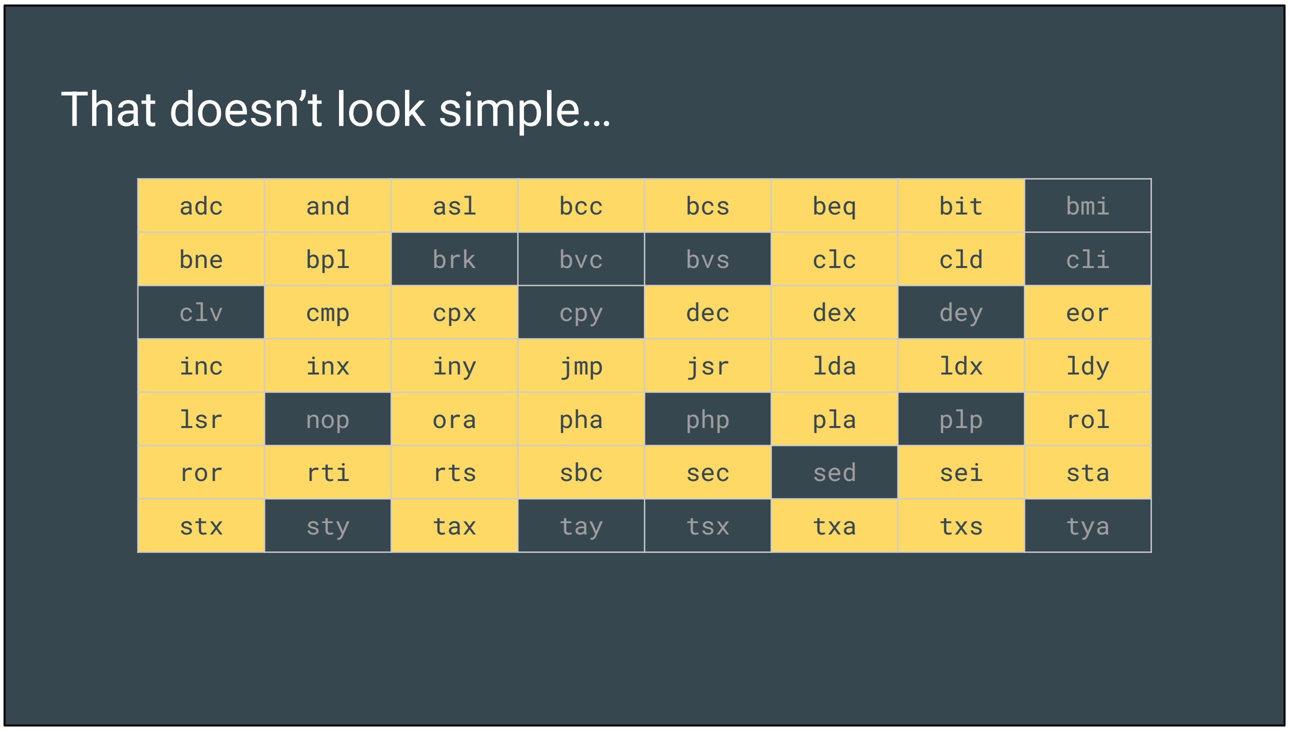
Apple][, Atari 2600, VIC-20, Commodore64



Simplicity in Limitation

adc	and	asl	bcc	bcs	beq	bit	bmi
bne	bpl	brk	bvc	bvs	clc	cld	cli
clv	cmp	cpx	cpy	dec	dex	dey	eor
inc	inx	iny	jmp	jsr	lda	ldx	ldy
lsr	nop	ora	pha	php	pla	plp	rol
ror	rti	rts	sbc	sec	sed	sei	sta
stx	sty	tax	tay	tsx	txa	txs	tya

Here are all the commands available in 6502 assembly!
We will be introducing how to read these and how they operate shortly.



That doesn't look simple…

adc	and	asl	bcc	bcs	beq	bit	bmi
bne	bpl	brk	bvc	bvs	clc	cld	cli
clv	cmp	cpx	cpy	dec	dex	dey	eor
inc	inx	iny	jmp	jsr	lda	ldx	ldy
lsr	nop	ora	pha	php	pla	plp	rol
ror	rti	rts	sbc	sec	sed	sei	sta
stx	sty	tax	tay	tsx	txa	txs	tya

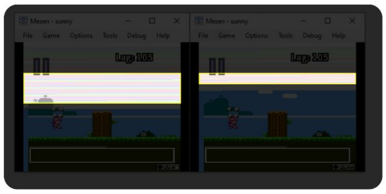
By the time we're done, you'll have example usage of all the highlighted commands.

Speak Directly to the Hardware

Matt Hughson (NES & Game Boy D
@matthughson

With some help from the #nesdev folks, Super Sunny World got a huge performance boost today!

Highlighted in the screenshot on the left is how much of the "frame time" was dedicated to moving 4 smashed blocks. On the right is the exact same effect taking 3x less time!

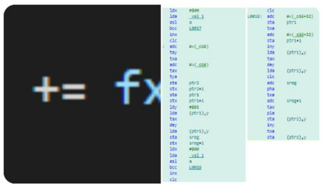

Mesen - sunny
File Game Options Tools Debug Help
Lag: 105
Mesen - sunny
File Game Options Tools Debug Help
Lag: 105

3:40 PM · March 1, 2023

Matt Hughson (NES & Game Boy D
@matthughson

As you might know, most of the game is written in C. Moving things around the screen involves lots of code like this (adding 2 16-bit values)

This looks simple enough, but compiles to ~50 assembly instructions. If written by hand in assembly, this would be about 10 instructions.

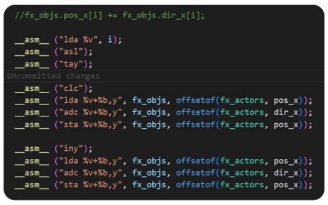


3:40 PM · March 1, 2023

Matt Hughson (NES & Game Boy D
@matthughson

Often when working in C on the NES, we can coax the compiler to generate decent assembly, but in this case we couldn't figure out how.

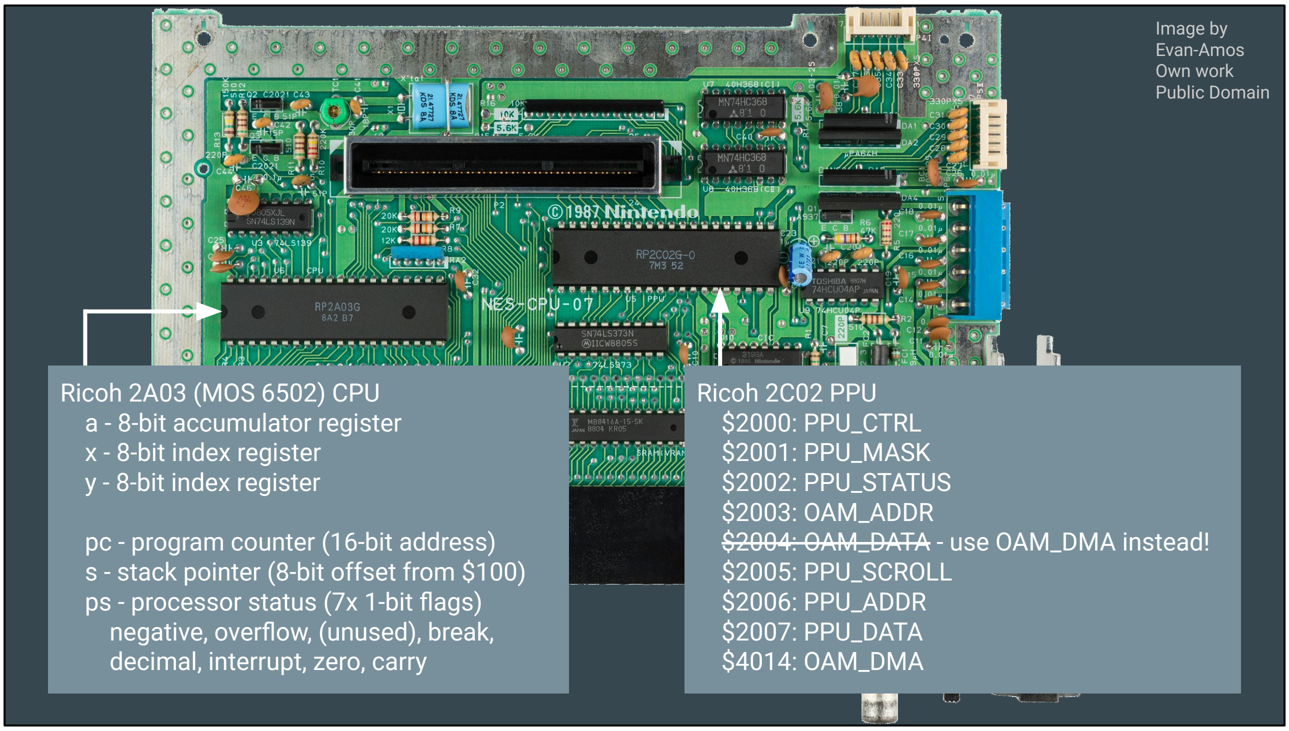
Luckily the compiler allows me to *inline* raw assembly (16bit add by lidnariq) in C. Here's what that 1 line of C becomes (6x perf improvement)



```
//fx_objs.pos_x[i] += fx_objs.dir_x[i];

__asm__ ("lda %v", i);
__asm__ ("asl");
__asm__ ("tay");
Uncommitted changes
__asm__ ("clc");
__asm__ ("lda %v+%b,y", fx_objs, offsetof(fx_actors, pos_x));
__asm__ ("adc %v+%b,y", fx_objs, offsetof(fx_actors, dir_x));
__asm__ ("sta %v+%b,y", fx_objs, offsetof(fx_actors, pos_x));

__asm__ ("iny");
__asm__ ("lda %v+%b,y", fx_objs, offsetof(fx_actors, pos_x));
__asm__ ("adc %v+%b,y", fx_objs, offsetof(fx_actors, dir_x));
__asm__ ("sta %v+%b,y", fx_objs, offsetof(fx_actors, pos_x));
```

3:40 PM · March 1, 2023

Matt Hughson, developer of Super Sunny World, Witch n' Wiz, and many others.
Give him a follow to see all the cool things he's doing on the NES.
Reading Rainbow: You don't have to take my word for it!



Ricoh 2A03 (MOS 6502) CPU
- a - 8-bit accumulator register
- x - 8-bit index register
- y - 8-bit index register

- pc - program counter (16-bit address)
- s - stack pointer (8-bit offset from $100)
- ps - processor status (7x 1-bit flags)
  negative, overflow, (unused), break, decimal, interrupt, zero, carry

Ricoh 2C02 PPU
- $2000: PPU_CTRL
- $2001: PPU_MASK
- $2002: PPU_STATUS
- $2003: OAM_ADDR
- ~~$2004: OAM_DATA~~ - use OAM_DMA instead!
- $2005: PPU_SCROLL
- $2006: PPU_ADDR
- $2007: PPU_DATA
- $4014: OAM_DMA

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.

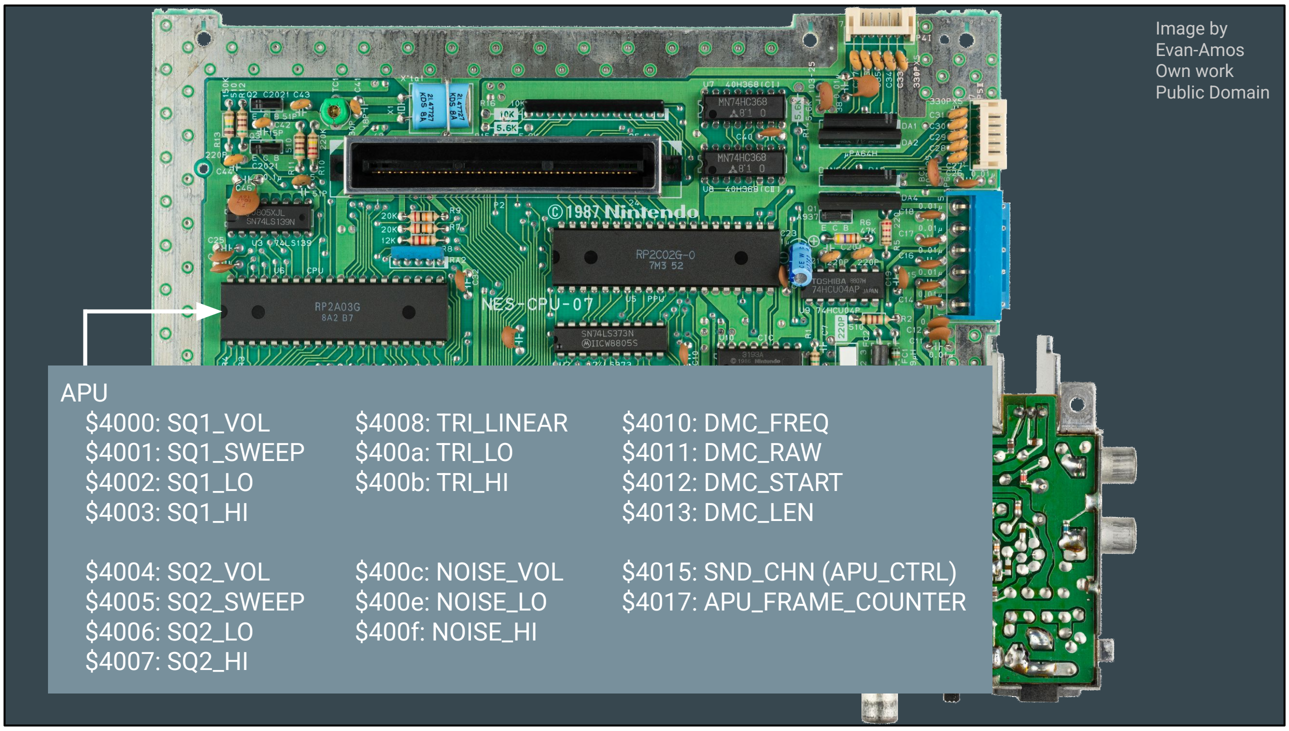
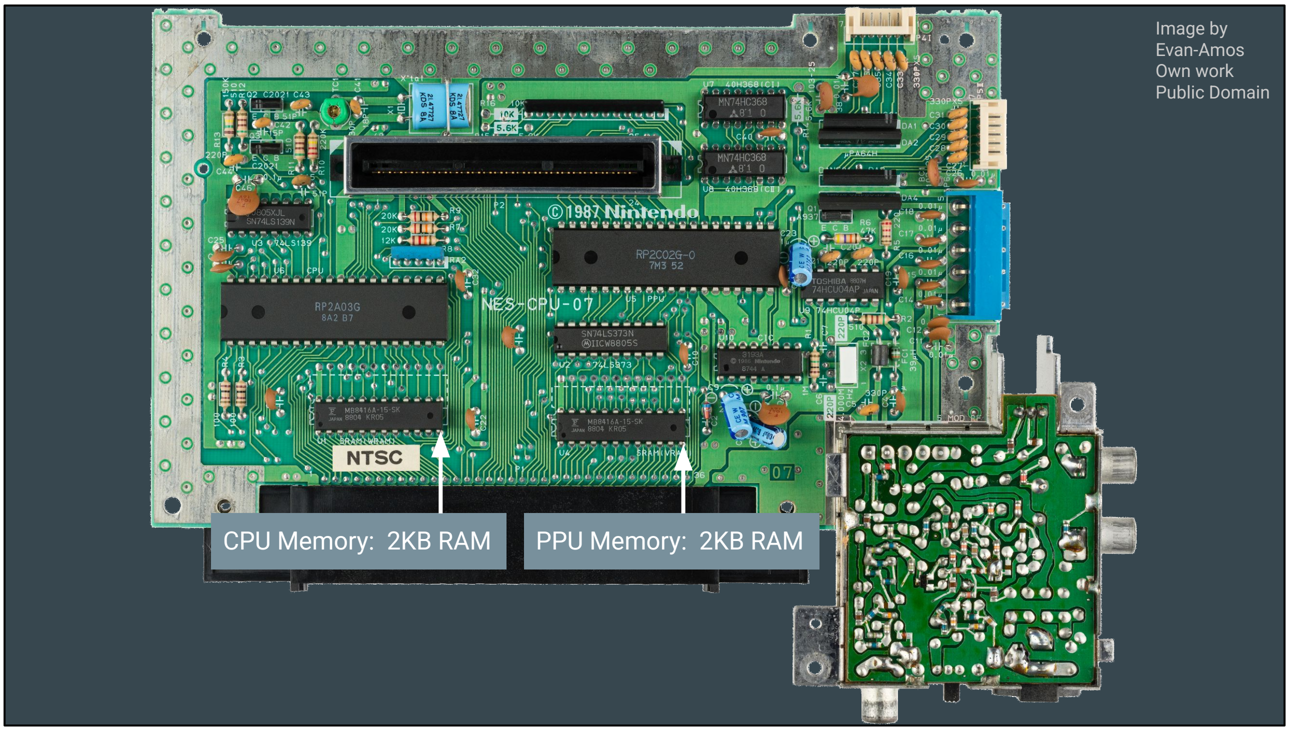


Image by
Evan-Amos
Own work
Public Domain

APU

$4000: SQ1_VOL	$4008: TRI_LINEAR	$4010: DMC_FREQ
$4001: SQ1_SWEEP	$400a: TRI_LO	$4011: DMC_RAW
$4002: SQ1_LO	$400b: TRI_HI	$4012: DMC_START
$4003: SQ1_HI		$4013: DMC_LEN
$4004: SQ2_VOL	$400c: NOISE_VOL	$4015: SND_CHN (APU_CTRL)
$4005: SQ2_SWEEP	$400e: NOISE_LO	$4017: APU_FRAME_COUNTER
$4006: SQ2_LO	$400f: NOISE_HI	
$4007: SQ2_HI		

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 (bleeps and bloops), one triangle wave channel (lower, sustained notes),
the noise channel (explosions, buzzers, percussion), and a Delta Modulation Channel for sample playback.

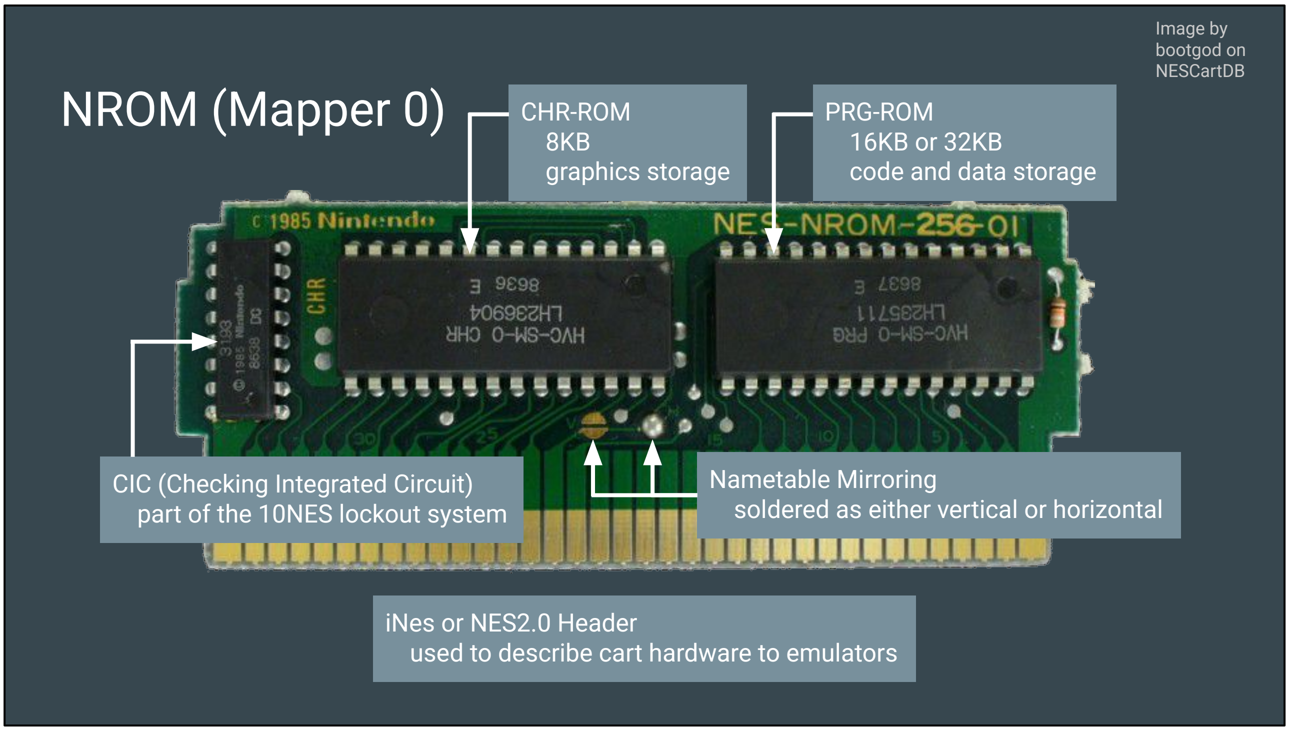


The CPU cannot directly access the PPU's memory.
It must issue commands via the PPU to transfer data.

