

Building Your First Emulator

Understanding How Emulators Work
by Building a NES from Scratch



Table of Contents

[Introduction](#)

[Who this book is for](#)

[What you'll achieve](#)

[What this book is NOT](#)

[Why Crystal \(and not C, and almost Ruby\).](#)

[How the book is organized](#)

[Feedback](#)

[Dedication](#)

[Chapter 1: Anatomy of a NES](#)

[What emulation means](#)

[A little historical context](#)

[What's under the hood](#)

[The emulator loop](#)

[Where we start](#)

[Chapter 2: Setup](#)

[Editor](#)

[Installing Crystal](#)

[Compile and run](#)

[Chapter 3: CPU](#)

[The 6502](#)

[Instruction families](#)

[The registers](#)

[How a CPU works \(short version\).](#)

[High-level project structure](#)

[Chapter 3.1: Our first two instructions](#)

[A bit of hexadecimal](#)

[Types in Crystal](#)

[The opcodes](#)

[The CPU](#)

[The instructions](#)

[The flags](#)
[The Bus](#)
[The Emulator](#)
[Debugging](#)
[Tests](#)
[Wrap up](#)

[Chapter 3.2: Store Instructions and Addressing Modes](#)

[Addressing Modes](#)
[Zero Page](#)
[Absolute](#)
[Implementation](#)
[Tests](#)
[Updated structure](#)
[Wrap up](#)

[Chapter 3.3: All Addressing Modes](#)

[The ones we already know](#)
[Zero Page with index](#)
[Absolute with index](#)
[Relative](#)
[Indirect](#)
[Indexed Indirect: \(\\$nn,X\)](#)
[Indirect Indexed: \(\\$nn\),Y](#)
[\(\\$nn,X\) vs \(\\$nn\),Y: which is which?](#)
[The complete module](#)
[Wrap up](#)

[Chapter 3.4: Completing Load and Store](#)

[Completing LDA](#)
[LDX and LDY](#)
[The Store instructions](#)
[STX and STY](#)
[Opcode table](#)
[Wiring it all up](#)
[Tests](#)
[Updated structure](#)

Wrap up

Chapter 3.5: Complex Instructions

New flag helpers

Branches, the 6502's if statements

Jumps, the 6502's goto

Comparisons, the invisible subtraction

Shifts: multiply by two

Arithmetic: ADC

Cleaning up the initialize

Wiring it all up

Updated structure

Wrap up

Chapter 3.6: Interrupts

Interrupt vectors

The Interrupts module

BRK, Break

RTI, Return from Interrupt

Wiring it up

Wrap up

Chapter 3.7: Completing the CPU, all remaining instructions

Complete table

Register increments and decrements

Register transfers

Flag operations

Stack operations

Logical: AND, ORA, EOR

Comparisons, remaining modes

Arithmetic: remaining ADC + SBC

Full shifts: ASL, LSR, ROL, ROR

Misc: INC, DEC, BIT, NOP

Wiring everything up

Wrap up

Chapter 4: The Cartridge

[What's inside a cartridge?](#)

[PRG-ROM: the code](#)

[CHR-ROM: the graphics](#)

[The iNES format](#)

[Mappers: why they exist and what they do](#)

[Mirroring: the story of the mirrored screens](#)

[What do we need to build?](#)

[Chapter 4.1: Reading a ROM](#)

[The Cartridge class](#)

[The format constants](#)

[The Mirroring enum](#)

[The parsing](#)

[Wiring it up](#)

[Test](#)

[Updated structure](#)

[Wrap up](#)

[Chapter 4.2: Mapper 0 and the First Boot](#)

[The translation problem](#)

[The Mapper class](#)

[Mapper 0: NROM](#)

[Connecting the mapper to the Cartridge](#)

[The Bus learns to read from the cartridge](#)

[The boot: from file to Program Counter](#)

[UnknownOpcode](#)

[nestest: trial by fire](#)

[Updated file structure](#)

[Wrap up](#)

[Chapter 5: The PPU](#)

[The NES screen](#)

[Tiles: the building blocks of everything](#)

[Pattern tables: the tile catalog](#)

[Nametables: the background map](#)

[Attribute tables: palettes by block](#)

[Palettes: the final colors](#)

[Sprites and OAM: the characters](#)
[PPU registers \(\\$2000-\\$2007\)](#)
[Timing: scanlines and cycles](#)
[NMI: the VBlank alarm](#)
[What we're going to build](#)