Image by
Evan-Amos
Own work
Public Domain

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)

Image by bootgod on NESCartDB

CHR-ROM
8KB
graphics storage

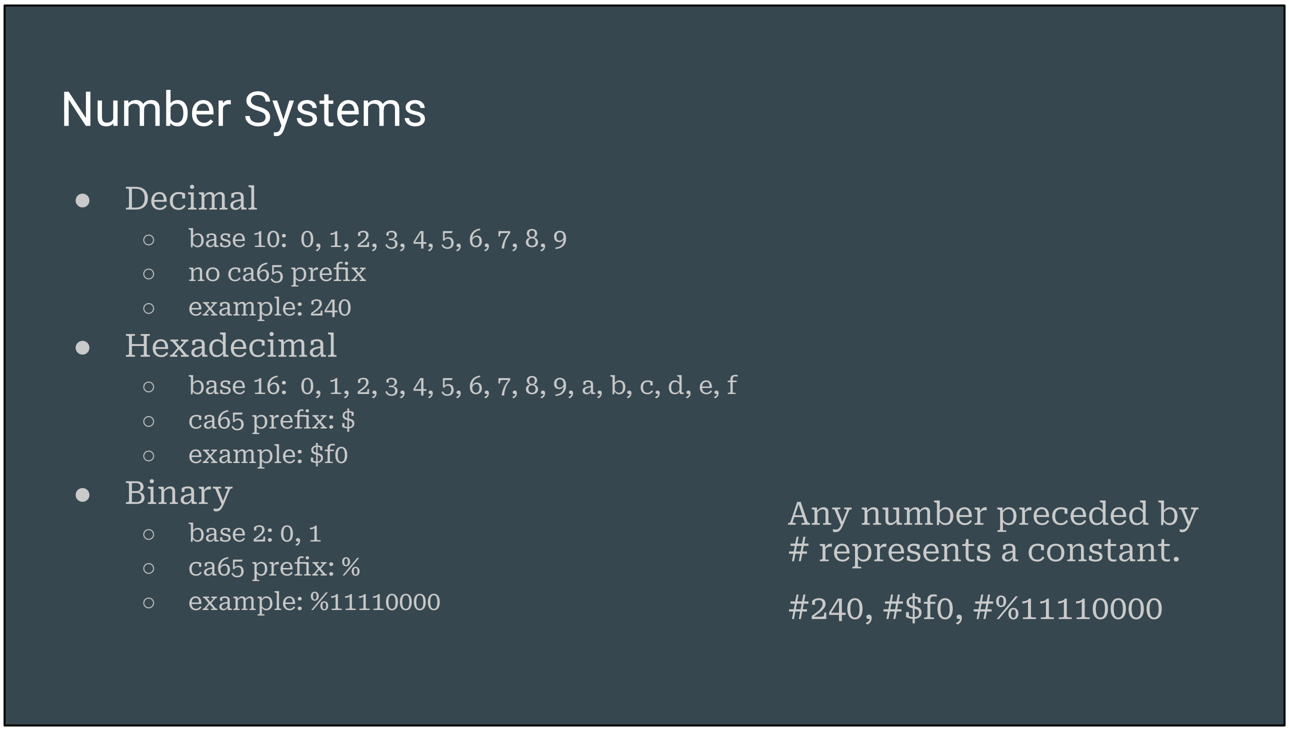
PRG-ROM
16KB or 32KB
code and data storage

CIC (Checking Integrated Circuit)
part of the 10NES lockout system

Nametable Mirroring
soldered as either vertical or horizontal

iNes or NES2.0 Header
used to describe cart hardware to emulators

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, which we will see in just a moment.
This has even been used in official products, like Animal Crossing and the NES Classic!



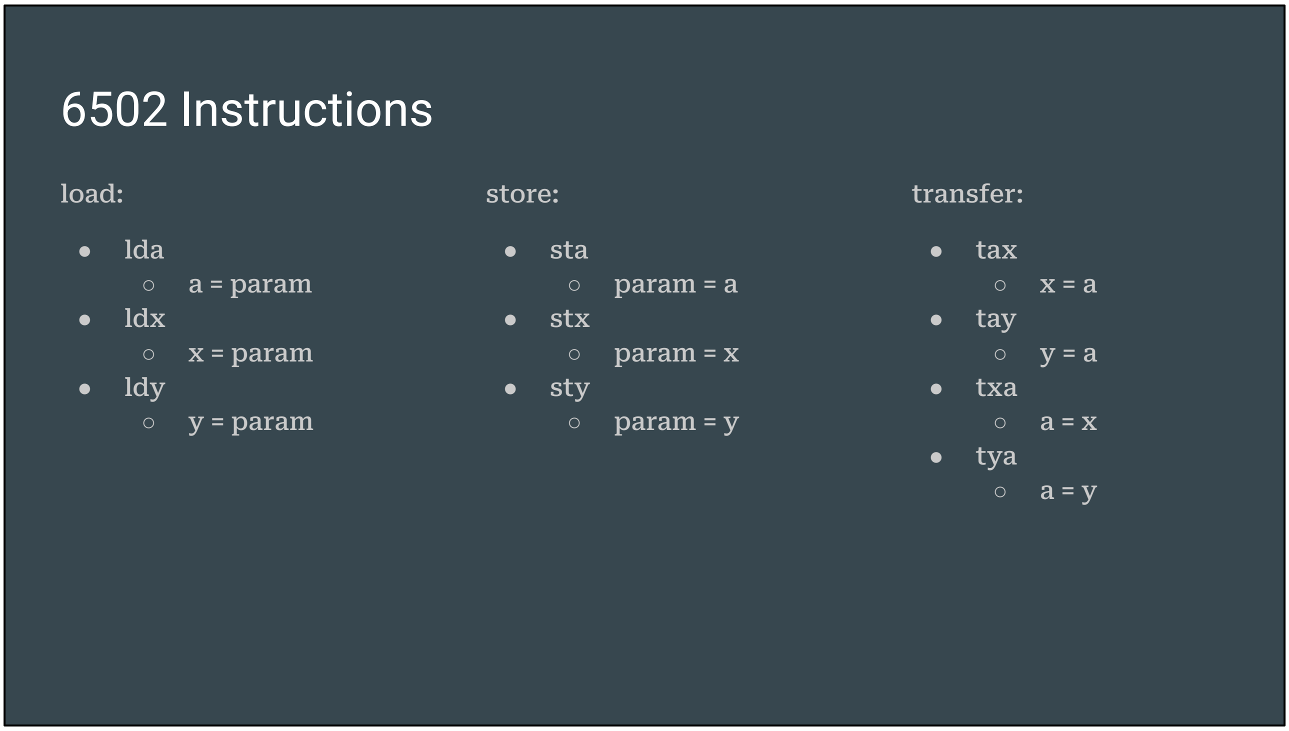
Number Systems

- Decimal
  - base 10: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - no ca65 prefix
  - example: 240
- Hexadecimal
  - base 16: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f
  - ca65 prefix: $
  - example: $f0
- Binary
  - base 2: 0, 1
  - ca65 prefix: %
  - example: %11110000

Any number preceded by # represents a constant.

#240, #$f0, #%11110000

You may have noticed letters in the APU registers. In fact, all addresses are typically represented in hexadecimal.
Hexadecimal is just a short-hand for groups of four binary digits.



6502 Instructions

load:

- lda
  - a = param
- ldx
  - x = param
- ldy
  - y = param

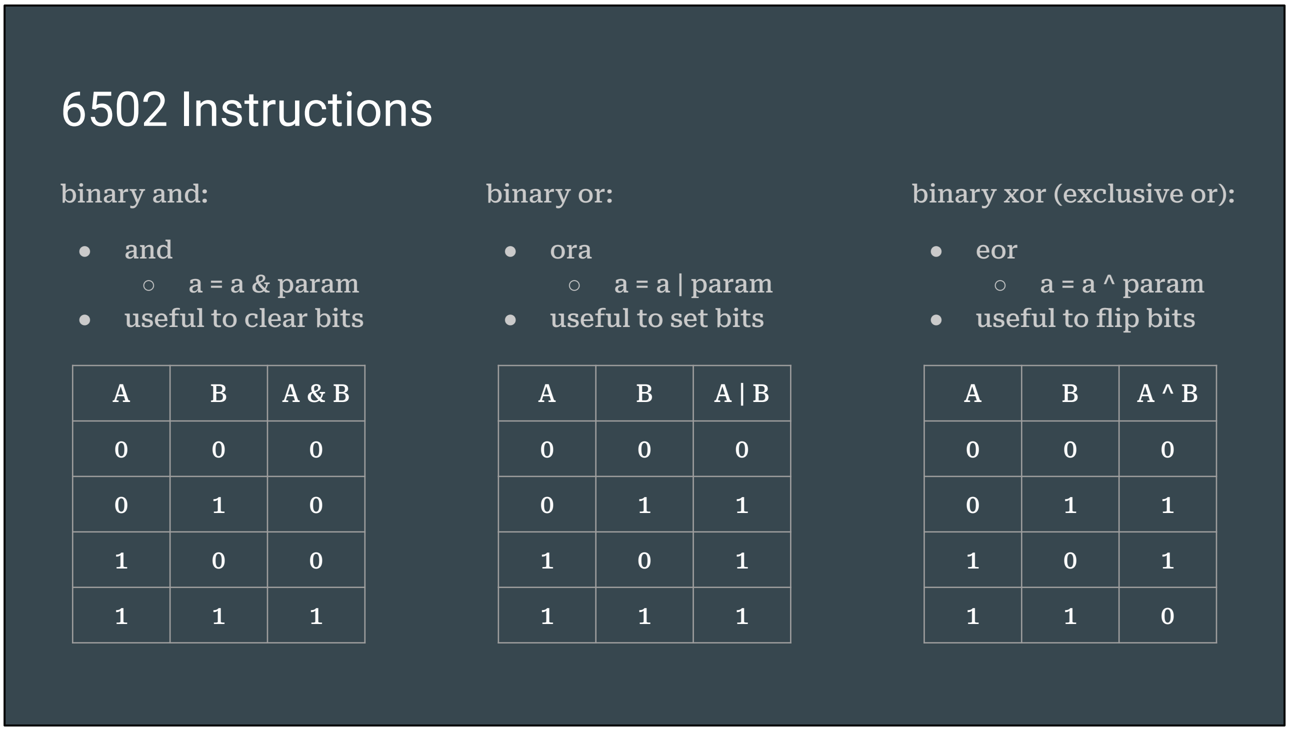
store:

- sta
  - param = a
- stx
  - param = x
- sty
  - param = y

transfer:

- tax
  - x = a
- tay
  - y = a
- txa
  - a = x
- tya
  - a = y

Load can accept a constant, variable name, an address, or what's called an indirection/pointer, with or without an offset.
Store can accept all of these except a constant, since the parameter is the destination.
Transfer is in the order of "transfer this to that" and never has a parameter.



6502 Instructions

binary and:

- and
  - a = a & param
- useful to clear bits

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

binary or:

- ora
  - a = a | param
- useful to set bits

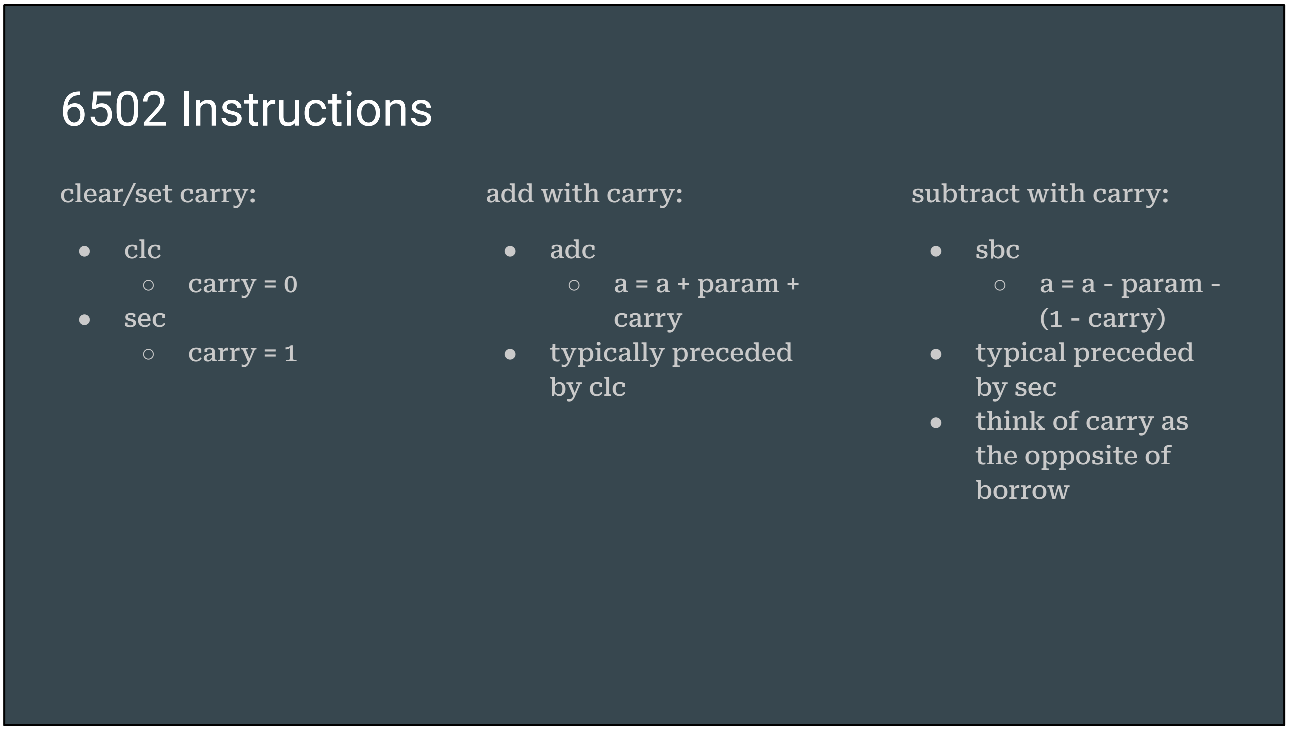
A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

binary xor (exclusive or):

- eor
  - a = a ^ param
- useful to flip bits

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

All of these instructions can accept a constant, variable name, an address, or what's called an indirection/pointer, with or without an offset, just like load.
And is useful to clear or isolate bits, or is useful to set bits, and xor is useful to flip bits.



6502 Instructions

clear/set carry:

- clc
    - carry = 0
- sec
    - carry = 1

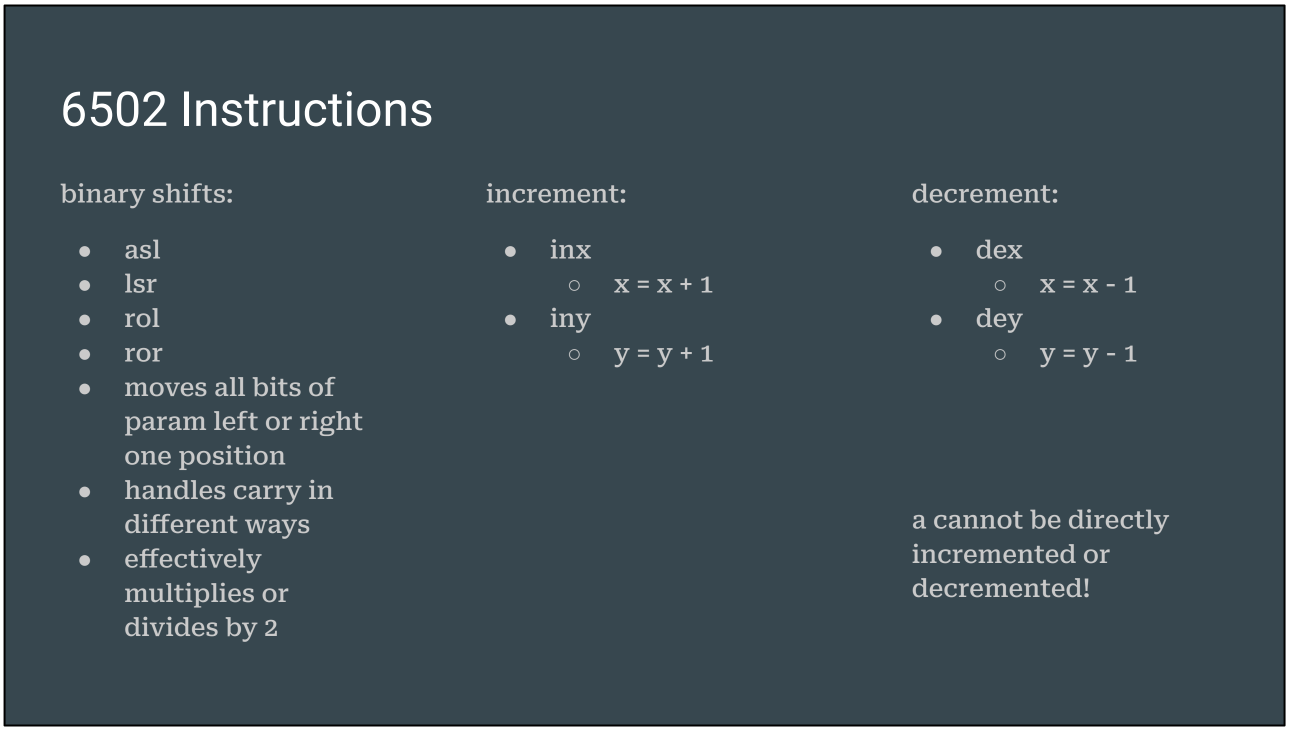
add with carry:

- adc
    - a = a + param + carry
- typically preceded by clc

subtract with carry:

- sbc
    - a = a - param - (1 - carry)
- typical preceded by sec
- think of carry as the opposite of borrow

Remember carry and borrow from elementary school? They're the same here!
Checking carry after an operation is useful to detect when the result exceeds 8 bits.



6502 Instructions

binary shifts:

- asl
- lsr
- rol
- ror
- moves all bits of param left or right one position
- handles carry in different ways
- effectively multiplies or divides by 2

increment:

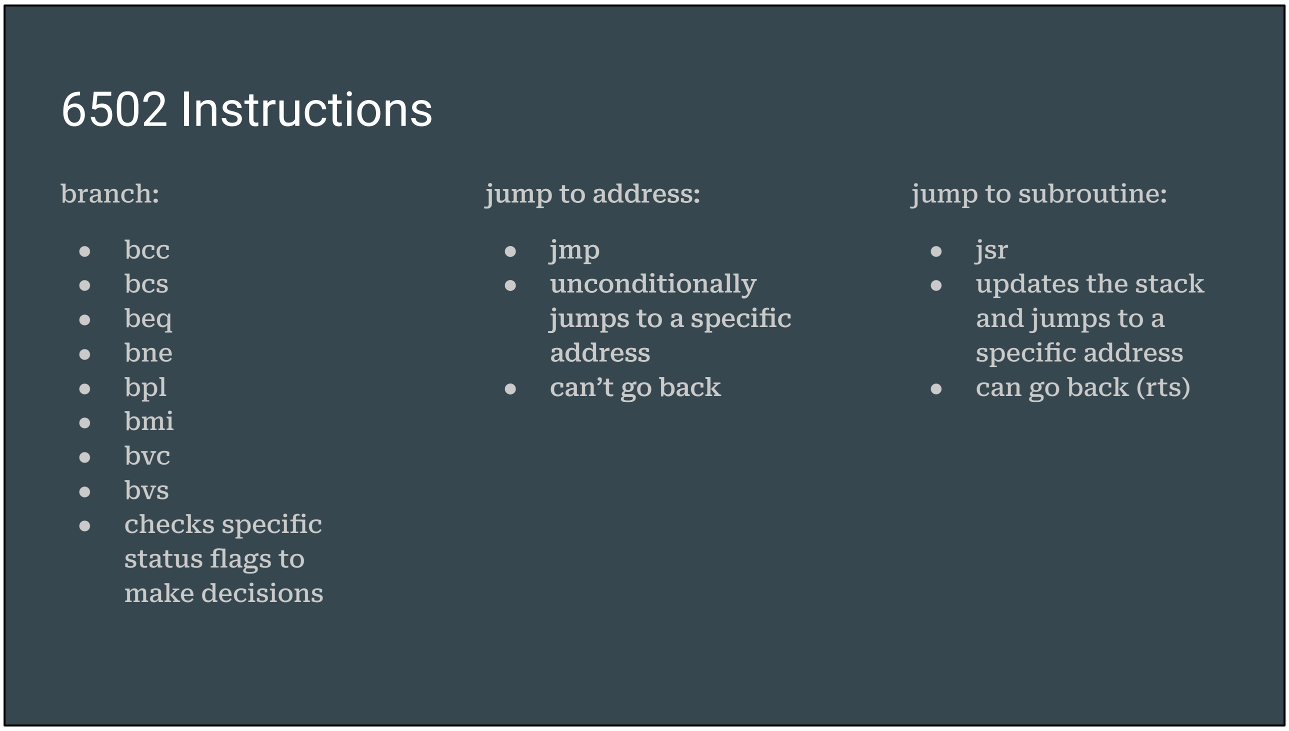
- inx
  - x = x + 1
- iny
  - y = y + 1

decrement:

- dex
  - x = x - 1
- dey
  - y = y - 1

a cannot be directly incremented or decremented!

The differences between shifts are beyond the scope of this talk, but we will see each!
x and y are often used for counting and offsetting, so only they can be incremented and decremented.
If you need to increment or decrement a, adc and sbc are usually best.
If you need arbitrary multiplication or division, you'll need to provide handwritten implementations, which will be expensive!



6502 Instructions

branch:

- bcc
- bcs
- beq
- bne
- bpl
- bmi
- bvc
- bvs
- checks specific status flags to make decisions

jump to address:

- jmp
- unconditionally jumps to a specific address
- can't go back

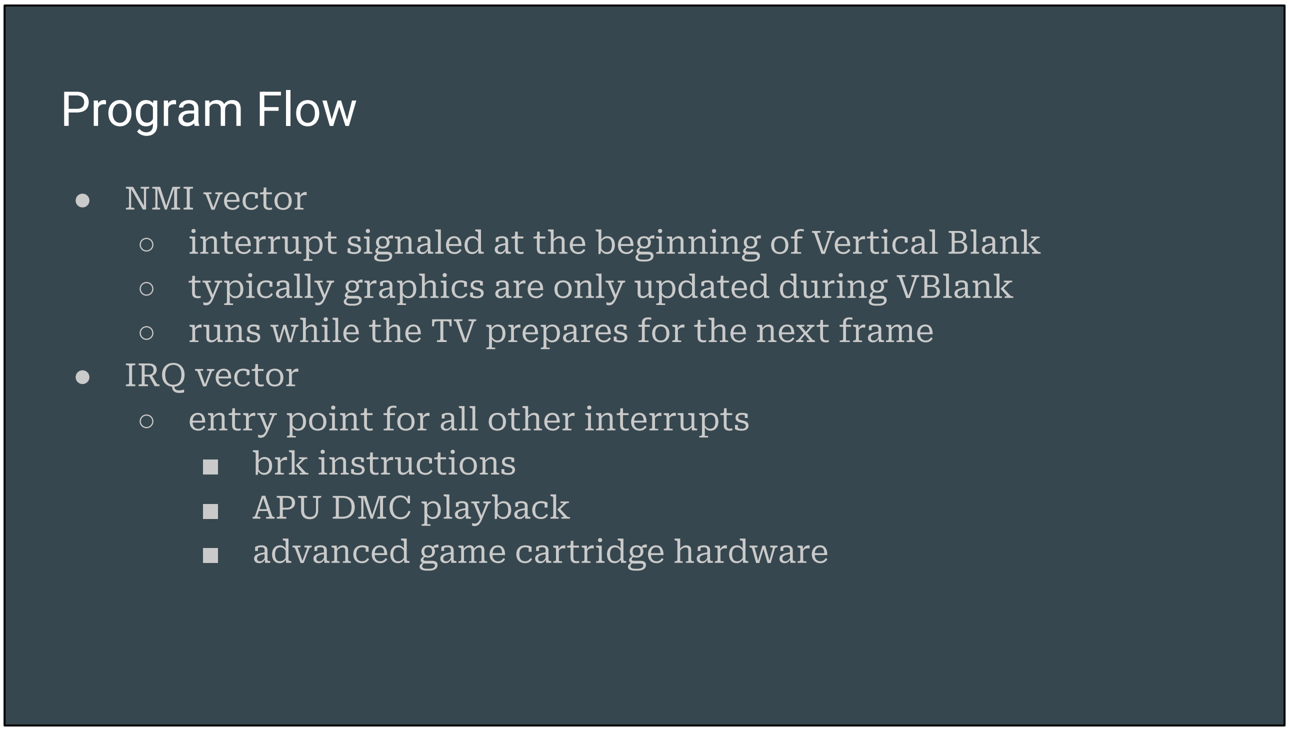
jump to subroutine:

- jsr
- updates the stack and jumps to a specific address
- can go back (rts)

Branches are how we make decisions, based on the various status flags we saw earlier, just like "if" clauses in more modern languages.
Because jmp doesn't update the stack, there's no automatic way to return from our destination.
jmp can be useful for chaining together subroutines that always execute in order and for "else" clauses.

Program Flow

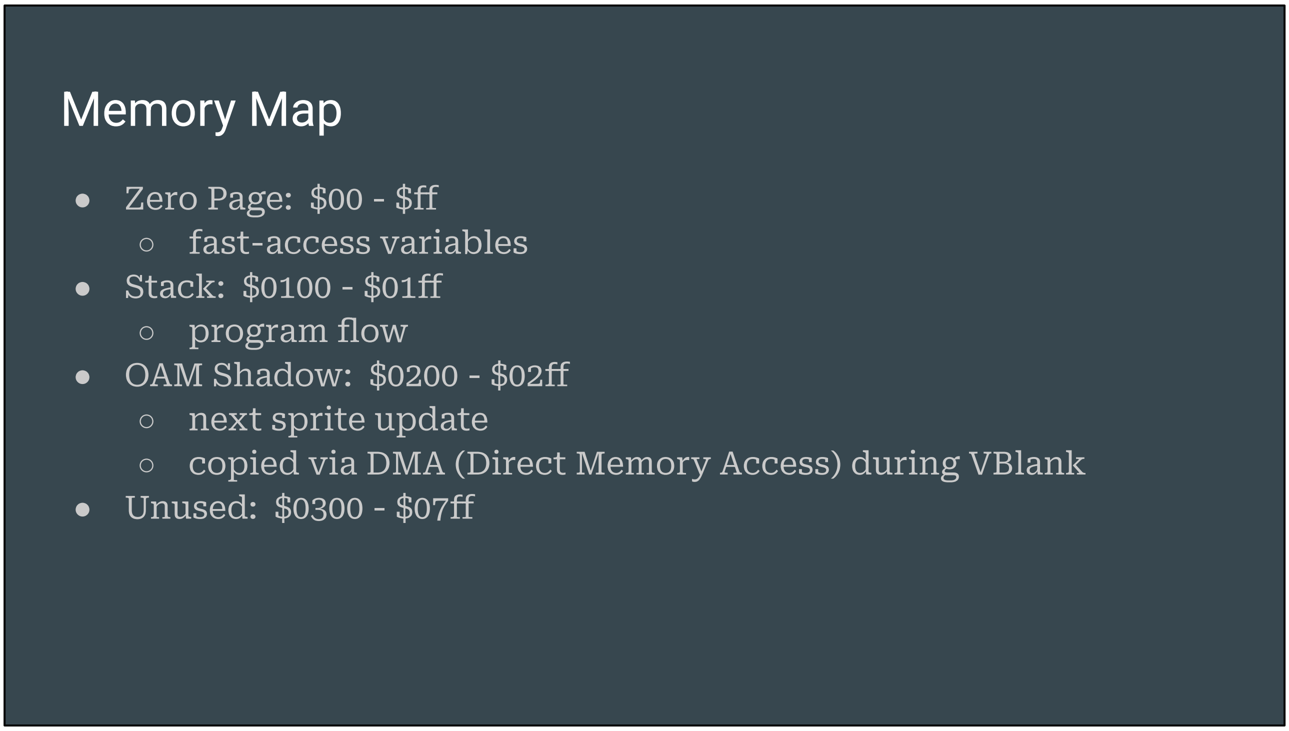
- reset vector
  - starting point for code execution
  - program counter jumps here during power-on and reset
  - responsible for game initialization
- game loop
  - typically an infinite loop after initialization
  - usually waits for a graphics update before iterating
  - responsible for per-frame 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

Interrupts are responsible for saving and restoring register values and status flags, as we will see.
IRQ can be enabled/disabled with sei/cli instructions, as we will see.

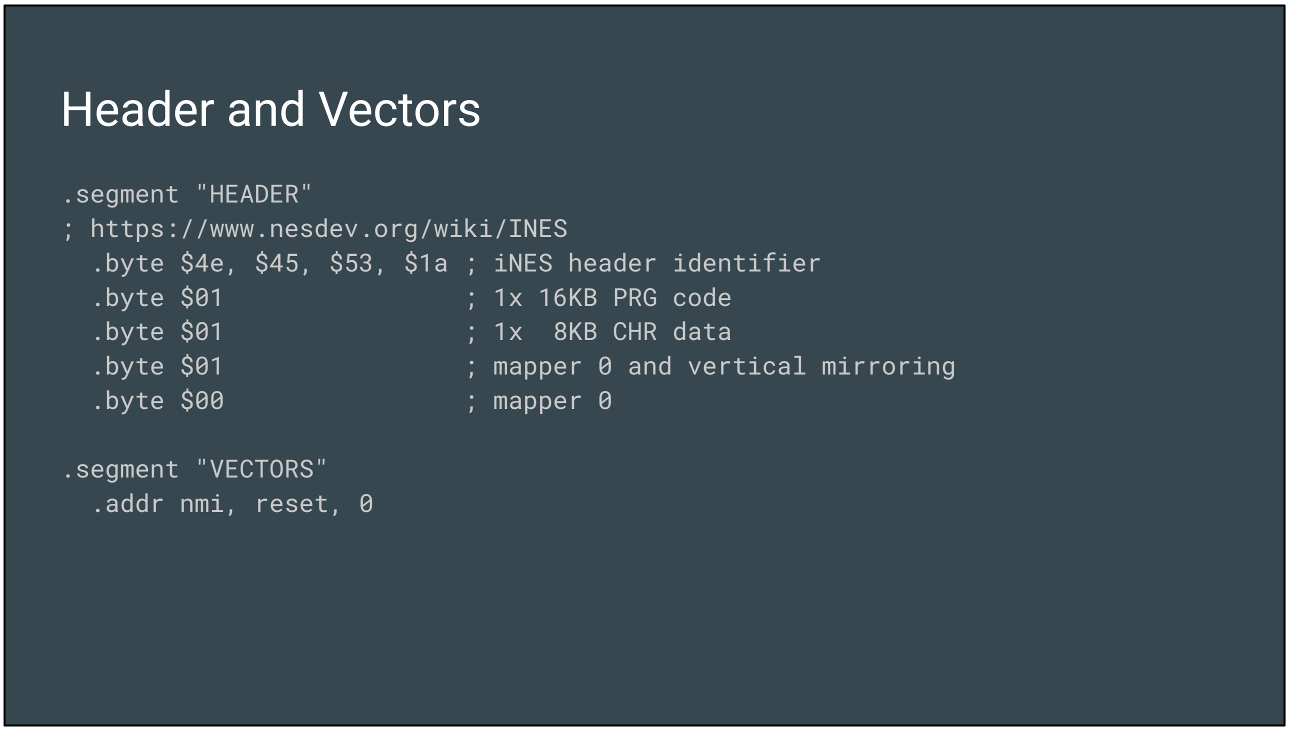


Memory Map

- 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
- Unused: $0300 - $07ff

OAM - Object Attribute Memory

Finally… let’s look at some code!



Header and Vectors

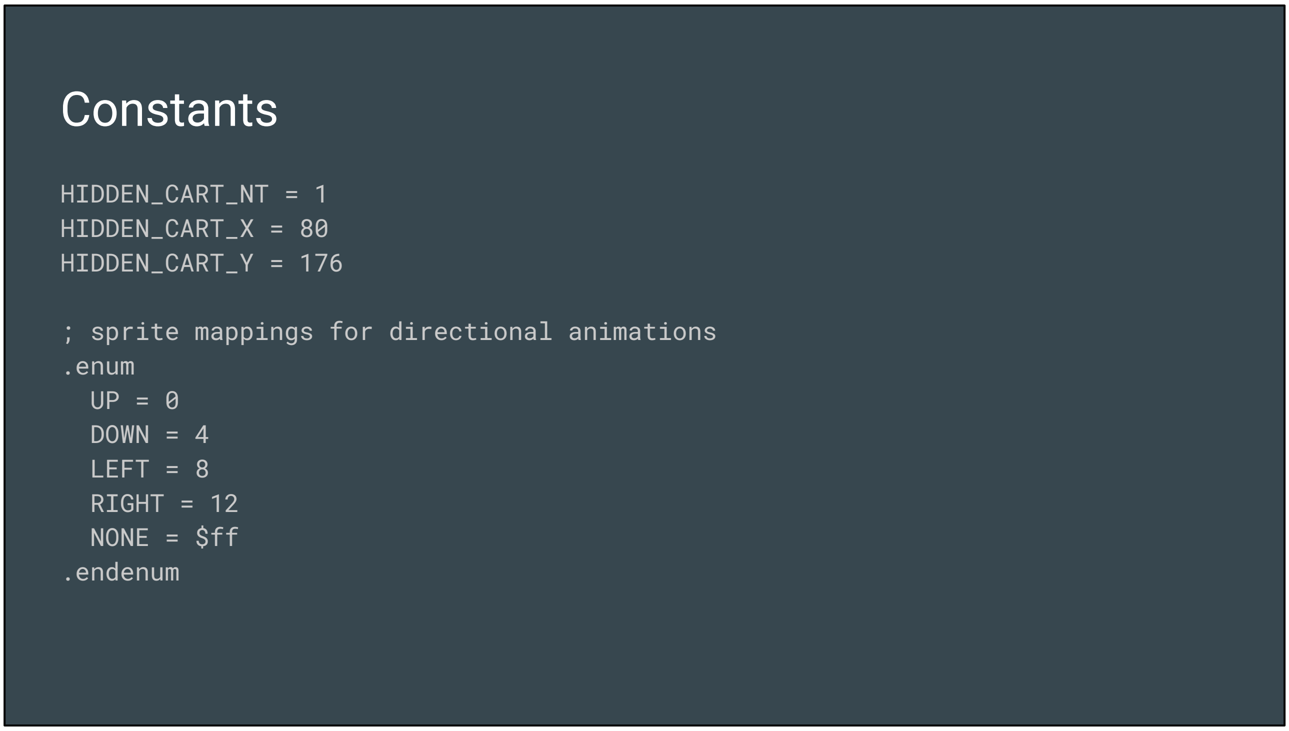
```
.segment "HEADER"
; https://www.nesdev.org/wiki/INES
  .byte $4e, $45, $53, $1a ; iNES header identifier
  .byte $01                ; 1x 16KB PRG code
  .byte $01                ; 1x  8KB CHR data
  .byte $01                ; mapper 0 and vertical mirroring
  .byte $00                ; mapper 0

.segment "VECTORS"
  .addr nmi, reset, 0
```

Segments are configured in the linker configuration file. The example project contains a config which describes the NROM format.
IRQ will never be called, so its value doesn’t matter.

Zero Page Variables

```
.segment "ZEROPAGE"
nametable_ptr: .res 2

joy1_current: .res 1
joy1_previous: .res 1

ppu_ctrl: .res 1
scroll_x: .res 1
scroll_y: .res 1

walk_cycle_sprite: .res 1
sprite_timer: .res 1

show_bubble: .res 1

waiting_for_nmi: .res 1
```

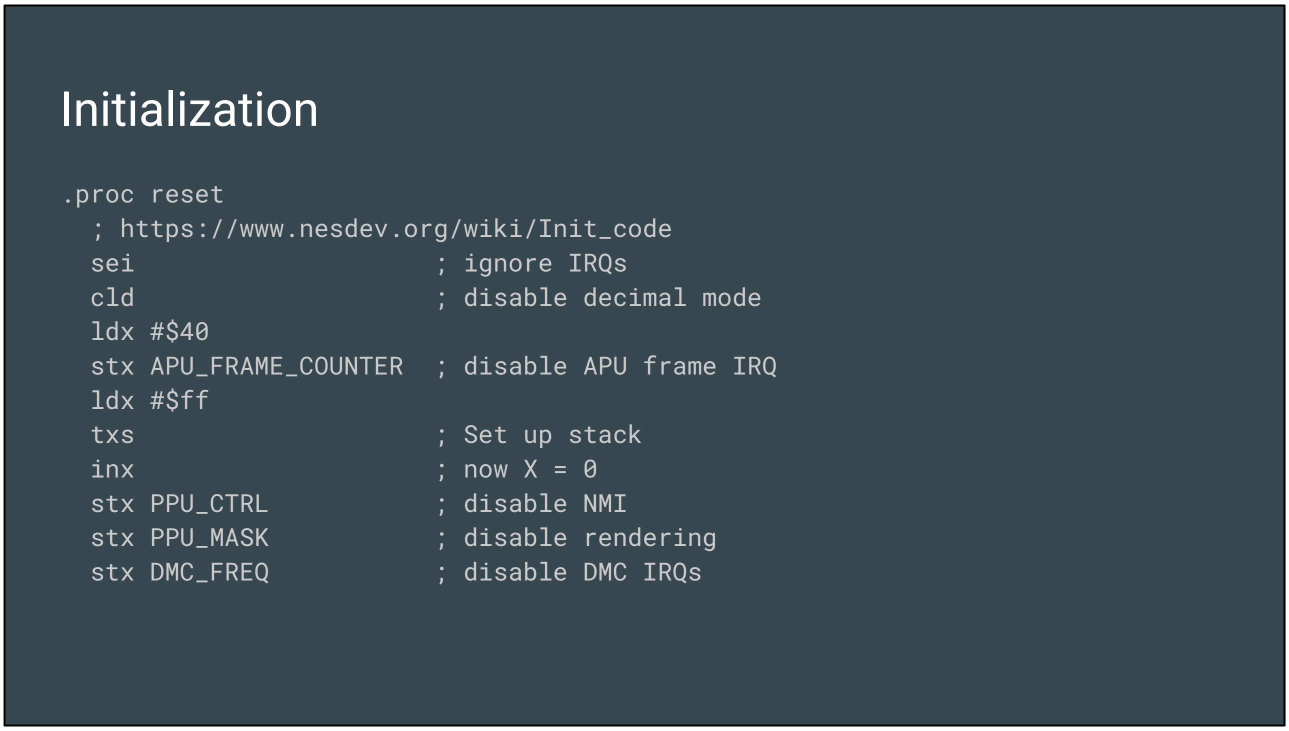
Here we let the linker sequentially place our variables in memory to avoid collisions.
“.res n” means reserve n bytes for each field.



Constants

```
HIDDEN_CART_NT = 1
HIDDEN_CART_X = 80
HIDDEN_CART_Y = 176

; sprite mappings for directional animations
.enum
  UP = 0
  DOWN = 4
  LEFT = 8
  RIGHT = 12
  NONE = $ff
.endenum
```

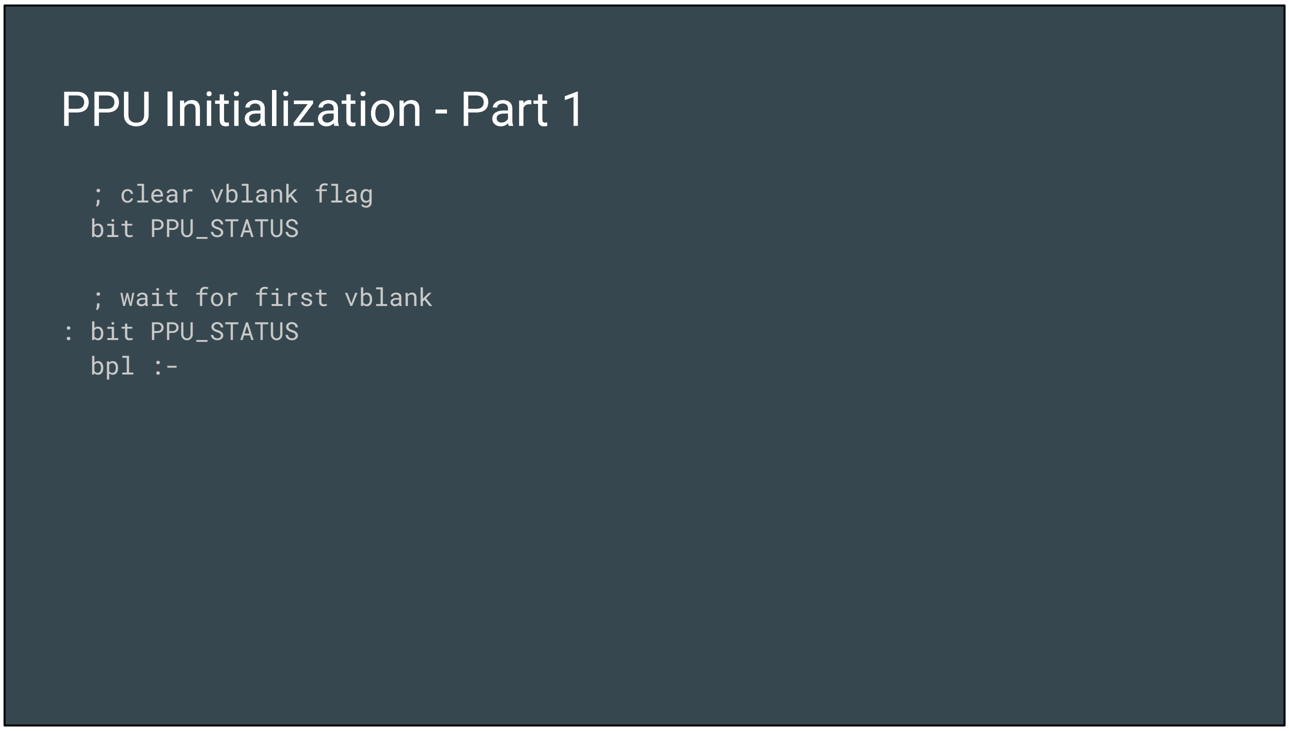
These constants will be replaced by ca65 at build-time, so there is no run-time penalty.
They can help make logic more readable.
Grouped constants are called enums, short for enumerations.



Initialization

```
.proc reset
  ; https://www.nesdev.org/wiki/Init_code
  sei                     ; ignore IRQs
  cld                     ; disable decimal mode
  ldx #$40
  stx APU_FRAME_COUNTER   ; disable APU frame IRQ
  ldx #$ff
  txs                     ; Set up stack
  inx                     ; now X = 0
  stx PPU_CTRL            ; disable NMI
  stx PPU_MASK            ; disable rendering
  stx DMC_FREQ            ; disable DMC IRQs
```

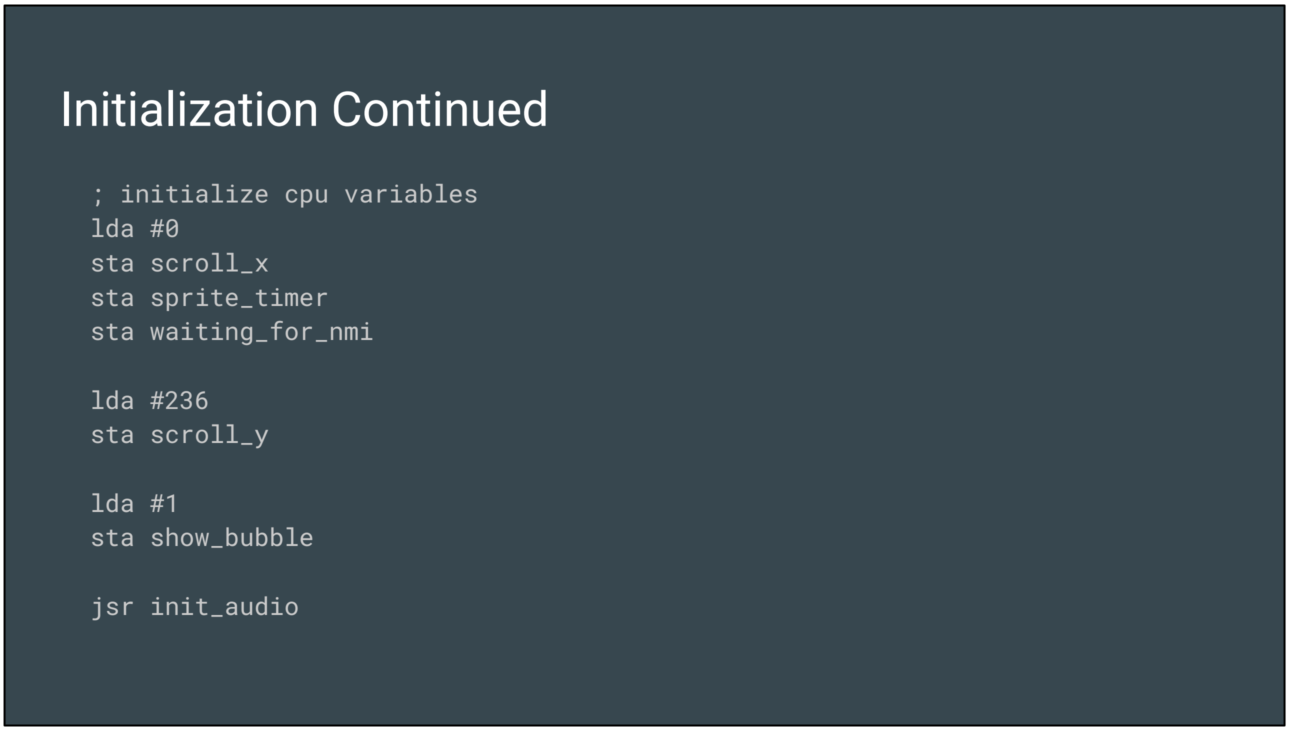
Standard NES initialization, used by many retail games.