[Chapter 5.1: Building the PPU](#)

[The master palette](#)
[The PPU skeleton](#)
[Constants: addresses and display](#)
[The loopy registers](#)
[The PPU registers](#)
[VRAM: the memory router](#)
[Timing and VBlank](#)
[NMI in the CPU](#)
[Integration into Bus and Emulator](#)
[Tests](#)
[Wrap up](#)

[Quick Break: A Brief Reflection on the PPU](#)

[Modern GPUs: general-purpose processors](#)
[The NES: every transistor counts](#)
[The escalation: from the NES to the SNES](#)
[The 3D transition: PlayStation and beyond](#)
[Back to the NES](#)

[Chapter 5.2: Opening a Window](#)

[SDL2](#)
[Bindings: Crystal Talking to C](#)
[The GUI Script](#)
[Verifying It Works](#)
[Wrap up](#)

[Chapter 5.3: Rendering the Background](#)

[What Mario Already Did](#)
[Register Helpers](#)
[Reading the Scroll Position](#)

[The Rendering: From Tile to Pixel](#)

[Hooking Into step](#)

[The Scroll Increments](#)

[The Complete step](#)

[The Moment of Truth](#)

[Wrap up](#)

[Chapter 5.4: Controls](#)

[The NES controller](#)

[Connecting to the Bus](#)

[New SDL2 bindings](#)

[The keyboard](#)

[Testing the controller](#)

[The Moment of Truth](#)

[Investigating the problem](#)

[OAM DMA](#)

[Sprite 0 hit](#)

[The Second Attempt](#)

[Why did it work?](#)

[Challenge: Playing Blind](#)

[Wrap up](#)

[Chapter 5.5: Sprites](#)

[Anatomy of a sprite](#)

[The constants](#)

[Finding the winning sprite](#)

[The sprite palette](#)

[The priority logic](#)

[Compiling with —release](#)

[The moment of truth](#)

[Wrap up](#)

[Chapter 6: Let's Play](#)

[Getting ROMs](#)

[Mapper 0 Games](#)

[Mapper 1 Games](#)

[Try a Mapper 1 Game](#)

[Mapper 1 and Beyond](#)
[Wrap up](#)

[Chapter 7: The APU](#)

[What Is Sound](#)
[Digital Waves](#)
[Waveforms](#)
[Audio Synthesis](#)
[The NES Channels](#)
[Anatomy of a Channel](#)
[The Frame Counter](#)
[APU Registers](#)
[The Mixer](#)
[From the NES to the Speaker](#)
[What We Don't Implement](#)
[What We're Going to Build](#)

[Chapter 7.1: Implementing the APU](#)

[The Constants](#)
[The Pulse Channel](#)
[The Triangle Channel](#)
[The Noise Channel](#)
[The APU Class](#)
[Connecting to the Bus](#)
[The Emulator](#)
[Audio with SDL2](#)
[The Moment of Truth](#)
[Wrap up](#)

[Chapter 8: Closing](#)

[What we actually learned](#)
[The third component everyone forgets](#)
[So what do I emulate next?](#)
[Improvements for this emulator](#)
[Final words](#)

[Appendix: Mappers](#)

Mapper 1 (MMC1).

OceanofPDF.com

OceanofPDF.com

Introduction

One night I was searching “Mario Bros NES online” at 2 AM. Pure nostalgia. I found an emulator, played for 15 minutes, died in world 2-3 as always, and closed the tab. But something stuck with me: how does that actually work?

I’d always been fascinated by the idea of virtual machines. How software can simulate hardware. How you can have infinite computers inside your computer, configuring their specs with a couple of clicks. But I’d never crossed from “**that’s interesting**” to “**I’m going to understand how it’s done**”.

That night I crossed. Three weeks later I had a working emulator. And here we are.

OceanofPDF.com

Who this book is for

People who know how to program (intermediate level at least) and like learning by doing. You don't need to know anything about emulation, retro hardware, or assembly. Just the itch to build something.

If you've ever thought "I'd love to take on a big project but I don't know where to start", this is a great candidate. It's bounded (we're not emulating a PS2), it's tangible (you end up with something that actually works and you can use), and there's that unique satisfaction of seeing your own code running a game from your childhood.

OceanofPDF.com

What you'll achieve

When you're done, you'll have a working NES emulator. One that you understand end to end: what each module does, why it's there, and how it fits together. It'll run games with mappers 0 and 1 (you'll learn what mappers are soon enough), which covers most of the classics: Super Mario Bros, Zelda, Metroid, Contra, Mega Man 2.

My quality test throughout the whole development was simple: if it runs Mario, we're good.

OceanofPDF.com

What this book is NOT

Let's be clear from the start:

- **It's not a perfect emulator:** We don't emulate undocumented CPU instructions, and we don't replicate every bug from the original hardware. There are projects that do that with surgical precision. This one prioritizes helping you understand what's happening.
- **It's not an exhaustive technical reference:** If you want the complete NES documentation, there are entire wikis for that. Here you'll find what you need to build something that works, explained in a way that won't put you to sleep.
- **It's not a book for absolute beginners:** I assume you know what a function is, what a loop is, and that you won't panic if you see code.

What it is: a practical guide to building your first emulator, written by someone who knew nothing about this stuff when they started.

OceanofPDF.com

Why Crystal (and not C, and almost Ruby)

Very few people write C or C++ on a daily basis. And the goal of this book is for you to understand how an emulator works, not to spend 15 minutes deciphering pointers.

That's why I started writing the emulator in Ruby. Readable, expressive, a joy to work with. Spoiler: also incredibly slow.

When I had a version that could minimally run Mario, I was getting 0.5 FPS. Half a frame per second. With optimizations I got to 2 FPS. Loading level 1-1 took about 4 minutes.

It was an interesting lesson: I'd always heard "Ruby is slow" as an abstract truism, something that didn't matter for most applications. And that's true, for most things it doesn't matter. But emulating hardware in real time isn't most things. It was the first time I hit a real limitation, not a theoretical one, where a language simply can't do the job.

So I rewrote everything in Crystal: a language created in Argentina that has the same syntax as Ruby but is statically typed and compiled. I know static typing can be a touchy subject if you're coming from Ruby, but the results speak for themselves: without any optimization, the emulator jumped to 120 FPS. The same code, 240 times faster.

All the code in this book is in Crystal. If you're coming from Ruby, you'll read it without any issues. If you've never seen Crystal before, you'll pick up a new language along the way.

OceanofPDF.com

How the book is organized

We start with a general overview of the NES and what it means to emulate a console. Then we dive straight into the CPU, followed by the cartridge reader, the PPU (the graphics unit), the GUI, input, some polish, and finally sound.

Each section has code. There's a repository with commits per chapter so you can follow along, but I recommend writing at least half of each chapter yourself, without copy-pasting. That's the difference between understanding how it works and just having something that works. For the repetitive parts that don't add to your understanding, go ahead and ctrl+c ctrl+v guilt-free.

All code is in English. It's an industry convention and it also makes it easier to compare the code against the [NES Wiki](#), which is where all the documentation about how the NES works lives.

And if by the end you want to play a game that needs a mapper we didn't cover, I'll leave that as homework. Feel free to submit it to the repo as a contribution if you want to share it with others.

OceanofPDF.com

Feedback

If you find bugs, have suggestions, or want to share your progress, you can reach me by email or Twitter. The details are at the end of the book.

OceanofPDF.com

Dedication

To the Japanese developers who programmed Super Mario Bros in 6502 assembly, fitting an entire game into 32KB of memory. Meanwhile, we complain when a single npm dependency is 50MB.

Let's begin.

[OceanofPDF.com](https://oceanofpdf.com)

Chapter 1: Anatomy of a NES

In this chapter we're going to look at what's inside a NES, what each part does, and how that translates into code. By the end you'll have a solid mental map of the whole system before we dive into the details.

OceanofPDF.com

What emulation means

Emulating is lying. It's convincing a program that it's running on hardware that doesn't exist.

A NES game is a sequence of instructions written for a specific processor (the 6502). When Nintendo manufactured cartridges in the 80s, those instructions ran on physical chips. We don't have those chips, but we can write software that behaves the same way.

The game says "add these two numbers and store the result here". Our emulator reads that instruction, does the math, and stores the result in the right place in simulated memory. The game never knows it's not running on a real NES.

Think of it like being a simultaneous interpreter, except instead of translating languages, you're translating hardware instructions into software operations. In real time. 60 times per second.

OceanofPDF.com

A little historical context

The NES (Nintendo Entertainment System) launched in Japan in 1983 as the “Famicom”. It hit the US in 1985, disguised as an “entertainment system” because the video game industry was so dead after the crash of ’83 that nobody wanted to touch another “video game console”.