PPU Initialization - Part 1

```
  ; clear vblank flag
  bit PPU_STATUS

  ; wait for first vblank
: bit PPU_STATUS
  bpl :-
```

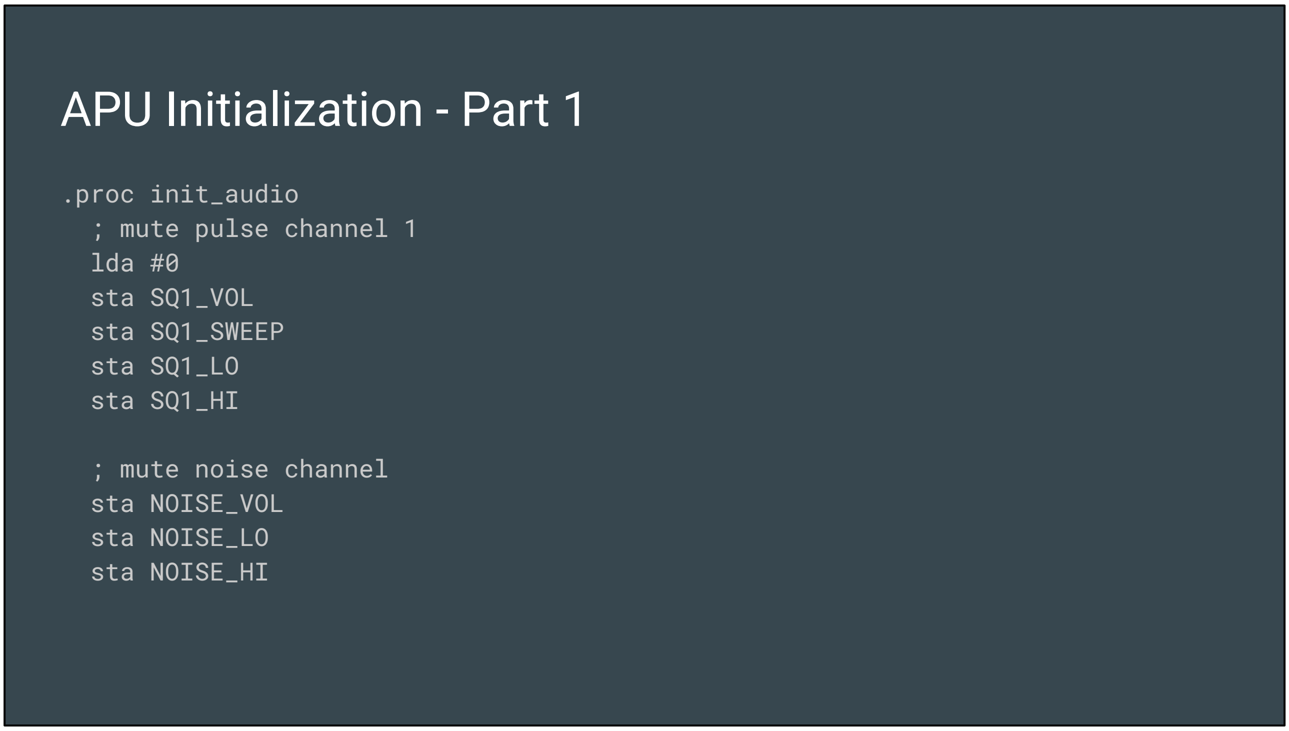
The PPU takes around two complete frames to “warm up.”
Clearing the VBlank flag ensures we don’t skip the first frame.
The bit instruction copies the value of PPU_STATUS to the processor status flags for easy branching.



Initialization Continued

```
; initialize cpu variables
lda #0
sta scroll_x
sta sprite_timer
sta waiting_for_nmi

lda #236
sta scroll_y

lda #1
sta show_bubble

jsr init_audio
```

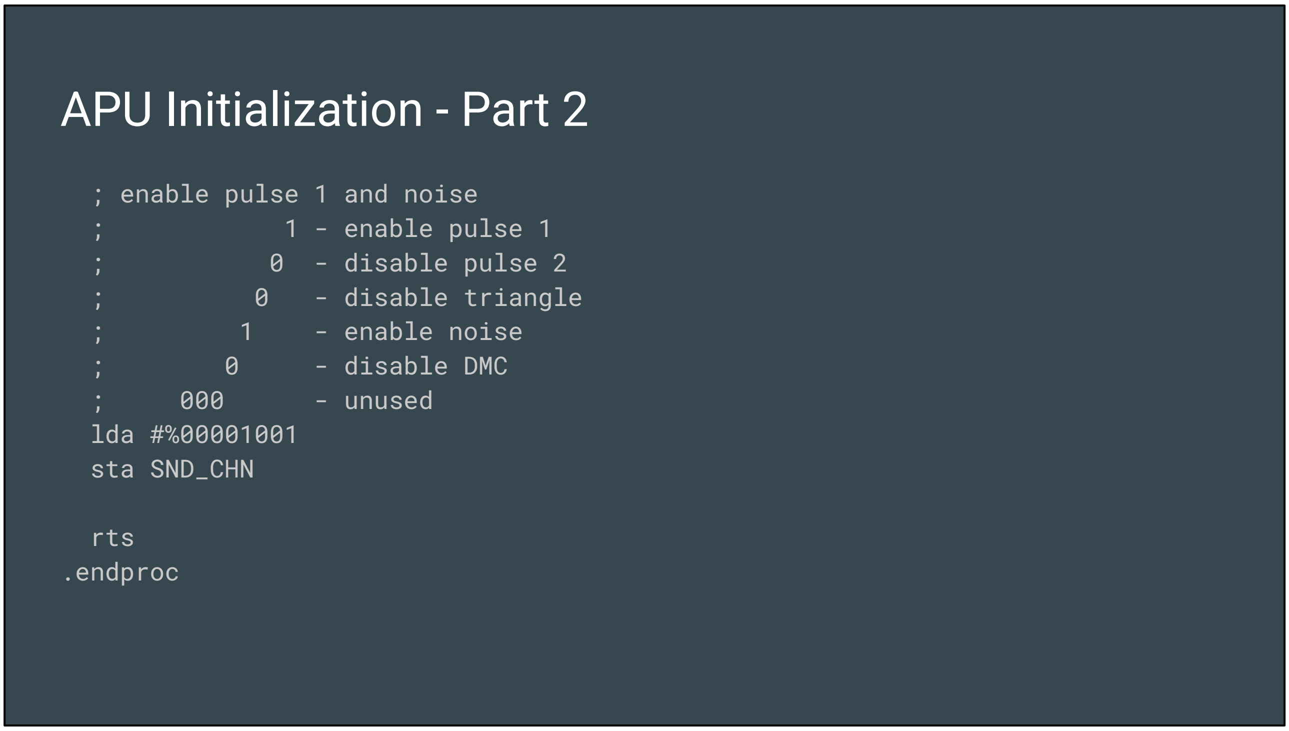
Here we initialize our gameplay variables and then jump to the “init_audio” subroutine.



APU Initialization - Part 1

```
.proc init_audio
  ; mute pulse channel 1
  lda #0
  sta SQ1_VOL
  sta SQ1_SWEEP
  sta SQ1_LO
  sta SQ1_HI

  ; mute noise channel
  sta NOISE_VOL
  sta NOISE_LO
  sta NOISE_HI
```

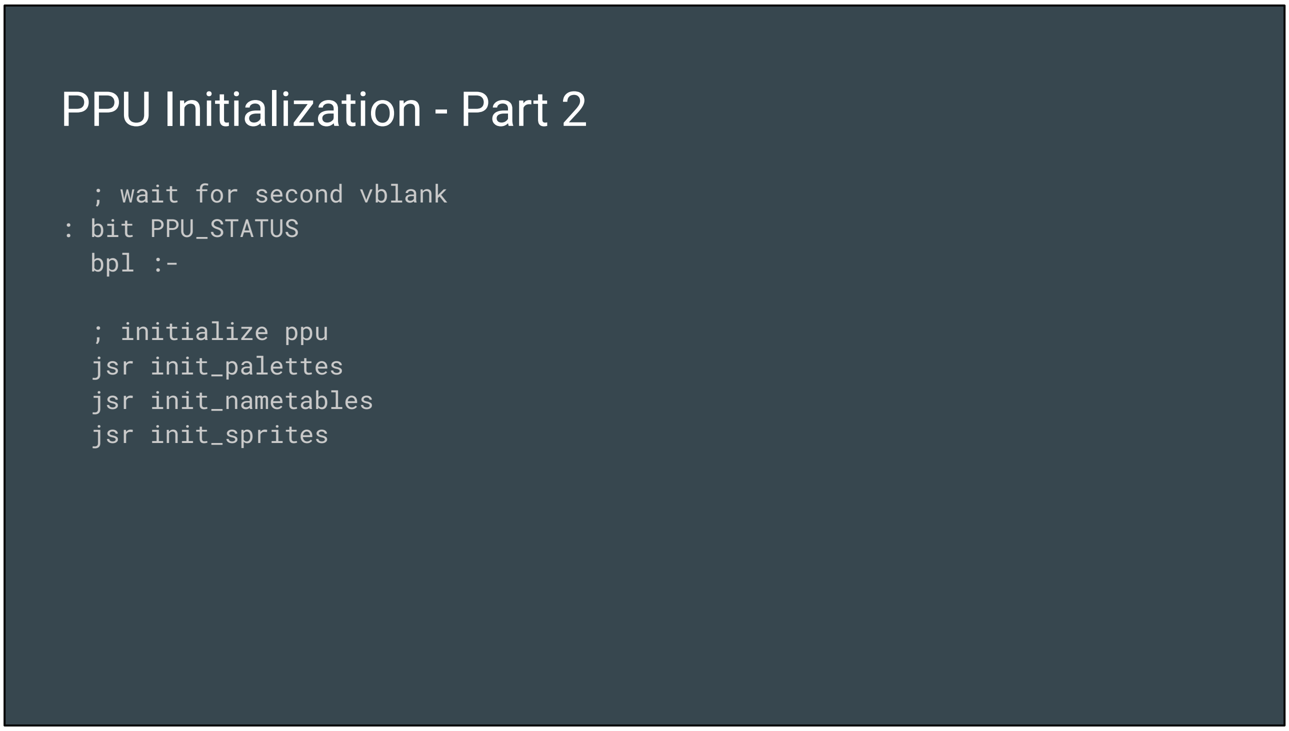
Clear out the controls for the first pulse channel and the noise channel.
These will be used for the “success” and “failure” sounds during gameplay.



APU Initialization - Part 2

```
  ; enable pulse 1 and noise
  ;            1 - enable pulse 1
  ;           0  - disable pulse 2
  ;          0   - disable triangle
  ;         1    - enable noise
  ;        0     - disable DMC
  ;     000      - unused
  lda #%00001001
  sta SND_CHN

  rts
.endproc
```

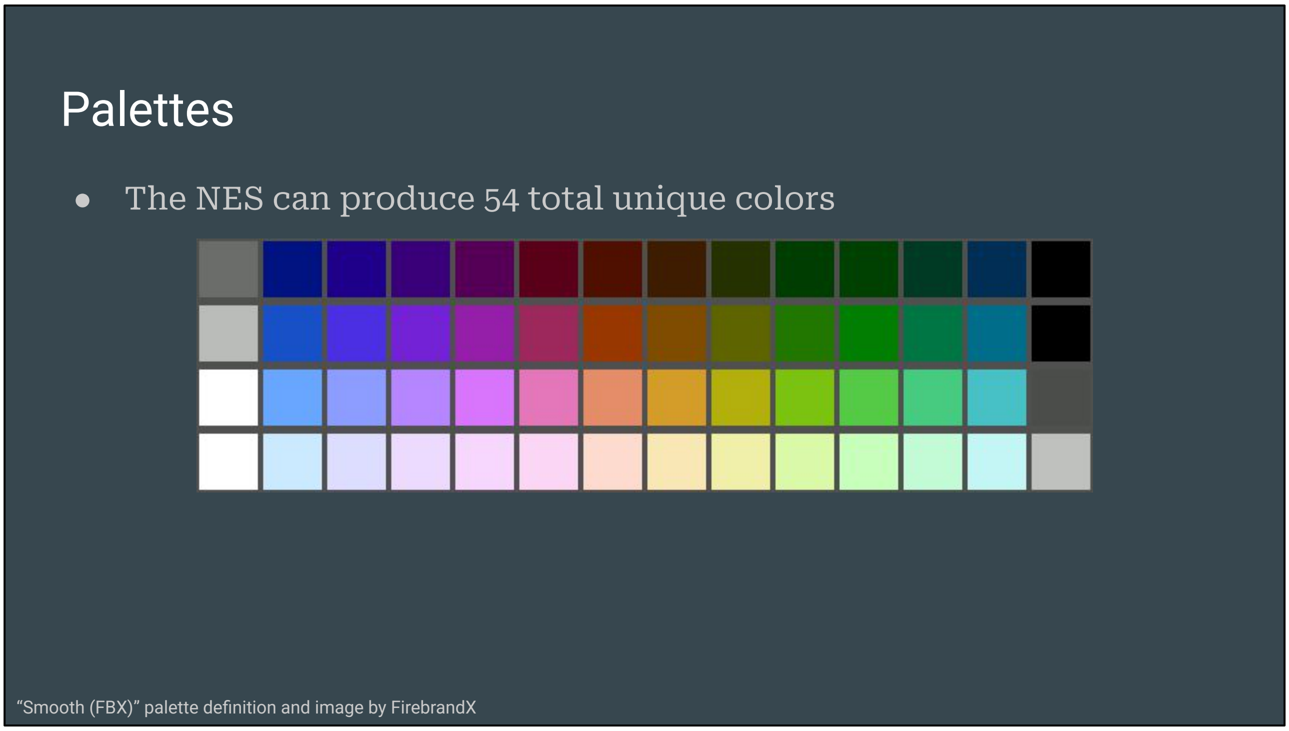
Enable the channels we plan to use in gameplay by setting the correct bits in SND_CHN.
Return from Subroutine to get back to our reset initialization code.



PPU Initialization - Part 2

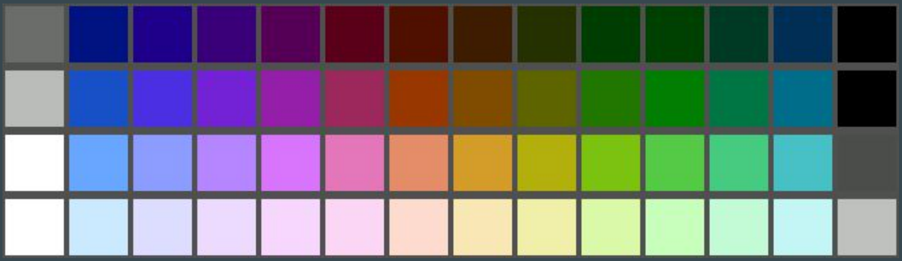
```
  ; wait for second vblank
: bit PPU_STATUS
  bpl :-

  ; initialize ppu
  jsr init_palettes
  jsr init_nametables
  jsr init_sprites
```

After the second VBlank, we can finally initialize our graphics.



Palettes

- The NES can produce 54 total unique colors

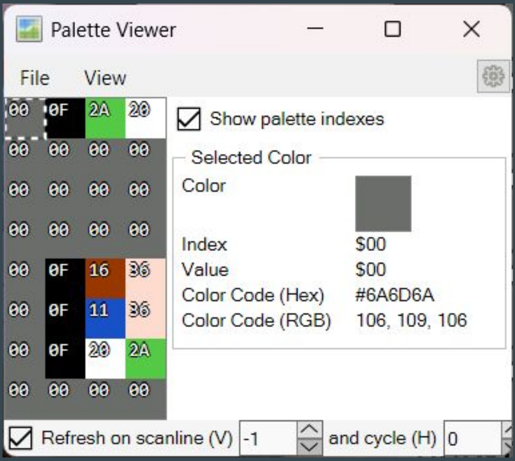


"Smooth (FBX)" palette definition and image by FirebrandX

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 as 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



00	0F	2A	20
00	00	00	00
00	00	00	00
00	00	00	00
00	0F	16	36
00	0F	11	36
00	0F	20	2A
00	00	00	00

[✓] Show palette indexes

Selected Color

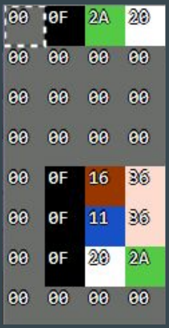
Color	
Index	$00
Value	$00
Color Code (Hex)	#6A6D6A
Color Code (RGB)	106, 109, 106

[✓] Refresh on scanline (V) -1 and cycle (H) 0

Palette Viewer inside Mesen

Example Palette Data

```
palettes:
; https://www.nesdev.org/wiki/PPU_palettes
  ; background palettes
  .byte $00, $0f, $2a, $20
  .byte $00, $00, $00, $00
  .byte $00, $00, $00, $00
  .byte $00, $00, $00, $00

  ; sprite palettes
  .byte $00, $0f, $16, $36
  .byte $00, $0f, $11, $36
  .byte $00, $0f, $20, $2a
  .byte $00, $00, $00, $00
```



```
00 0F 2A 20
00 00 00 00
00 00 00 00
00 00 00 00
00 0F 16 36
00 0F 11 36
00 0F 20 2A
00 00 00 00
```

Found in “data.inc”

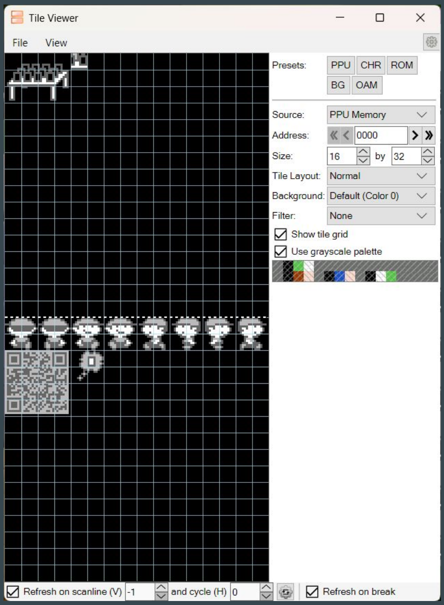
Palette Initialization

```
  ; set ppu address to palette entries ($3f00)
  lda #$3f
  sta PPU_ADDR
  lda #0
  sta PPU_ADDR

  ; loop through each palette entry, 32 total
  ldx #0
: lda palettes, x
  sta PPU_DATA
  inx
  cpx #32
  bne :-
```

This is a snippet from “init_palettes.”
Destination address is updated via a pair of writes to PPU_ADDR, since all addresses are 16-bit.
This process can be reset by reading from PPU_STATUS, but is unnecessary here.

Pattern Tables

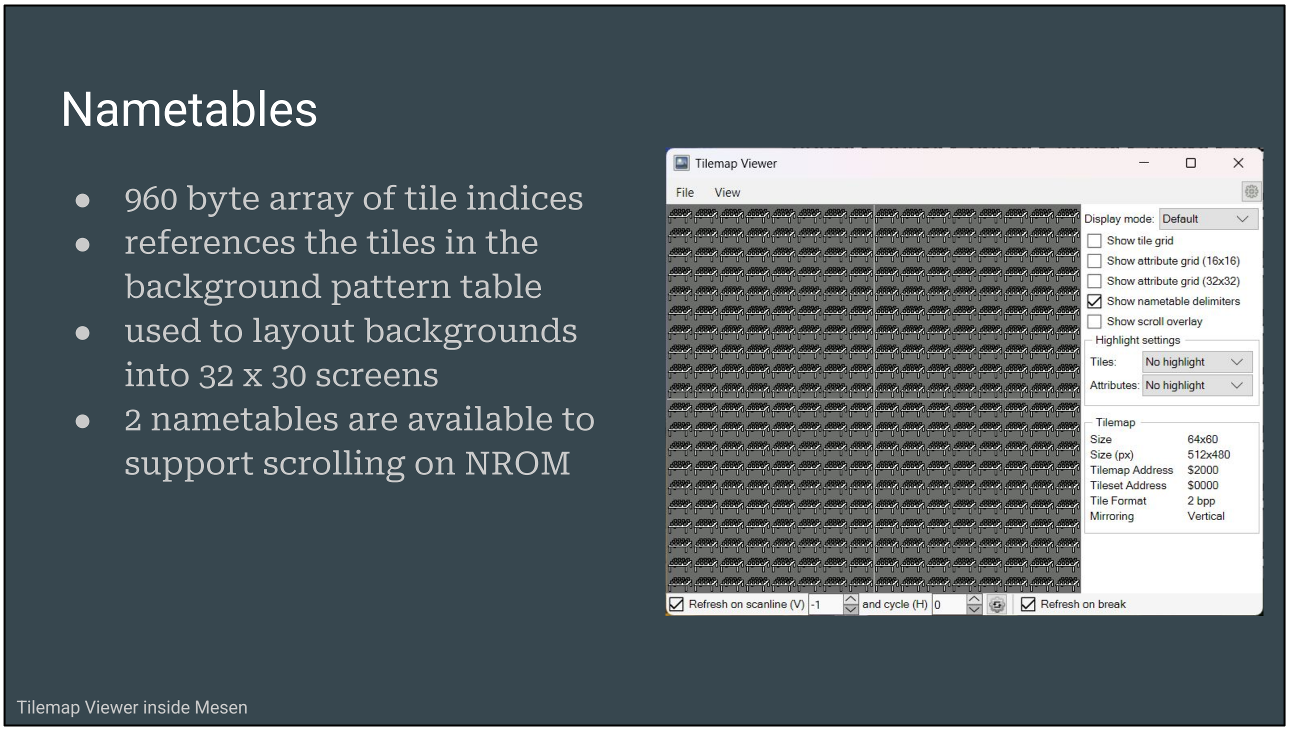
- 1KB array of tiles
- stored as packed indices into a palette
- 2 pattern tables are available to store graphics on NROM



Source: PPU Memory
Address: 0000
Size: 16 by 32
Tile Layout: Normal
Background: Default (Color 0)
Filter: None
☑ Show tile grid
☑ Use grayscale palette
☑ Refresh on scanline (V) -1 and cycle (H) 0
☑ Refresh on break

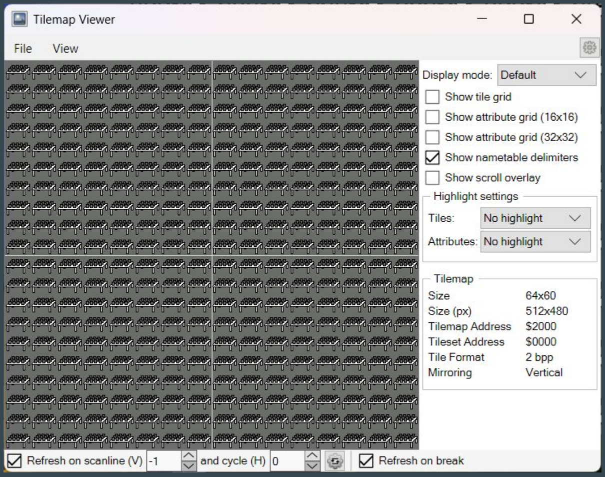
Tile Viewer inside Mesen

Backgrounds and sprites can reference the same or separate pattern tables. There are various tools available to help format the tile data, which will be shared towards the end of this talk.



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



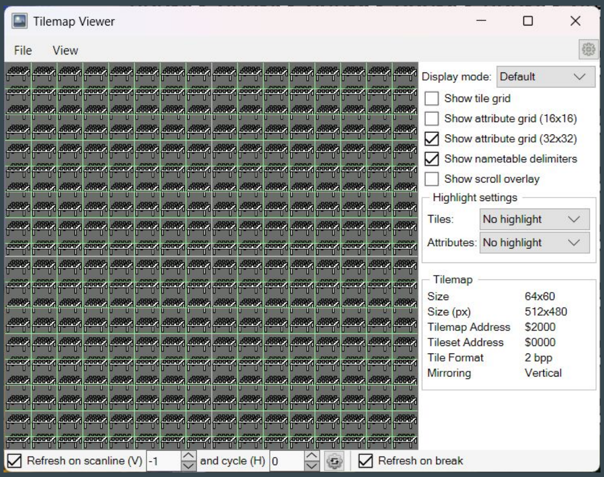
Tilemap	
Size	64x60
Size (px)	512x480
Tilemap Address	$2000
Tileset Address	$0000
Tile Format	2 bpp
Mirroring	Vertical

Tilemap Viewer inside Mesen

There are various tools available to help format the tile data, which will be shared towards the end of this talk.
We will be using vertical mirroring, which 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



Display mode: Default

- [ ] Show tile grid
- [ ] Show attribute grid (16x16)
- [✓] Show attribute grid (32x32)
- [✓] Show nametable delimiters
- [ ] Show scroll overlay

Highlight settings

Tiles: No highlight
Attributes: No highlight

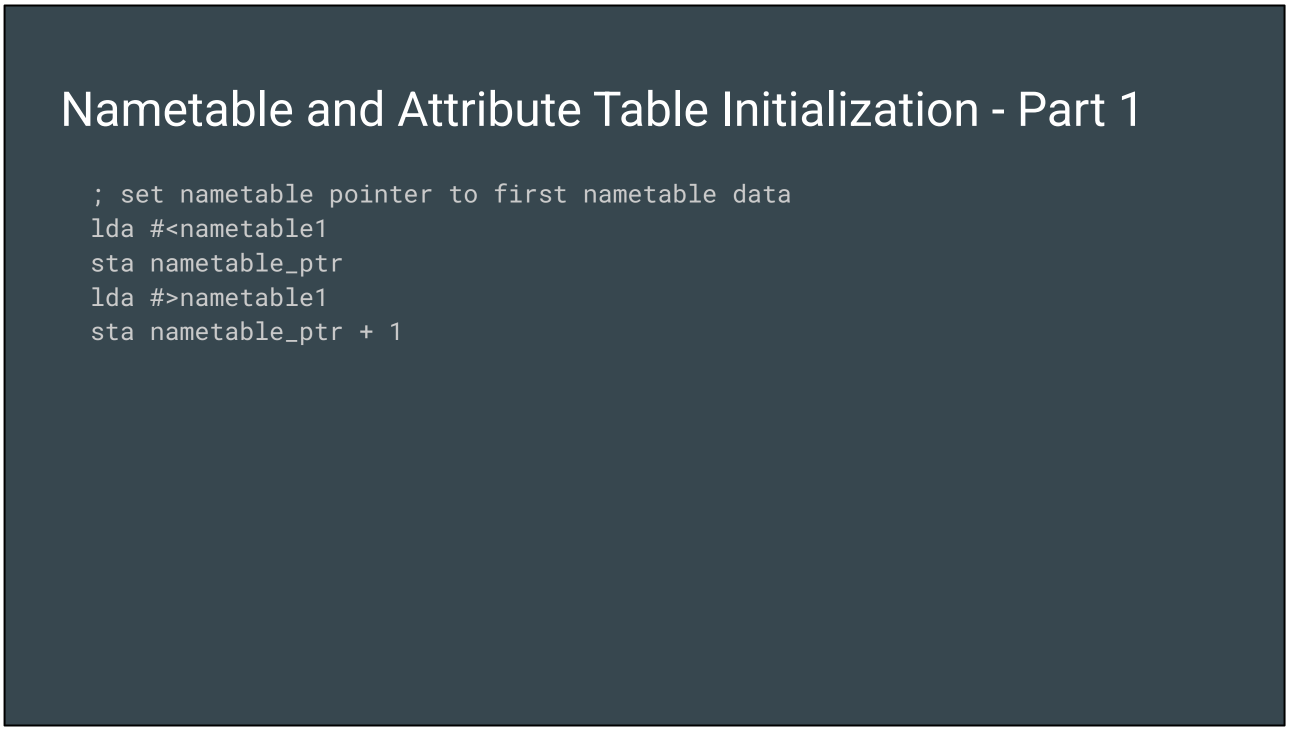
Tilemap

Size	64x60
Size (px)	512x480
Tilemap Address	$2000
Tileset Address	$0000
Tile Format	2 bpp
Mirroring	Vertical

[✓] Refresh on scanline (V) -1 and cycle (H) 0 [✓] Refresh on break

Tilemap Viewer inside Mesen

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



Nametable and Attribute Table Initialization - Part 1

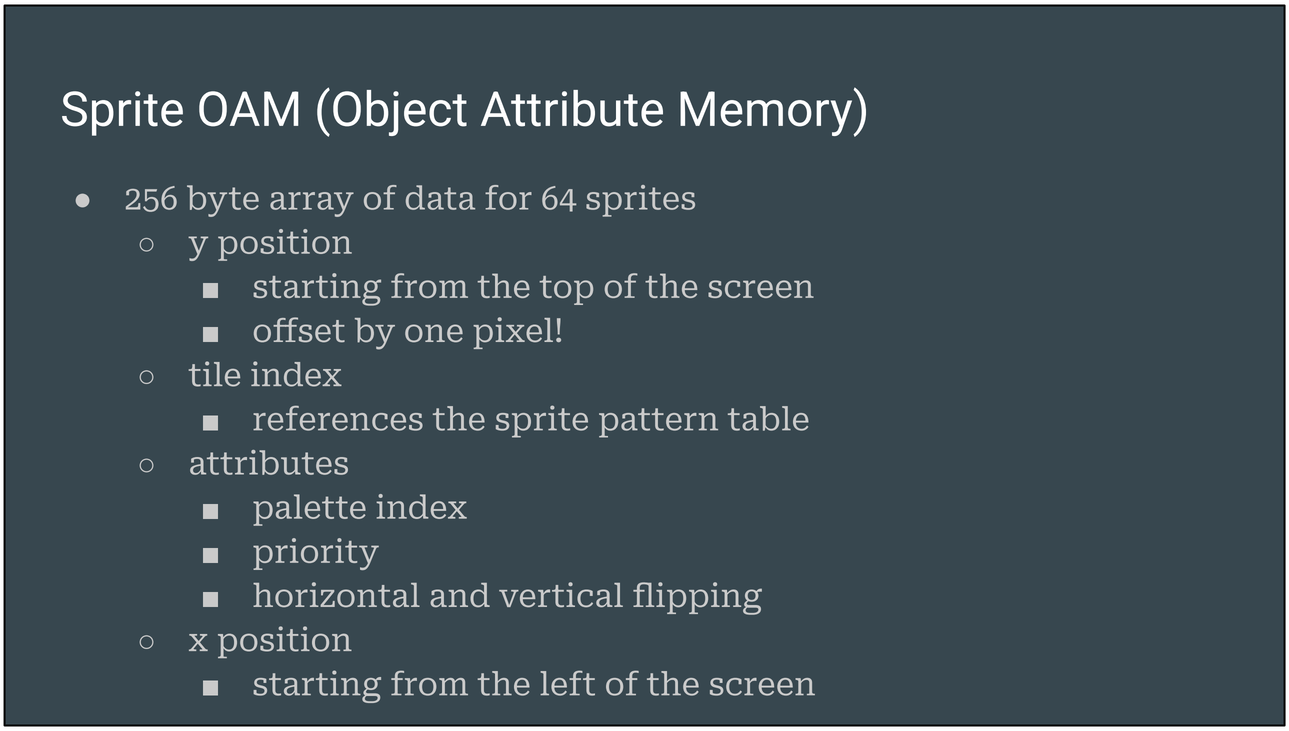
```
; set nametable pointer to first nametable data
lda #<nametable1
sta nametable_ptr
lda #>nametable1
sta nametable_ptr + 1
```

This is a snippet from “init_nametables.”
Pointers are a huge topic and confusing to many new programmers, which we will unfortunately gloss over today. Just know that instead of holding data, they “point” to other variables which hold the data.
Fortunately, this code can be reused to load any nametable, regardless of contents.

Nametable and Attribute Table Initialization - Part 2

```
  ldx #4
  ldy #0
  ; use nametable pointer + offset to look up next tile
: lda (nametable_ptr), y
  sta PPU_DATA

  iny
  bne :-

  ; move pointer to next 256 tiles
  inc nametable_ptr + 1
  dex
  bne :-
```

This is a snippet from “init_nametables.”
x and y are used together here as nested loops, for 4 * 256 = 1024 iterations total.
Remember, nametable + attribute table size is 960 + 64 = 1024.

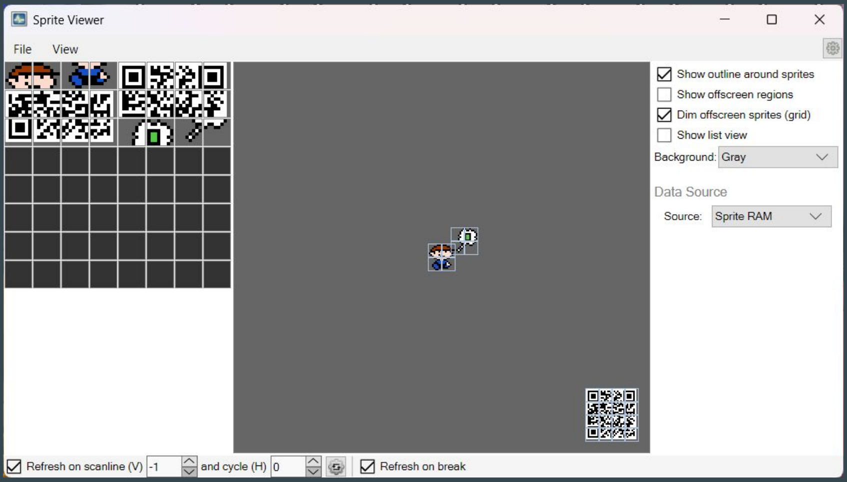


Sprite OAM (Object Attribute Memory)

- 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)

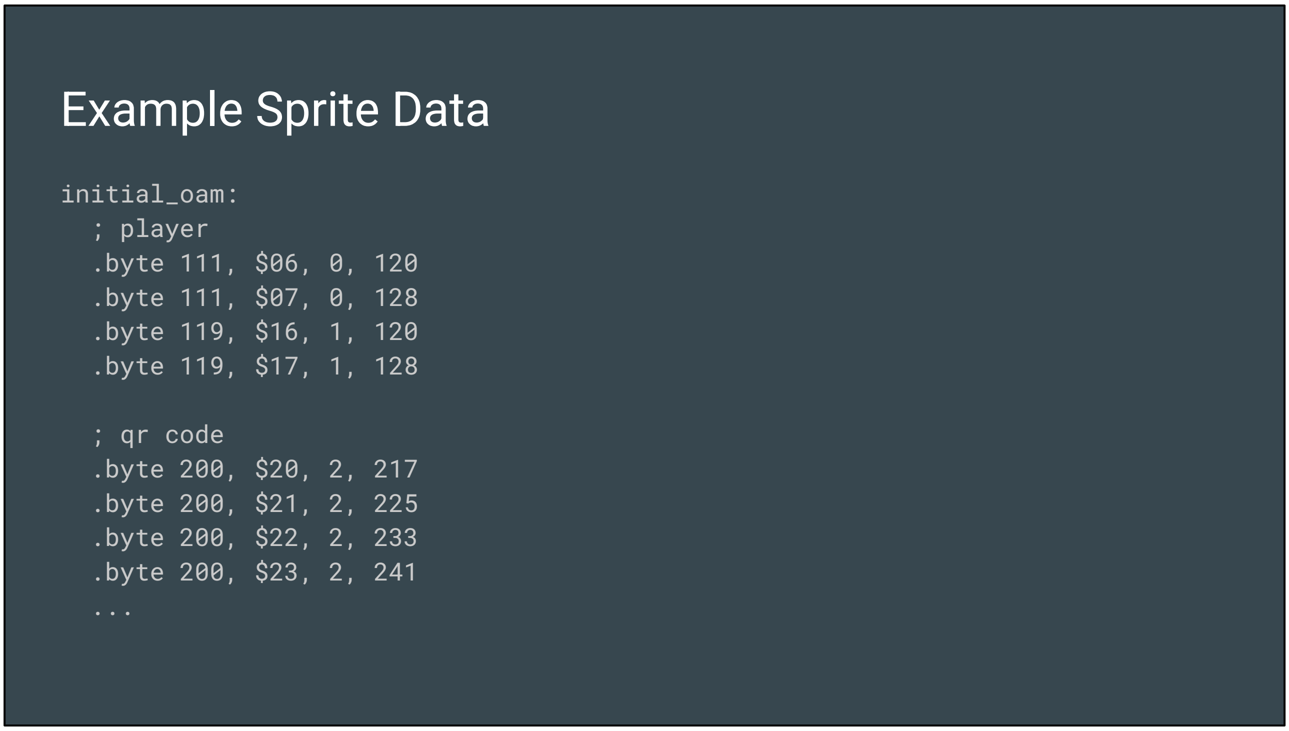


[✓] Show outline around sprites
[ ] Show offscreen regions
[✓] Dim offscreen sprites (grid)
[ ] Show list view
Background: Gray

Data Source
Source: Sprite RAM

[✓] Refresh on scanline (V) -1 and cycle (H) 0 [✓] Refresh on break

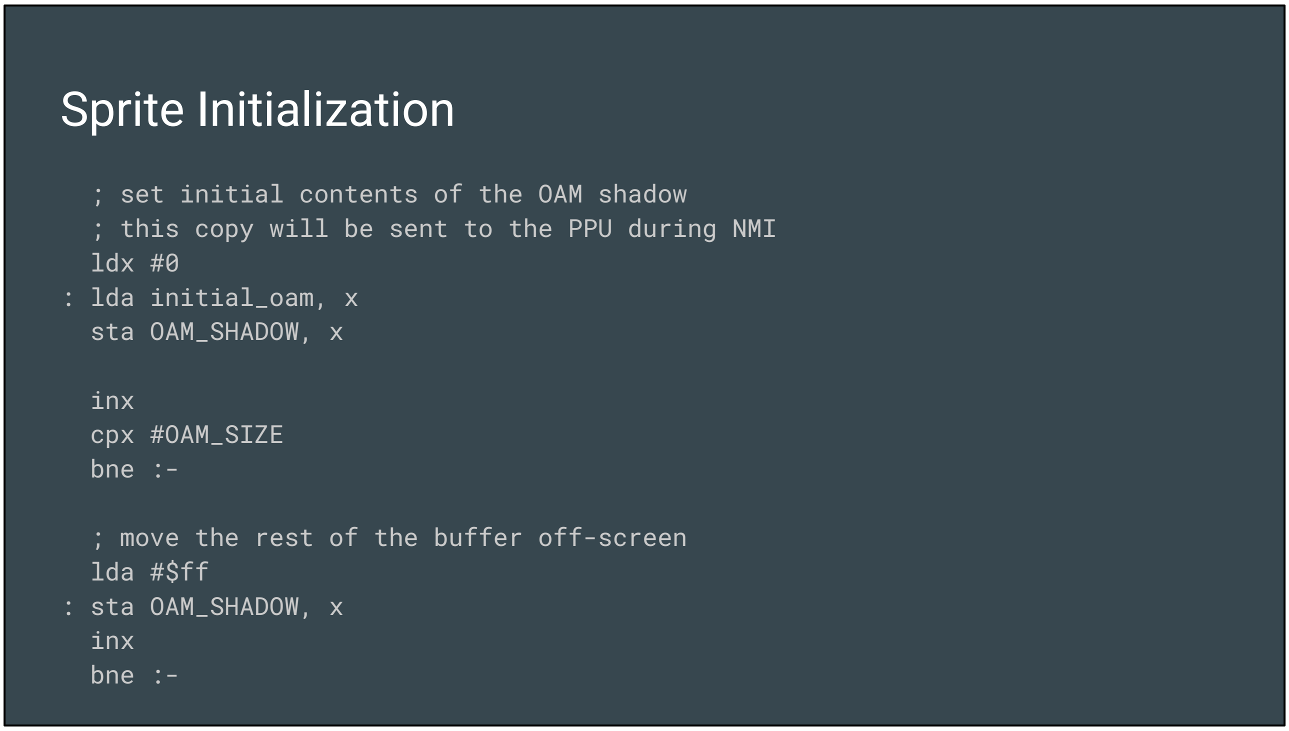
Sprite Viewer inside Mesen



Example Sprite Data

```
initial_oam:
  ; player
  .byte 111, $06, 0, 120
  .byte 111, $07, 0, 128
  .byte 119, $16, 1, 120
  .byte 119, $17, 1, 128

  ; qr code
  .byte 200, $20, 2, 217
  .byte 200, $21, 2, 225
  .byte 200, $22, 2, 233
  .byte 200, $23, 2, 241
  ...
```

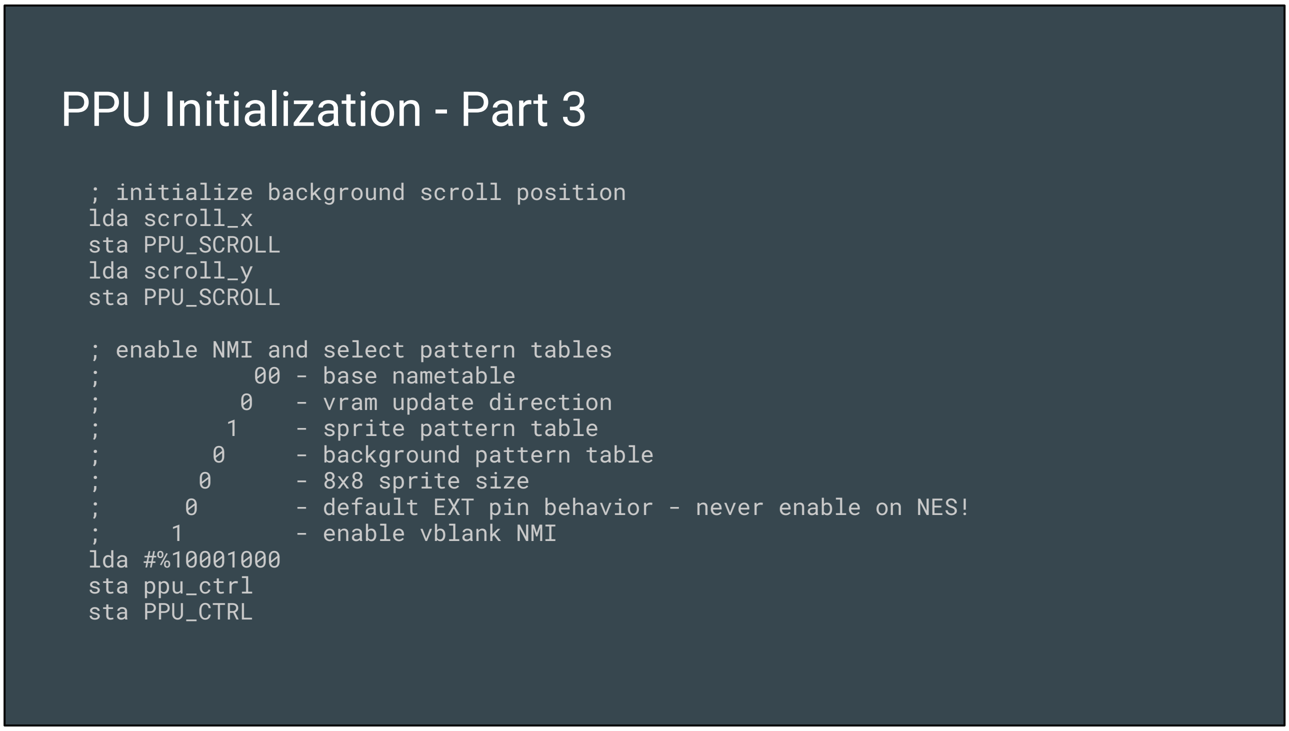
Found in “data.inc”
Note that you can mix and match number systems here, to improve clarity.