Some numbers for perspective:

- **CPU:** 1.79 MHz (your phone runs at ~3000 MHz)
- **RAM:** 2 KB (this paragraph weighs more than that)
- **Resolution:** 256x240 pixels, 54 possible colors on screen

With those constraints, developers made Super Mario Bros, Zelda, Metroid, and Mega Man. The entire Super Mario Bros cartridge is 32 KB. A photo of your lunch is 3 MB.

Next time you complain about JavaScript being slow, remember that people pulled off parallax scrolling in 2 KB of RAM writing in assembly.

OceanofPDF.com

What's under the hood

The NES has four main components that we care about:

CPU (Central Processing Unit)

The brain. A Ricoh 2A03, which is basically a MOS 6502 with some modifications. It executes the game's instructions: the logic, the physics, what happens when you stomp a Goomba.

It has 2 KB of RAM to work with. Sounds tiny (and it is), but the cartridges brought their own ROM with the game code, so those 2 KB were just for temporary variables, enemy positions, player state, etc.

PPU (Picture Processing Unit)

The eyes. It handles drawing everything you see on screen: the backgrounds, the sprites, Mario, the clouds, the blocks.

It has its own memory: 2 KB of VRAM (Video RAM) for backgrounds, plus additional memory on the cartridge for the graphics (the "tiles" that make up the images). The PPU is its own beast. It doesn't take instructions like "draw Mario at position X,Y". It's more primitive than that: it takes 8x8 pixel patterns and arranges them on screen according to tables that the CPU configures. It's elegant and horrible at the same time.

APU (Audio Processing Unit)

The ears. It generates sound: the bleeps, the bloops, the level music. We'll save this for the end of the book because it's independent of everything else, and the emulator works perfectly without audio (in silence, the way we code at 3 AM).

Cartridge

The cartridge isn't just storage. It contains:

- **PRG-ROM:** The game code (the instructions the CPU executes)
- **CHR-ROM:** The graphics (the tiles the PPU uses)
- **Mapper:** Additional circuitry for larger games

The earliest games fit in the NES's memory space just fine. But as games got more ambitious, they needed more space than the NES could address. Mappers solve this with “bank switching”: swapping sections of cartridge memory in and out as the game needs them. Don't worry about mappers for now. We'll start with simple games (mapper 0) and add complexity later.

Here's how it all connects:

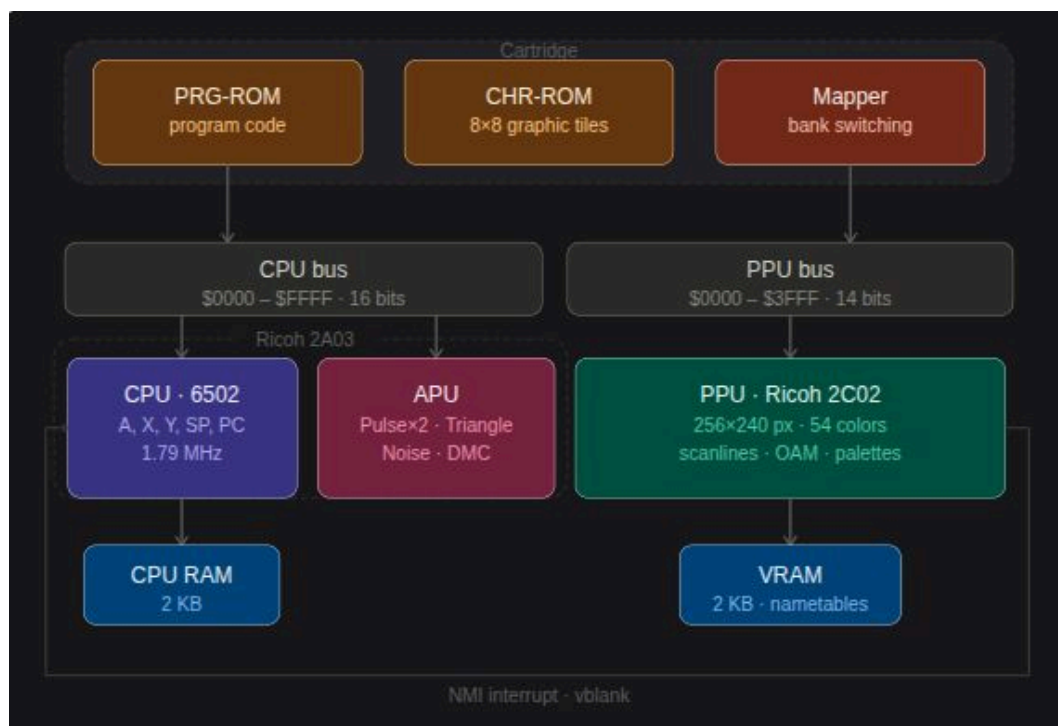


Figure 1. NES Diagram

The CPU runs the game logic and tells the PPU what to display. The PPU draws frames. Both run in parallel, coordinated by a master clock.

The emulator loop

Here's the heart of what we're going to build:

```
1  loop do
2    # 1. CPU executes one instruction
3    cpu_cycles = cpu.step
4
5    # 2. PPU runs 3 cycles for each CPU cycle
6    (cpu_cycles * 3).times do
7      ppu.step
8    end
9
10   # 3. Repeat forever (or until you close the emulator)
11 end
```

Let's break it down.

Why is `cpu_cycles` a variable?

Not all CPU instructions take the same amount of time. Some are fast (2 cycles), others slower (up to 7 cycles). `LDA #$00` (load a value into a register) takes 2 cycles. `JSR $8000` (jump to a subroutine) takes 6.

When we call `cpu.step`, the CPU executes one complete instruction and returns how many cycles it took. We need that number to know how many PPU cycles to run.

Why 3 PPU cycles for each CPU cycle?

The PPU runs at 5.37 MHz and the CPU at 1.79 MHz. Divide them and you get ~3. This means that for every CPU cycle, the PPU advances three cycles.

If the CPU executes a 4-cycle instruction, the PPU has to advance 12 cycles (4 x 3) to stay synchronized.

The simplification we're making

There are two ways to emulate this:

- **Tick-perfect (the hard way):** You emulate each individual cycle. CPU does one tick, PPU does three ticks. CPU does another tick, PPU does

three more. And so on, for every cycle of every instruction. This means you have to track exactly where the CPU is mid-instruction: “I already read the opcode, now I’m reading the operand, now I’m computing the address...”. An instruction can be half-executed between one tick and the next. This is what emulators aiming for absolute accuracy do, and it’s considerably more complex to implement.

- **Instruction by instruction (ours):** The CPU executes the full instruction in one go, and then the PPU “catches up” by running all its cycles together. It’s less accurate but much simpler to implement and understand. For 99% of games, our approach works perfectly. The cases where it fails are games that pull very specific tricks that depend on exact cycle-by-cycle timing between CPU and PPU. Those games exist (Battletoads is famous for crashing emulators on level 2 if the timing isn’t exact), but they’re the exception. Mario, Zelda, Mega Man, Contra - they all run perfectly with our approach.

We’re choosing simplicity over perfection here. And it’s not just a pedagogical decision: many popular emulators (including several you can find online) use exactly this strategy. It works well for the vast majority of games.

OceanofPDF.com

Where we start

We're going to build the CPU first.

Why? Because it's the most self-contained component. We can implement the entire CPU and test it without needing anything else. There are test ROMs that verify your CPU works correctly, instruction by instruction.

Once the CPU works, we add the cartridge so we can load games. Then the PPU so we can see something on screen. Then input so we can actually play. And finally sound, because the bleeps can wait.

Each piece we add makes the emulator a little more real. It's like building a digital Frankenstein: first the brain, then the eyes, then the hands.

OceanofPDF.com

Chapter 2: Setup

This is the most boring chapter in the book, but it'll be short, and we need to get it out of the way before jumping into the CPU. We're going to set up the development environment for coding in Crystal.

OceanofPDF.com

Editor

My recommendation is VSCode with the Crystal Language extension. You can find it by searching “Crystal Language” in the extensions marketplace.

Why VSCode? Because it’s one of the few editors with decent Crystal support. The language is niche and doesn’t have the community size of Python or JavaScript, so options are limited. The extension gives you syntax highlighting, some autocompletion, and formatting. It’s not a full IDE, but it gets the job done.

If you prefer Vim, Emacs, or another editor, there are plugins available. Search “crystal” in your editor’s package manager and you’ll probably find something.

OceanofPDF.com

Installing Crystal

There are several ways to install Crystal. You can find the full list on the official [Crystal](#) page. I'd recommend using a package/version manager like asdf, homebrew, or scoop (if you're on Windows), but feel free to install it however works best for you.