Sprite Initialization

```
  ; set initial contents of the OAM shadow
  ; this copy will be sent to the PPU during NMI
  ldx #0
: lda initial_oam, x
  sta OAM_SHADOW, x

  inx
  cpx #OAM_SIZE
  bne :-

  ; move the rest of the buffer off-screen
  lda #$ff
: sta OAM_SHADOW, x
  inx
  bne :-
```

This is a snippet from “init_sprites.”
Setting just the y position to $ff is sufficient for moving an usused sprite off-screen, but it’s actually faster to set all fields instead of advancing the x offset by 4.



PPU Initialization - Part 3

```
; initialize background scroll position
lda scroll_x
sta PPU_SCROLL
lda scroll_y
sta PPU_SCROLL

; enable NMI and select pattern tables
;           00 - base nametable
;          0   - vram update direction
;         1    - sprite pattern table
;        0     - background pattern table
;       0      - 8x8 sprite size
;      0       - default EXT pin behavior - never enable on NES!
;     1        - enable vblank NMI
lda #%10001000
sta ppu_ctrl
sta PPU_CTRL
```

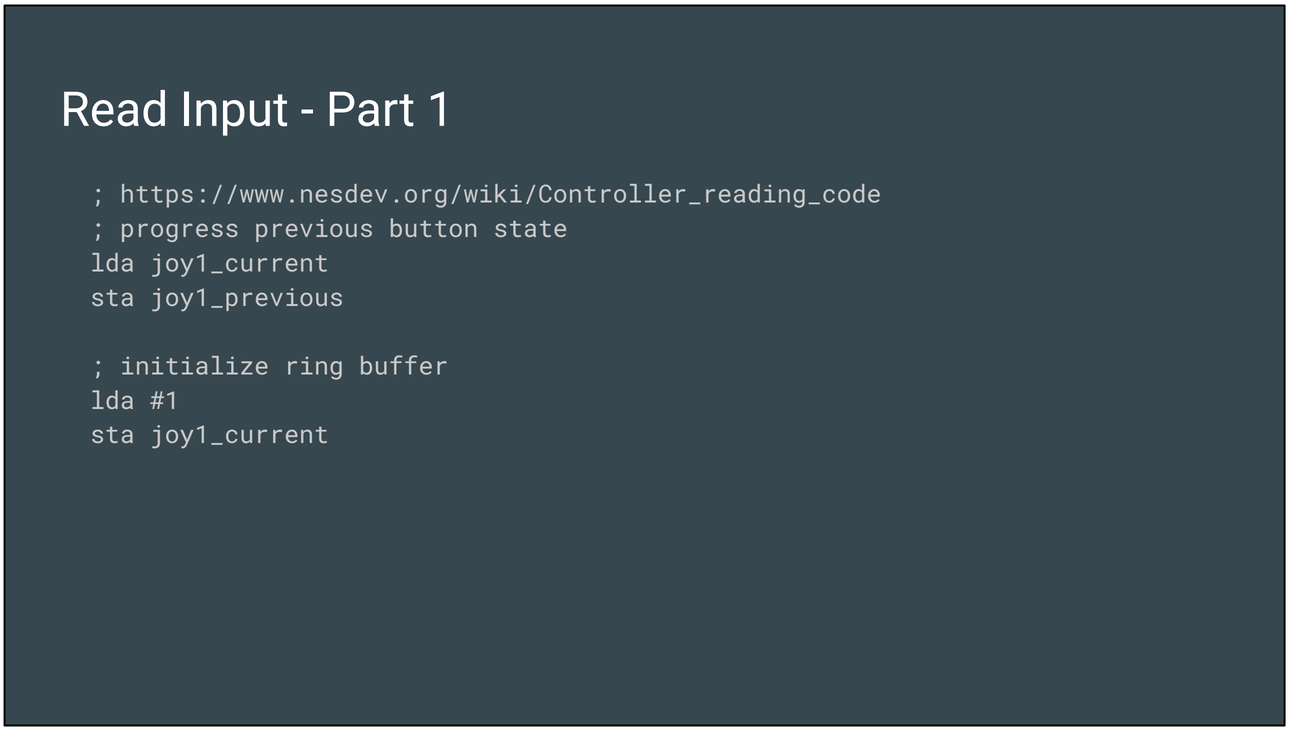
It’s always good practice to reset PPU scroll position after we modify PPU memory to avoid graphical glitches.
After that, we can finally enable the VBlank NMI.

Finally… initialization is complete!

Game Loop

```
game_loop:
  ; wait for frame to be completed
  inc waiting_for_nmi
: lda waiting_for_nmi
  bne :-

  jsr handle_input
  jsr update_animation

  jmp game_loop
```

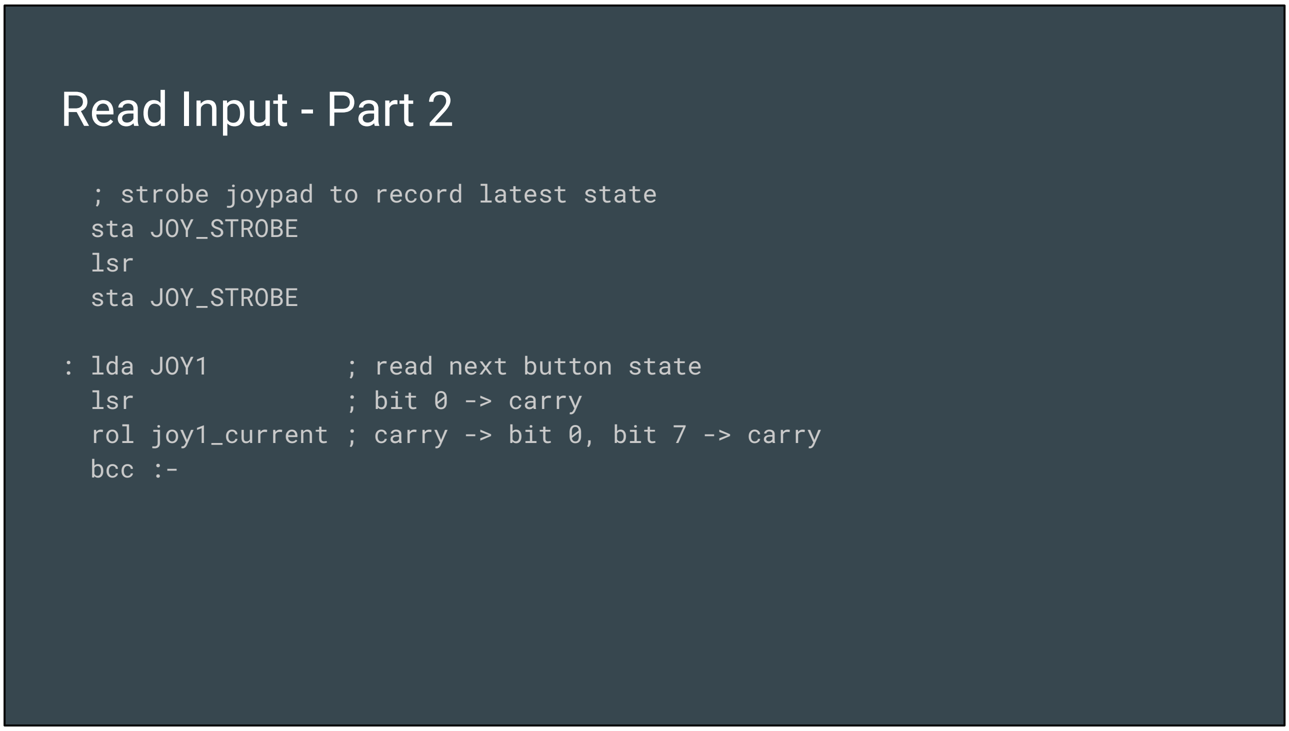
NMI will eventually store 0 into waiting_for_nmi.
After that, we can continue to handle_input, which immediately calls read_joypad.



Read Input - Part 1

```
; https://www.nesdev.org/wiki/Controller_reading_code
; progress previous button state
lda joy1_current
sta joy1_previous

; initialize ring buffer
lda #1
sta joy1_current
```

We'll move through this quickly, as it's more standardized code from [nesdev.org](https://www.nesdev.org).
We want to keep track of the previous button state, so we can detect when a button was pressed instead of just "up" or "down."
Initializing joy1_current to 1 is setting up an optimization in the next code snippet.



Read Input - Part 2

```
  ; strobe joypad to record latest state
  sta JOY_STROBE
  lsr
  sta JOY_STROBE

: lda JOY1         ; read next button state
  lsr              ; bit 0 -> carry
  rol joy1_current ; carry -> bit 0, bit 7 -> carry
  bcc :-
```

As mentioned previously, we strobe the controller so it updates the button states.
We then read each button state, one at a time, and through clever use of the carry bit, transfer these states into joy1_current.
It’s worth mentioning that if we wanted to capture anything other than the standard NES controller input, like a zapper, this carry trick could not be used.

Handle Input Example Case

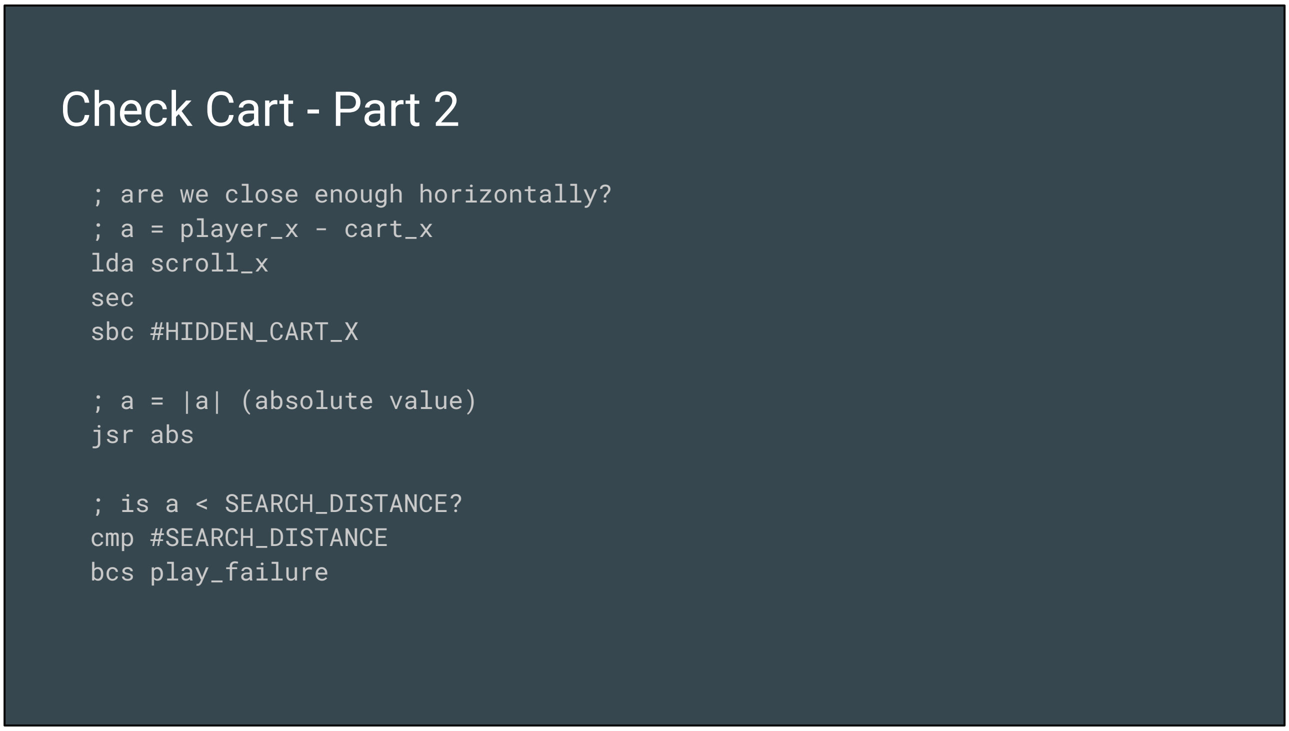
```
  ; is A currently pressed?
  lda joy1_current
  and #BUTTON_A
  beq :+

  ; was A just pressed?
  lda joy1_previous
  and #BUTTON_A
  bne :+

  jsr check_cart

  ; is B currently pressed?
: lda joy1_current
```

handle_input then checks the updated button states, along with the previous button states, to identify button presses.
Each button press has an associated subroutine to handle the event.
If the button isn't down, or has been down, this subroutine call is skipped.

Check Cart - Part 1

```
; are we in the right nametable?
lda ppu_ctrl
and #1

cmp #HIDDEN_CART_NT
bne play_failure
```

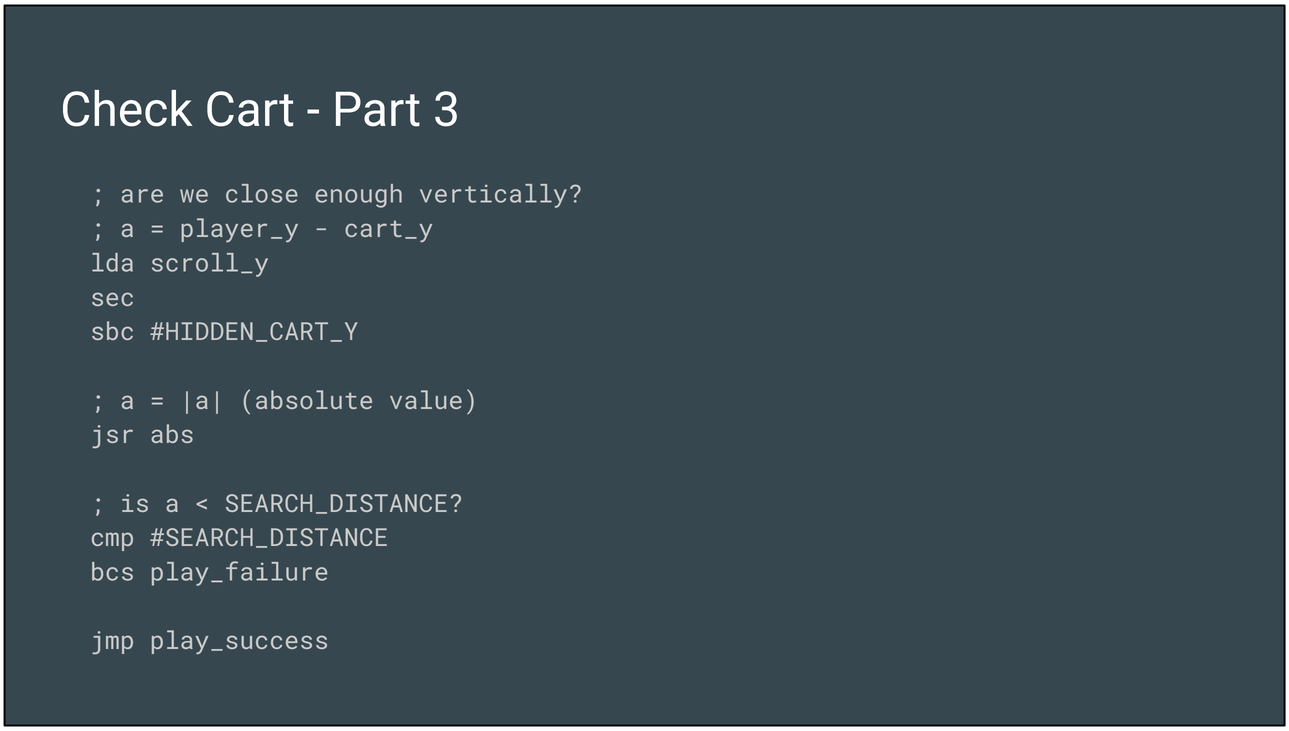
There are several checks to see if we’ve succeeded in finding our dream cart.
Recall that nametables can be thought of as screens, so we first check if we’re on the correct “screen.”



Check Cart - Part 2

```
; are we close enough horizontally?
; a = player_x - cart_x
lda scroll_x
sec
sbc #HIDDEN_CART_X

; a = |a| (absolute value)
jsr abs

; is a < SEARCH_DISTANCE?
cmp #SEARCH_DISTANCE
bcs play_failure
```

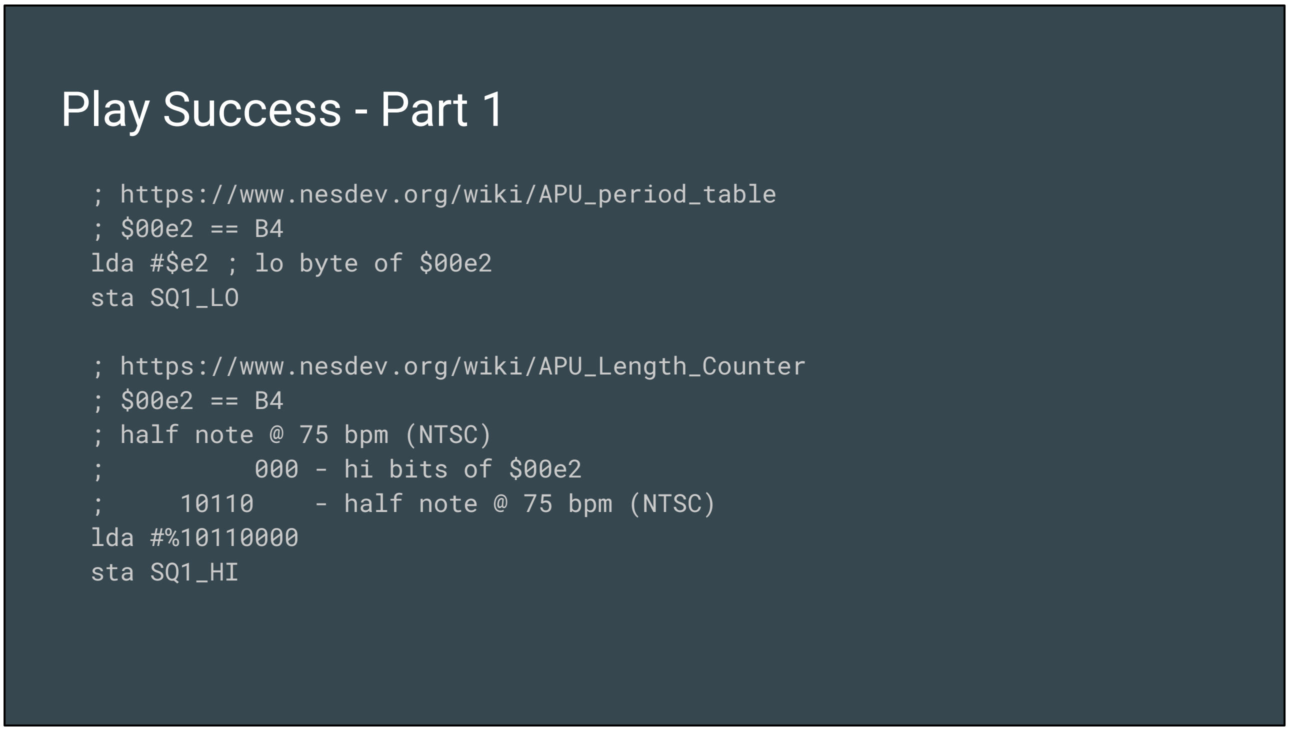
Checking the actual distance from the player to the cart would require an expensive, handwritten square root operation, so we will only check Manhattan distance.
Taking the absolute value of the horizontal distance uses two's complement, which we unfortunately don't have time to get into today, but the source code includes a link to wikipedia's explanation if you'd like to dig into it.
Finally, we use cmp and check the carry bit to determine if horizontal distance is less than the constant SEARCH_DISTANCE.



Check Cart - Part 3

```
; are we close enough vertically?
; a = player_y - cart_y
lda scroll_y
sec
sbc #HIDDEN_CART_Y

; a = |a| (absolute value)
jsr abs

; is a < SEARCH_DISTANCE?
cmp #SEARCH_DISTANCE
bcs play_failure

jmp play_success
```

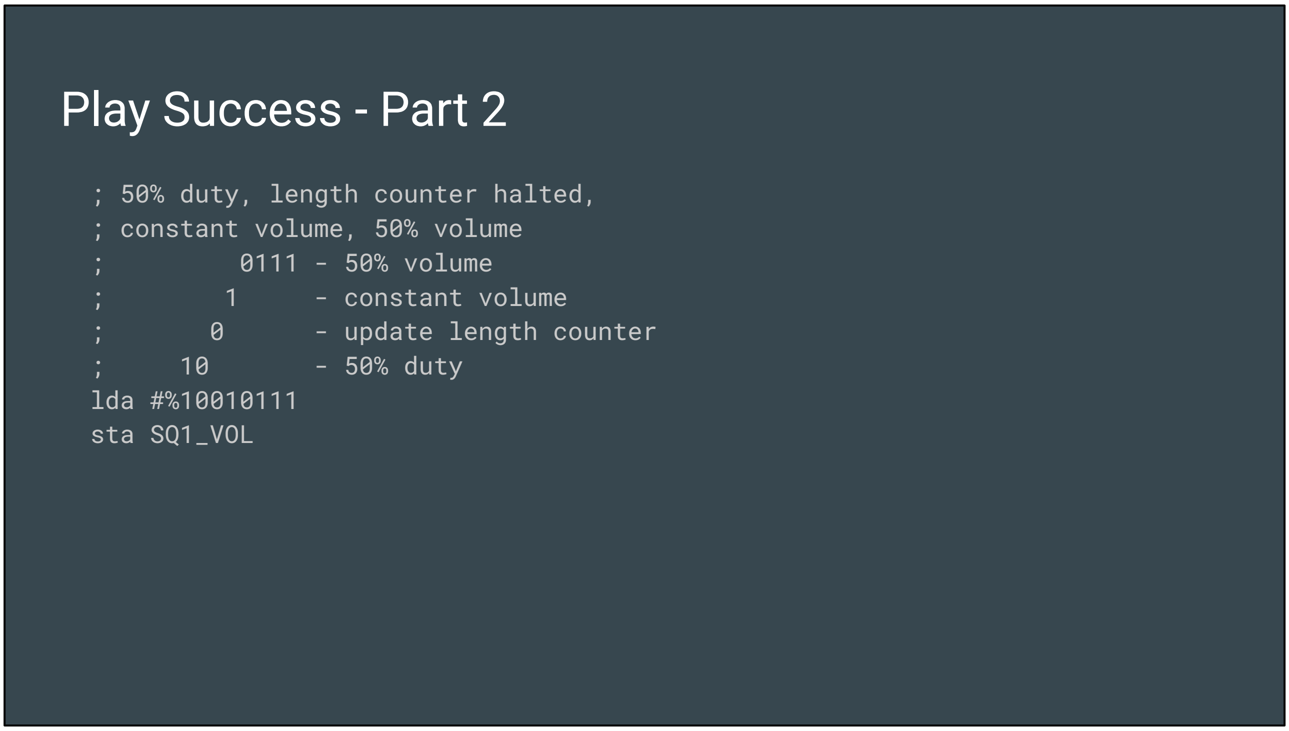
Same check, only in the vertical direction this time. Nothing new!
If all of these checks pass, we unconditionally jump to play_success.



Play Success - Part 1

```
; https://www.nesdev.org/wiki/APU_period_table
; $00e2 == B4
lda #$e2 ; lo byte of $00e2
sta SQ1_LO

; https://www.nesdev.org/wiki/APU_Length_Counter
; $00e2 == B4
; half note @ 75 bpm (NTSC)
;          000 - hi bits of $00e2
;     10110    - half note @ 75 bpm (NTSC)
lda #%10110000
sta SQ1_HI
```

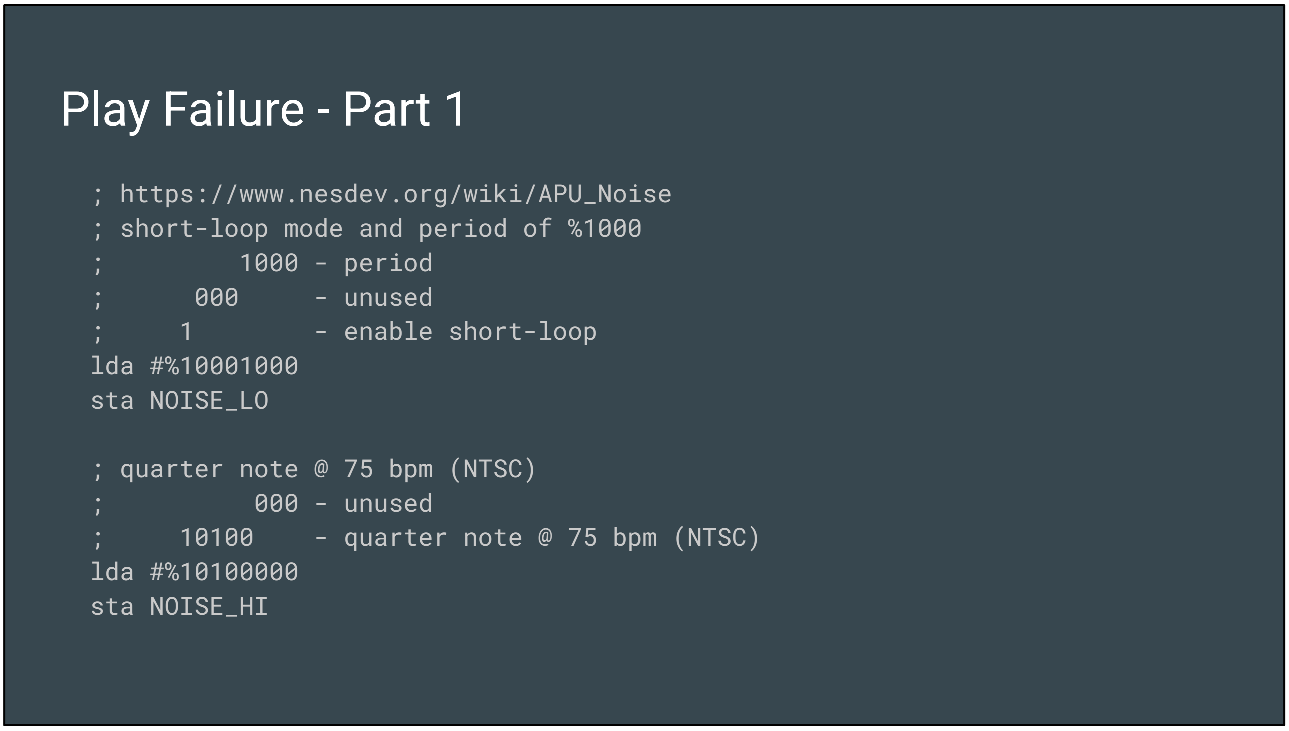
Success is signaled by a beep on the pulse channel.
Using the APU period table and length counter table, we initialize this beep to be a B4 note that lasts two beats at 75 beats per minute.
This will be slightly out of tune and slightly longer in duration on a PAL console, due to frame timing.



Play Success - Part 2

```
; 50% duty, length counter halted,
; constant volume, 50% volume
;         0111 - 50% volume
;        1     - constant volume
;       0      - update length counter
;     10       - 50% duty
lda #%10010111
sta SQ1_VOL
```

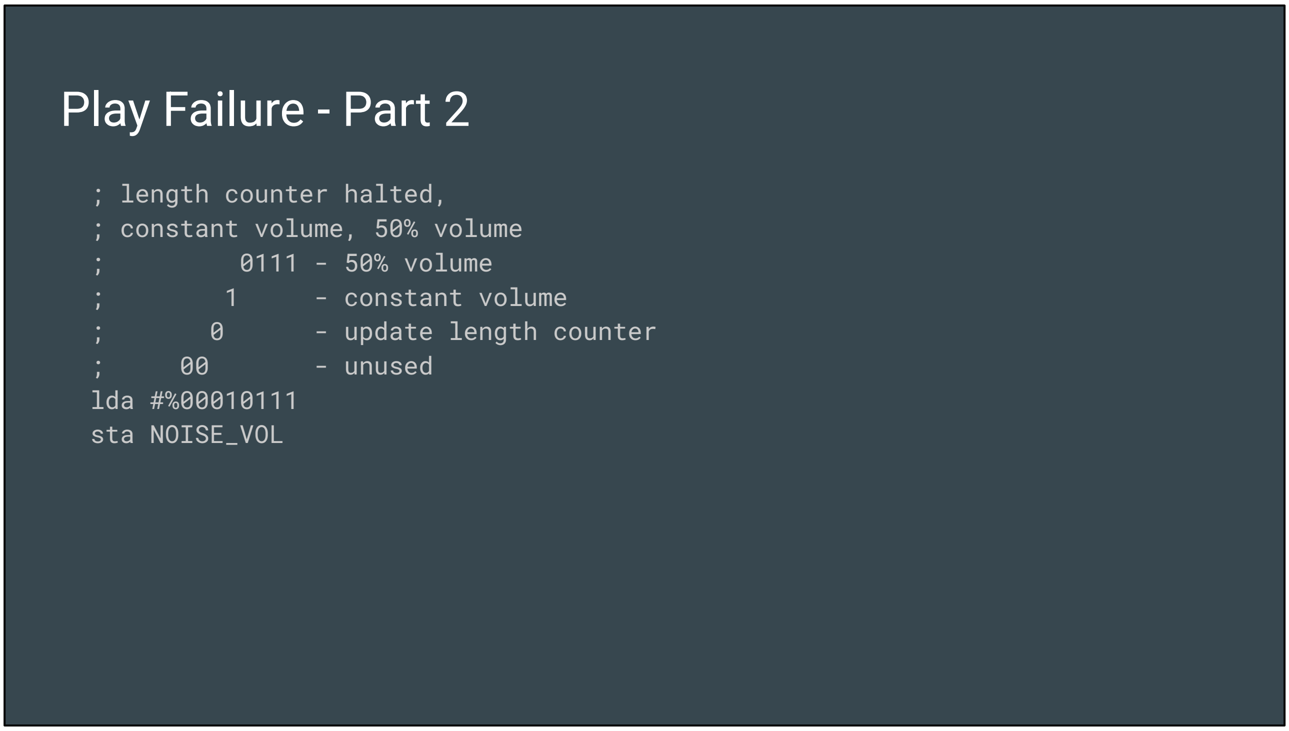
Finally, we set the note’s volume to 50%, specify that it has a constant volume (not using a volume envelope), set the length counter to count down and halt the note after the duration has passed, and set the wave duty to 50%, which affects the shape of the wave and “fullness” of the note.
Unfortunately, sound programming is usually under discussed, and we will be continuing this tradition today.
Fortunately, there is a very good sound library for the NES that we will introduce at the end of the talk.



Play Failure - Part 1

```
; https://www.nesdev.org/wiki/APU_Noise
; short-loop mode and period of %1000
;         1000 - period
;      000     - unused
;     1        - enable short-loop
lda #%10001000
sta NOISE_LO

; quarter note @ 75 bpm (NTSC)
;          000 - unused
;     10100    - quarter note @ 75 bpm (NTSC)
lda #%10100000
sta NOISE_HI
```

Failure is signaled by a buzz on the noise channel.
The noise channel has two looping modes, which changes the resulting sound, along with the period. Here we use the shorter loop mode.
Using the same length counter table, we choose a shorter note duration than the one used for success.



Play Failure - Part 2

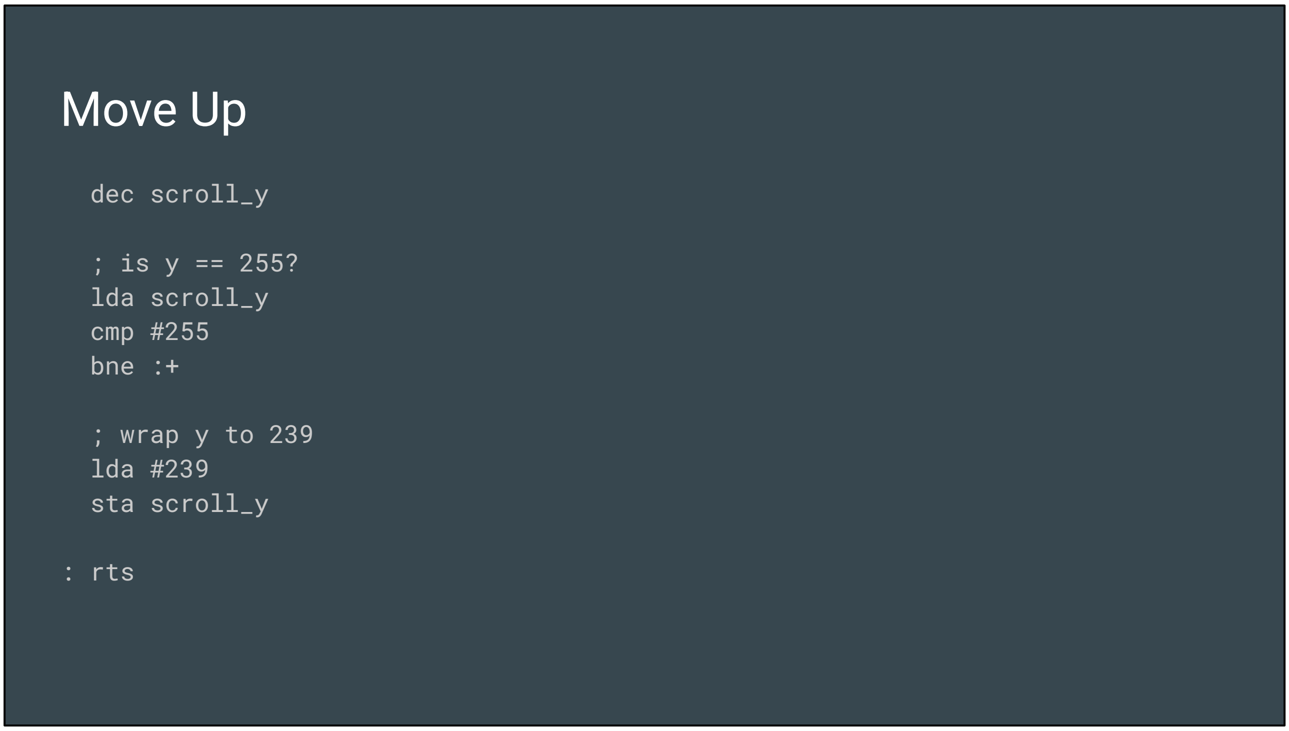
```
; length counter halted,
; constant volume, 50% volume
;         0111 - 50% volume
;        1     - constant volume
;       0      - update length counter
;     00       - unused
lda #%00010111
sta NOISE_VOL
```

These parameters are identical to those used previously, omitting the wave duty, which is only available on the pulse channels.

Move Right

```
  inc scroll_x

  ; is x == 0?
  lda scroll_x
  ; no need to cmp #0, lda sets the zero flag
  bne :+

  ; flip base nametable
  lda ppu_ctrl
  eor #1
  sta ppu_ctrl

: rts
```

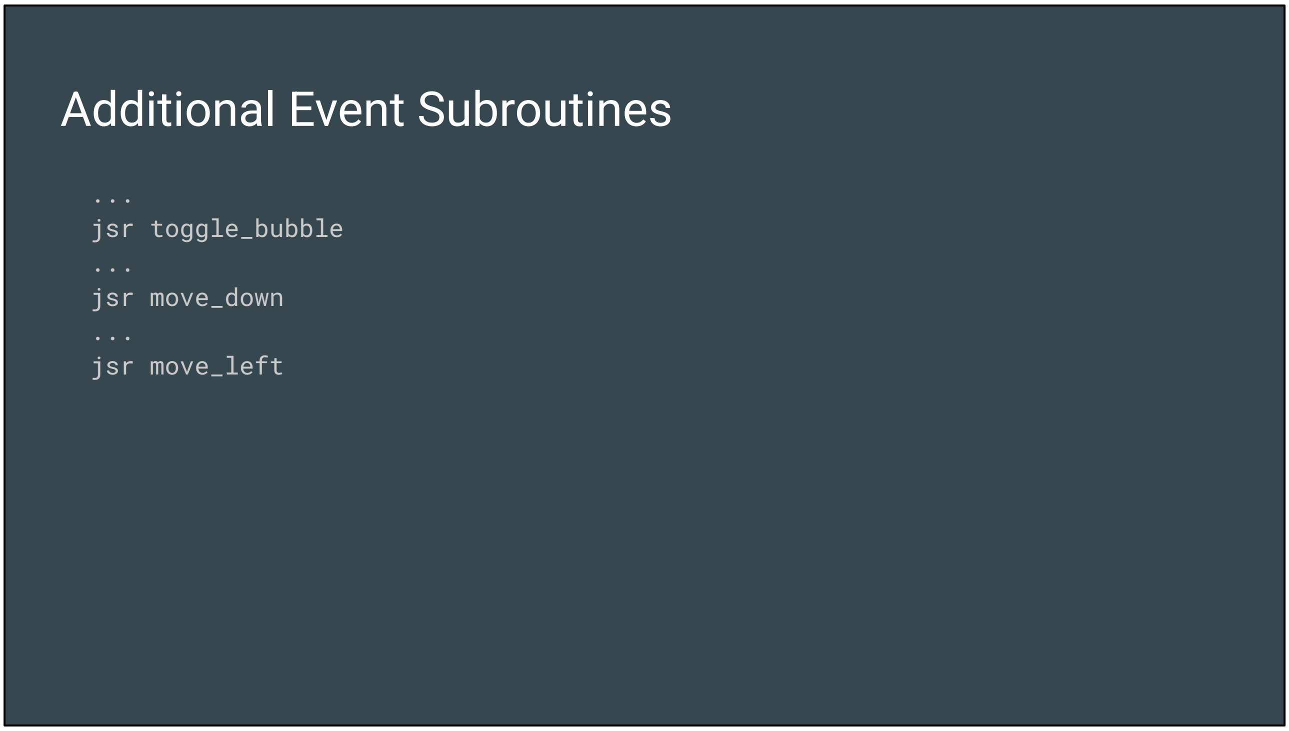
To move to the right, we need to increment the value of scroll_x. If you wanted to move faster or slower, you could adc a different amount here.
We then need to check if we've overflowed back to 0. If we have, we need to progress to the other nametable, since our nametables are arranged horizontally.
Recall that eor can be useful for flipping specific bits, in this case the bit corresponding with the base nametable.



Move Up

```
  dec scroll_y

  ; is y == 255?
  lda scroll_y
  cmp #255
  bne :+

  ; wrap y to 239
  lda #239
  sta scroll_y

: rts
```

Moving up is very similar to move right, just with different variables and values.
We first decrement scroll_y, then check to see if we need to wrap. There is no need to update the base nametable, since we are in vertical mirroring.
Recall that nametables are 256 x 240, so we must wrap back to 239.



Additional Event Subroutines

```
...
jsr toggle_bubble
...
jsr move_down
...
jsr move_left
```

The player has a few other actions that they can take, but the corresponding code should be obvious based on what we’ve already covered.
The thought bubble is hidden by moving it off-screen, just like the unused sprites we looked at during initialization.
These can be explored after the talk, for the sake of time.

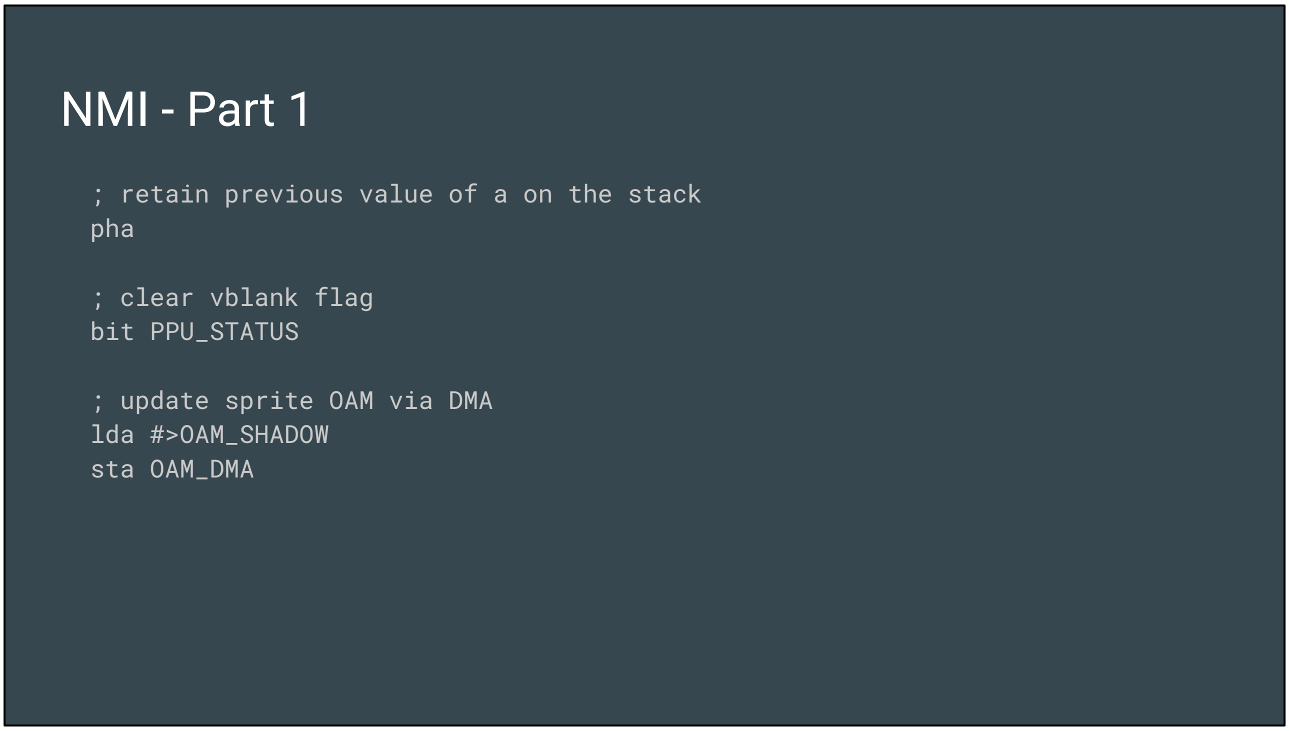
Update Animation

- the most complicated subroutine in the example
- takes advantage of the sprite pattern table layout
- based on a timer updated every frame while the player is moving

- Since there is no “right way” to handle animations like this, and for the sake of time, we won’t be covering this code during the talk.

- Luckily, you already have all the 6502 assembly skills you need to decipher this logic!

We interrupt this talk… for graphics!