[OceanofPDF.com](https://oceanofpdf.com)

Compile and run

To make sure everything works, create a `main.cr` file like this:

```
1 # main.cr
2
3 puts "Hello there" # Yes, it's the same as Ruby.
```

Now we have two ways to run it. Make sure you try both before moving on to the next chapter.

Quick option (interpret)

```
1 crystal run main.cr
```

This compiles and runs in a single step. It's slower because it compiles every time, but it's handy for development.

Release option (compile)

```
1 crystal build main.cr -o hello --release
```

This generates an executable called `hello` (or `hello.exe` on Windows). The `--release` flag enables optimizations, so the binary is faster but takes longer to compile.

Finally, to run it:

```
1 # Linux or Mac
2 ./hello
3
4 # Windows
5 hello.exe
```

If everything went well, you should see “Hello there” in your terminal.

And that's it! You've got everything you need to write Crystal. Let's get into the CPU already.

OceanofPDF.com

Chapter 3: CPU

The CPU is the brain of the NES. It reads instructions, executes them, and tells the rest of the system what to do. If the emulator were a kitchen, the CPU would be the chef: it reads the recipe (the program), uses whatever ingredients are on hand (the registers), and goes step by step until the dish is done. If it skips a step or misreads an instruction, everything falls apart.

In this chapter we're going to implement the full CPU. By the time we're done, you'll be able to run 6502 assembly programs and watch your emulator execute them instruction by instruction. Later, once we have the cartridge reader, we'll run a test ROM to make sure everything works.

OceanofPDF.com

The 6502

The NES uses a variant of the MOS 6502, the same processor found in the Commodore 64, the Apple II, and the Atari 2600. It was cheap, simple, and for its time, pretty capable.

The 6502 has 56 official instructions. I say “official” because there’s a bunch of undocumented instructions that exist as side effects of how the chip was designed. Some games use them, most don’t. We’re going to implement the 56 official ones. If you want to run some obscure game that uses the illegal ones later, that’s on you.

Now, 56 sounds manageable until you find out that those 56 instructions expand into **151 different opcodes**. That’s because the same instruction can work in different ways: sometimes it reads from memory, sometimes from a register, sometimes from an address you calculate by adding things together. Each combination gets its own code. Don’t worry, we’ll take it one step at a time.

OceanofPDF.com

Instruction families

Instructions are grouped into families based on what they do:

Family	What it does	Examples
Load/Store	Move data between registers and memory	LDA, LDX, LDY, STA, STX, STY
Transfer	Move data between registers	TAX, TAY, TXA, TYA, TSX, TXS
Arithmetic	Addition, subtraction	ADC, SBC
Logical	AND, OR, XOR, shifts	AND, ORA, EOR, ASL, LSR, ROL, ROR
Comparison	Compare values and set flags	CMP, CPX, CPY
Branches	Conditional jumps	BEQ, BNE, BCS, BCC, BMI, BPL, BVS, BVC
Jumps	Unconditional jumps	JMP, JSR, RTS, RTI
Stack	Push and pop to the stack	PHA, PHP, PLA, PLP
Flags	Modify flags directly	CLC, SEC, CLI, SEI, CLV, CLD, SED
Other	Miscellaneous	INC, DEC, INX, INY, DEX, DEY, BIT, NOP, BRK

You don't need to memorize any of this right now. The table is here so you have the big picture and can come back to it when we're implementing.

The registers

If you've never worked with computer architecture, think of registers as the CPU's built-in scratchpad. RAM is the backpack: lots of room, but you have to open it, dig around, pull stuff out. Registers are your pockets: few and small, but access is instant. When you need to do an operation, you pull data from RAM into the registers, do your thing, and store the result back.

The 6502 has very few registers. This was normal at the time: memory was expensive, transistors were expensive, everything was expensive. You got the bare minimum and made it work.

- **Accumulator (A)** - 8 bits The main register. Almost all arithmetic and logical operations go through here. If you want to add two numbers, one of them has to be in A.
- **Index X and Index Y** - 8 bits each Used for indexing memory. For example, "load the value at address $0x200 + X$ ". They also work as loop counters.
- **Program Counter (PC)** - 16 bits The address of the next instruction to execute. The CPU reads from wherever the PC points, executes, and advances the PC. Jumps and branches modify it to go to other parts of the code.
- **Stack Pointer (SP)** - 8 bits pointing to the top of the stack in memory.
- **Status (P)** - 8 bits This one is special: it doesn't store data, it stores *state*. Each bit is a flag that tells you something about what just happened:

1	7	6	5	4	3	2	1	0
2	N	V	U	B	D	I	Z	C

- **N (Negative)**: The result was negative (bit 7 = 1)
- **V (Overflow)**: There was overflow in signed operations
- **U (Unused)**: Always 1, not used
- **Z (Zero)**: The result was zero

- **C (Carry)**: There was a carry in addition or a borrow in subtraction
- **I (Interrupt Disable)**: IRQ interrupts are disabled
- **D (Decimal)**: Decimal mode. The NES ignores it, but the flag exists
- **B (Break)**: The interrupt was triggered by software (BRK)

What's the point of all this? Decision making. Branch instructions check these flags to decide whether to jump or not. For example: BEQ (Branch if Equal) jumps if the Z flag is 1, meaning the last operation gave a zero result. That's how you implement an `if` in assembly: you do a comparison (which sets the flags), then a branch that checks the relevant flag.

The Status register is what gives a program the ability to make decisions. Without it, the CPU could only execute instructions in sequence, blindly marching forward.

OceanofPDF.com

How a CPU works (short version)

A CPU does the same thing over and over, in an infinite loop:

1. **Fetch:** Read the byte at the address the PC (program counter) points to
2. **Decode:** Figure out what instruction it is and how to get its operands
3. **Execute:** Execute the instruction
4. **Repeat:** Advance the PC accordingly and go back to step 1

Some instructions are 1 byte long (just the opcode), others are 2 (opcode + one operand), others are 3 (opcode + a 16-bit address). The PC advances by the right amount after each one.

For example:

```
1 PC = 0x0000
2
3 1. Read the byte at 0x0000 -> it's 0xA9 (LDA immediate
instruction)
4 2. Advance PC by 1 -> 0x0001
5 3. LDA needs an operand, read the byte at 0x0001 (PC) -> it's
0x42
6 4. Advance PC by 1 -> 0x0002
7 5. Execute: load 0x42 into register A
8 6. Go back to step 1, now with PC at 0x0002
9
10 (0x is a prefix indicating the number is in hexadecimal format)
```

Each instruction takes a certain number of cycles to execute. A NOP takes 2 cycles, an LDA can take anywhere from 2 to 6 depending on the addressing mode. This matters because the PPU (the graphics chip) runs in parallel and expects the CPU to take the right amount of time. If your emulator executes everything instantly, synchronization breaks down. Don't worry about this for now, we'll deal with it later.

High-level project structure

This will be our file structure:

```
1 nes-emulator/  
2 |— bin/  
3 |   └─ test_cpu.cr      # Entry point for testing  
4 |— src/  
5 |   └─ nes.cr          # Main module  
6 |       └─ nes/  
7 |           └─ emulator.cr # Coordinates components  
8 |           └─ bus.cr      # Communication between components  
9 |           └─ cpu.cr      # The CPU
```

The philosophy is simple: one class per component. As we go, each NES module (CPU, PPU, APU, etc.) will get its own file. We'll also break things into smaller, focused files to keep complexity manageable and make everything easier to read and debug.

The **Emulator** is the top-level coordinator. The **Bus** is the communication layer: in real hardware, the CPU doesn't talk directly to memory or the PPU. Everything goes through a data bus. We replicate that because it keeps the code clean and true to the real architecture.

Here's what goes in each file for now:

```
1 # src/nes.cr  
2  
3 require "../nes/emulator"  
4  
5 module NES  
6   VERSION = "1.0"  
7 end
```

```
1 # src/nes/emulator.cr  
2  
3 require "../bus"  
4 require "../cpu"  
5
```

```
6 module NES
7   class Emulator
8     getter cpu : CPU
9     getter bus : Bus
10
11     def initialize # this is how you call the constructor in Ruby
and Crystal
12       @bus = Bus.new
13       @cpu = CPU.new(@bus)
14     end
15
16     def whats_up_bro
17       puts "Whats up bro"
18     end
19   end
20 end
```

```
1 # src/nes/cpu.cr
2
3 module NES
4   class CPU
5     def initialize(bus : Bus)
6       @bus = bus
7     end
8   end
9 end
```

```
1 # src/nes/cpu.cr
2
3 module NES
4   class Bus
5     end
6 end
```

```
1 # bin/test_cpu.cr
2
3 require "../src/nes"
4
5 emulator = NES::Emulator.new
6 emulator.whats_up_bro
```

The code for this part is [here](#)

Now run this to make sure everything is working:

```
1 crystal run bin/test_cpu.cr
```

You should see “Whats up bro”

If you see that, you’re good to go. If not, check that you have Crystal installed correctly (go back to chapter 2 if needed).