NMI - Part 1

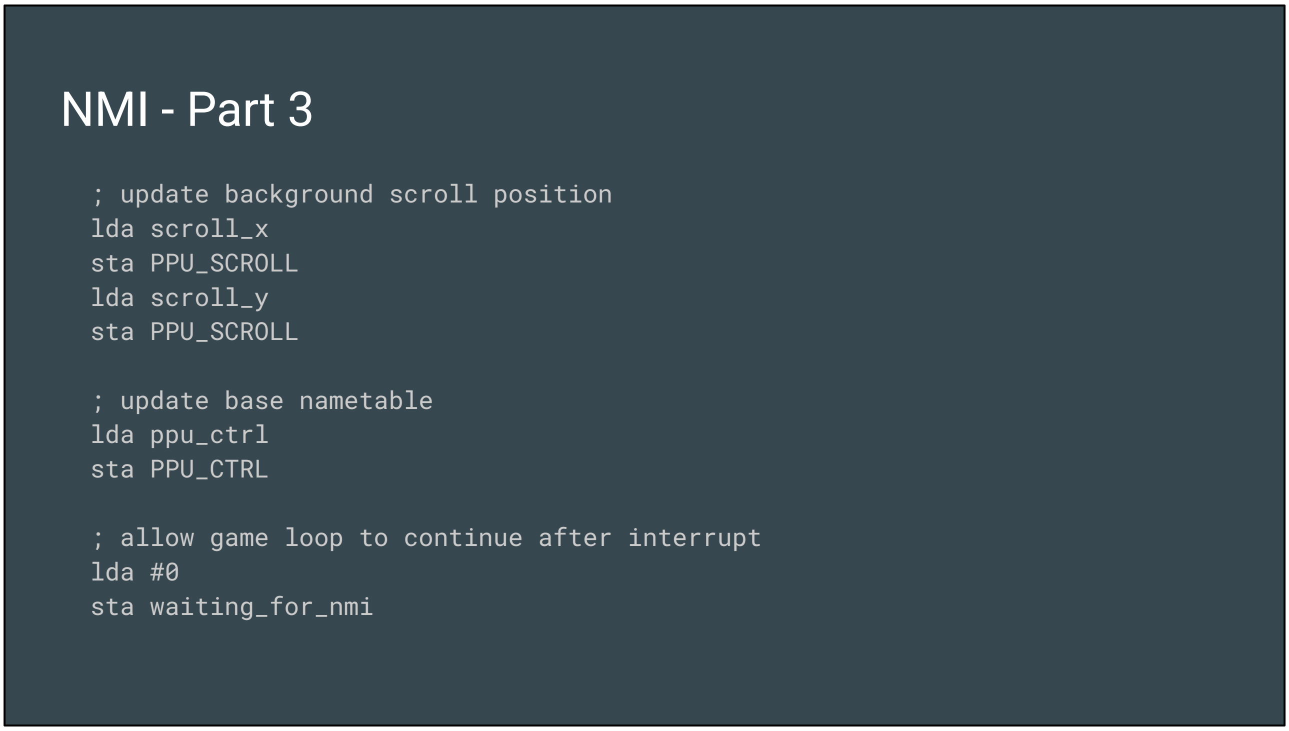
```
; retain previous value of a on the stack
pha

; clear vblank flag
bit PPU_STATUS

; update sprite OAM via DMA
lda #>OAM_SHADOW
sta OAM_DMA
```

Since we don’t know when the interrupt will occur, we must preserve all the registers we will affect during the interrupt.
The processor status flags are automatically preserved for us.
The entire OAM is updated via Direct Memory Access by writing the high-byte of the OAM shadow’s address to OAM_DMA.

NMI - Part 2

```
; show backgrounds and sprites, including leftmost 8 pixels
;            0 - disable grayscale rendering
;           1  - show background in left-most 8 pixels on screen
;          1   - show sprites in left-most 8 pixels on screen
;         1    - draw backgrounds
;        1     - draw sprites
;       0      - disable red emphasis
;      0       - disable green emphasis
;     0        - disable blue emphasis
lda #%00011110
sta PPU_MASK
```

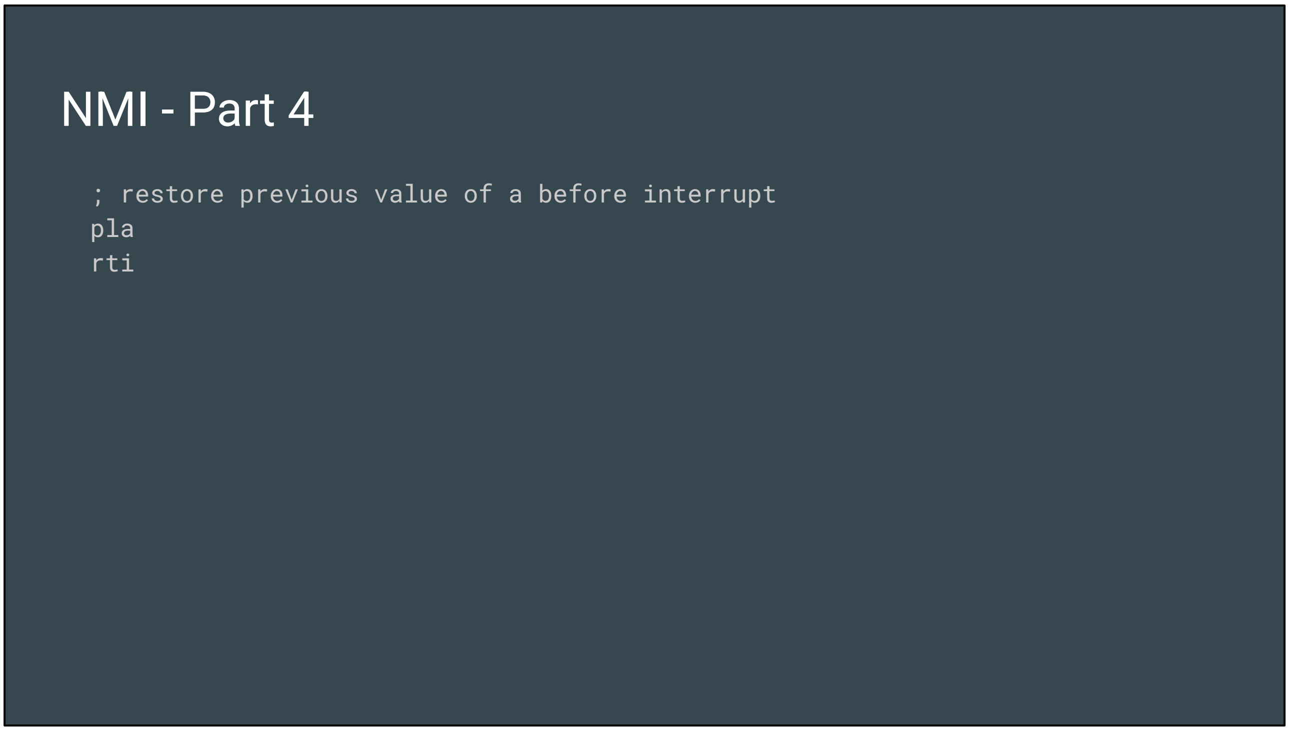
Enabling the backgrounds and sprites during each NMI ensures that no partial frames are drawn during initialization.



NMI - Part 3

```
; update background scroll position
lda scroll_x
sta PPU_SCROLL
lda scroll_y
sta PPU_SCROLL

; update base nametable
lda ppu_ctrl
sta PPU_CTRL

; allow game loop to continue after interrupt
lda #0
sta waiting_for_nmi
```

Scroll position is updated via a pair of writes to PPU_SCROLL, similar to PPU_ADDR.

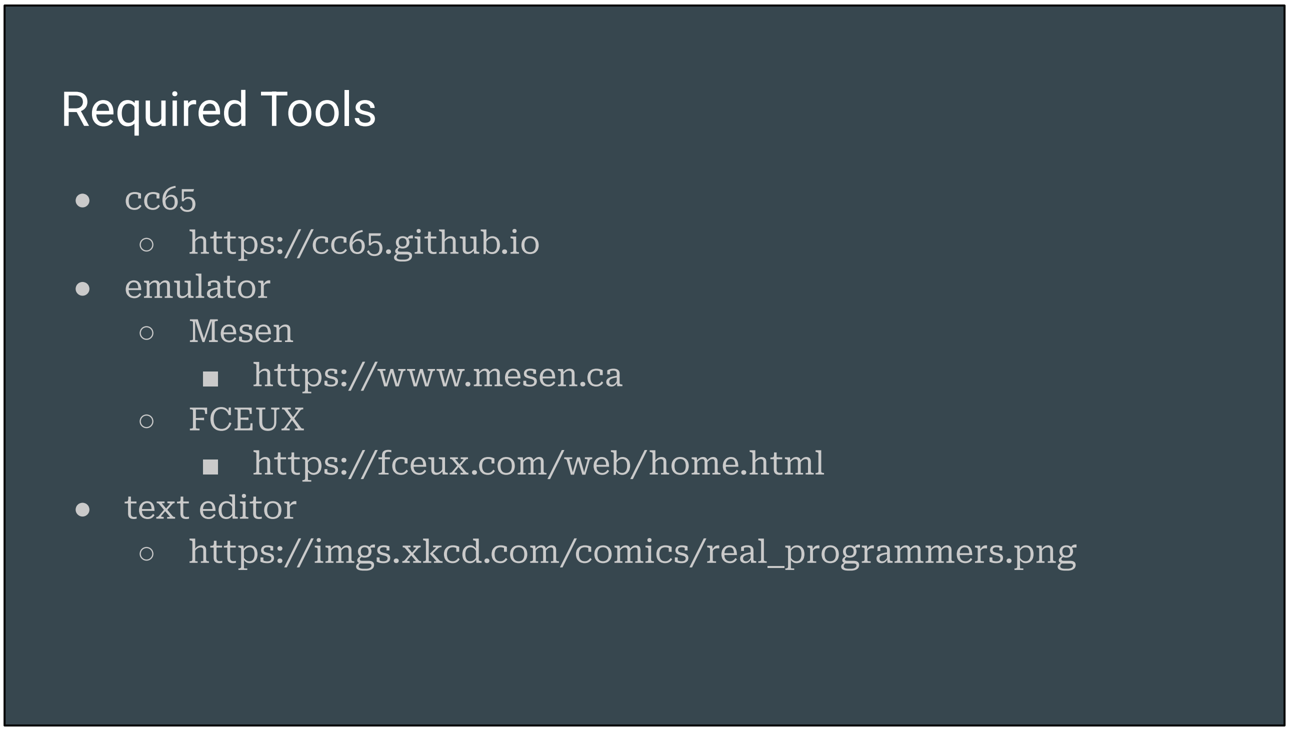


NMI - Part 4

```
; restore previous value of a before interrupt
pla
rti
```

Finally, the contents of a is restored before a Return from Interrupt instruction.
rti is necessary here, so that the processor status flags are restored correctly and code execution returns to the point when the interrupt took place.

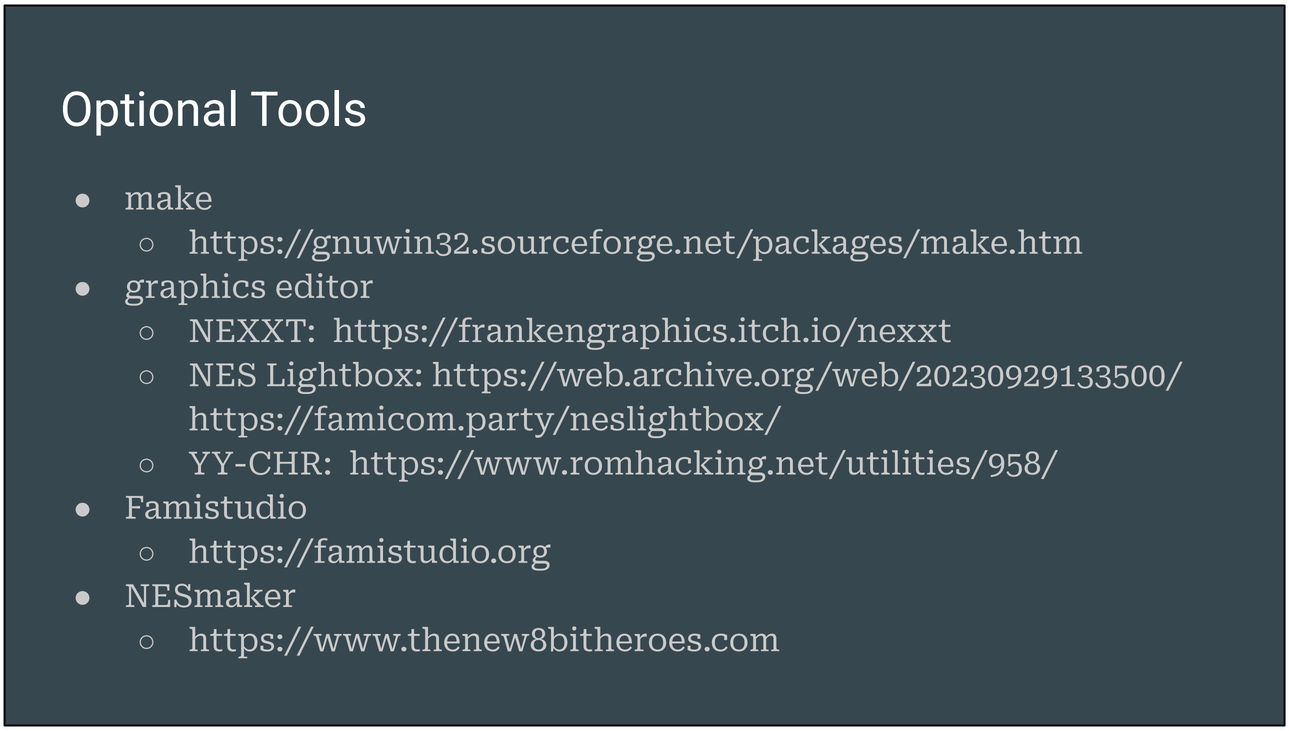
We made it!



Required Tools

- cc65
    - https://cc65.github.io
- emulator
    - Mesen
        - https://www.mesen.ca
    - FCEUX
        - https://fceux.com/web/home.html
- text editor
    - https://imgs.xkcd.com/comics/real_programmers.png

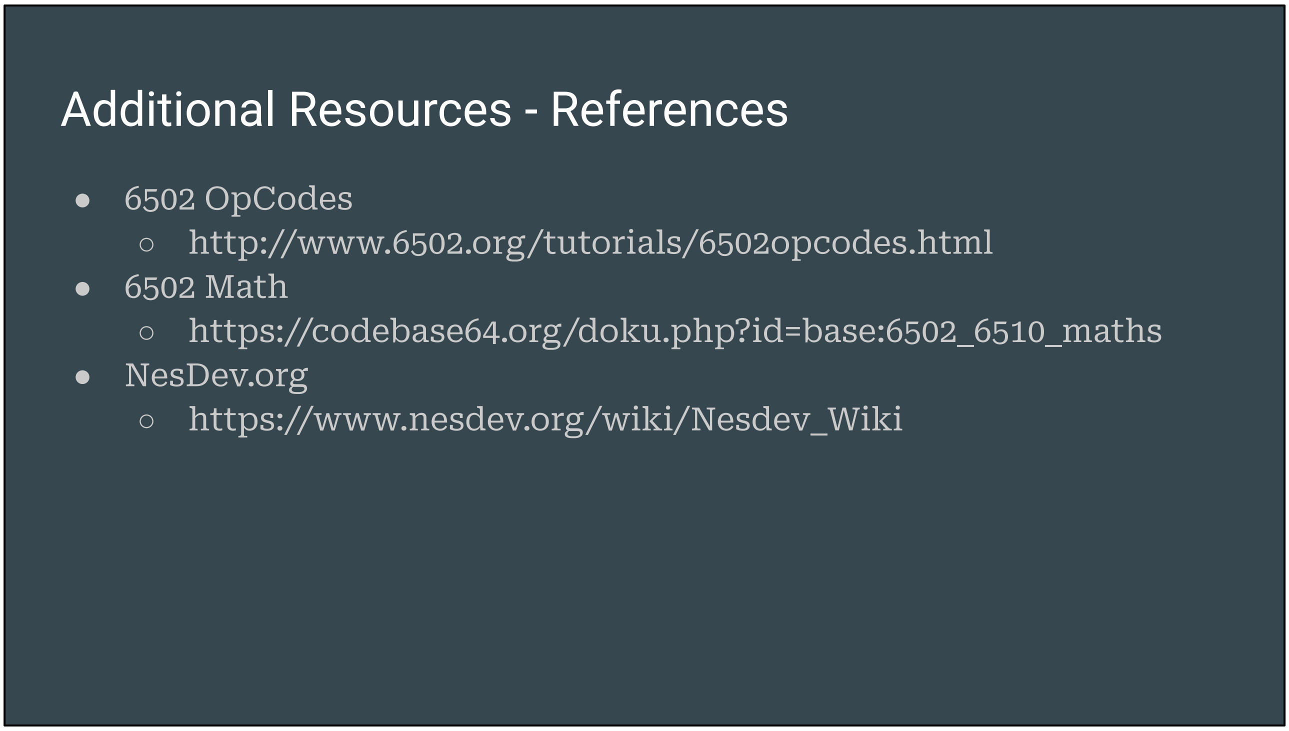
cc65 is a development package for the 6502, containing an assembler, linker, and C compiler.
Mesen and FCEUX both have good tools built in for NES development. The event viewer in Mesen is invaluable!
Of course you'll need your favorite text editor to write your code. If in doubt, please reference this XKCD page.



Optional Tools

- make
  - https://gnuwin32.sourceforge.net/packages/make.htm
- graphics editor
  - NEXXT: https://frankengraphics.itch.io/nexxt
  - NES Lightbox: https://web.archive.org/web/20230929133500/https://famicom.party/neslightbox/
  - YY-CHR: https://www.romhacking.net/utilities/958/
- Famistudio
  - https://famistudio.org
- NESmaker
  - https://www.thenew8bitheroes.com

A Makefile has been included with the example project for simple building. Simpler rebuild scripts are also available, but don't offer incremental building.
All of these graphics editors are compatible with the graphics formats presented in this project. It's also possible to process your own PNGs! Use what works for you.
Famistudio is an absolutely amazing music editor and playback engine for the NES. It has unmatched functionality and tutorials to get you up to speed.
This talk has been focused on 6502 assembly, but if you're not ready to take that plunge, NESmaker is a great alternative.



Additional Resources - References

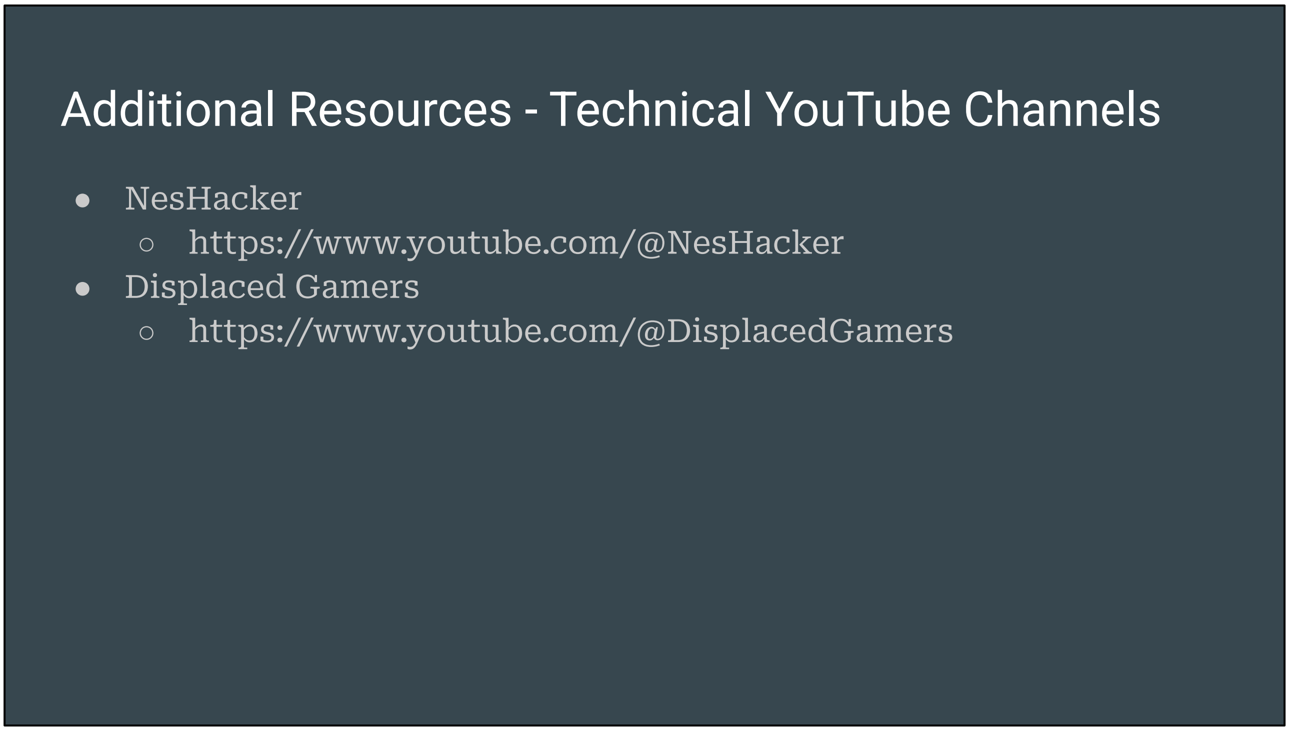
- 6502 OpCodes
  - http://www.6502.org/tutorials/6502opcodes.html
- 6502 Math
  - https://codebase64.org/doku.php?id=base:6502_6510_maths
- NesDev.org
  - https://www.nesdev.org/wiki/Nesdev_Wiki

This has been a whirlwind tour of an introduction to NES programming, but thankfully, there are so many online resources available.
Whether you are looking for a deeper explanation of something, or just want to hear someone else's take, these should have you covered.
This list is by no means exhaustive and is always open to suggestions!

Additional Resources - Online Courses

- Pikuma
  - https://pikuma.com/courses/nes-game-programming-tutorial
- Nerdy Nights
  - https://nerdy-nights.nes.science

Pikuma is a set of premium online courses taught by Gustavo Pezzi, a computer science lecturer from London.
Nerdy Nights is a series of online tutorials written by Brian Parker.



Additional Resources - Technical YouTube Channels

- NesHacker
  - https://www.youtube.com/@NesHacker
- Displaced Gamers
  - https://www.youtube.com/@DisplacedGamers

Ryan at NesHacker does a great job introducing concepts vital to NES, and lately Game Boy, programming.
Displaced Gamers routinely takes thorough deep-dives through the code of retail NES games.

Questions, Suggestions, and/or Gratuitous Praise?

- Come by the indie developer booths, between the content creators and console freeplay area, to check out everything Sleepy Bits is doing on the NES, including an exclusive preview of our next game, and let's chat about any questions you might have!

- @chimbraca and @SleepyBitsNES on Twitter
- @grendell and @SleepyBitsGames on Bluesky
- @SleepyBitsGames on YouTube for game teasers and trailers

- grayson@sleepybitsgames.com

Thank you!!

Most importantly, remember to have fun!
Thanks so much for attending this talk today, and have an awesome weekend.
Thank you!