Alright, now let’s make the CPU do something useful.

OceanofPDF.com

Chapter 3.1: Our first two instructions

We're going to start with two instructions: LDA (Load Accumulator) and INX (Increment X). They're simple, but enough to build the entire CPU structure and verify that it works.

OceanofPDF.com

A bit of hexadecimal

If you're not comfortable with hexadecimal yet, now's the time to get there. We'll be using it everywhere: memory addresses, register values, opcodes. It's the language of low-level programming.

Hexadecimal is base 16. After 9 comes A, B, C, D, E, F. Like this:

Decimal	Hexadecimal	Binary
0	0x00	0000 0000
1	0x01	0000 0001
2	0x02	0000 0010
3	0x03	0000 0011
4	0x04	0000 0100
5	0x05	0000 0101
6	0x06	0000 0110
7	0x07	0000 0111
8	0x08	0000 1000
9	0x09	0000 1001
10	0x0A	0000 1010
11	0x0B	0000 1011
12	0x0C	0000 1100
13	0x0D	0000 1101
14	0x0E	0000 1110
15	0x0F	0000 1111
16	0x10	0001 0000
...	...	...
255	0xFF	1111 1111

Why hexadecimal and not binary? Because it's more compact. A byte (8 bits) is always represented with exactly two hexadecimal digits. 0xFF is easier to read than 11111111.

The Windows calculator (Programmer mode) or any online calculator can convert between bases. Use it until you get the hang of it.

OceanofPDF.com

Types in Crystal

Crystal is statically typed, and when working with bytes we need to be explicit about sizes. Here are the suffixes you'll see all the time:

- `_u8` => unsigned 8 bits (0 to 255)
- `_u16` => unsigned 16 bits (0 to 65535)

For example:

```
1 value = 0xA9_u8      # 8 bits, the LDA opcode
2 address = 0x8000_u16  # 16 bits, a memory address
```

If you come from dynamic languages like Ruby or Python, this might feel verbose. But when you're emulating hardware, you need exact control over how many bits everything takes up. An 8-bit register has to be 8 bits, not 64.

OceanofPDF.com

The opcodes

Each CPU instruction has a numeric code called an opcode. When the CPU reads 0xA9 from memory, it knows it has to execute LDA immediate. When it reads 0xE8, it executes INX.

We'll centralize all opcodes in a module:

```
1  # src/nes/cpu/op_codes.cr
2
3  module NES
4    class CPU
5      module OpCodes
6        # LDA
7        CODE_LDA_IMMEDIATE = 0xA9_u8
8
9        # Register Increments
10       CODE_INX = 0xE8_u8
11
12       # Cycles per Operation Code
13       CYCLES = {
14         CODE_LDA_IMMEDIATE => 2,
15         CODE_INX => 2
16       }
17     end
18   end
19 end
```

0xA9 in binary is 10101001. That's what the real CPU saw: a sequence of ones and zeros that triggered specific circuits. We're simulating that behavior in software.

The CYCLES hash stores how many cycles each instruction takes. We'll need it later to synchronize with the PPU.

The CPU

Alright, now the CPU for real. Create `src/nes/cpu.cr`. We'll build it up piece by piece.

First the basic structure with the registers:

```
1 # src/nes/cpu.cr
2
3 require "./cpu/op_codes"
4 require "./cpu/instructions"
5 require "./cpu/flags"
6
7 module NES
8   class CPU
9     include Flags
10    include Instructions
11    include OpCodes
12
13    getter a      : UInt8  # Accumulator
14    getter x      : UInt8  # X register
15    getter y      : UInt8  # Y register
16    getter sp     : UInt8  # Stack Pointer
17    getter pc     : UInt16 # Program Counter
18    getter status : UInt8  # Status register (flags)
```

The getters are the registers we saw before: A, X, Y, SP, PC, and Status.

Now the initialize:

```
1 # src/nes/cpu.cr
2
3 def initialize(bus : Bus)
4   @bus = bus
5   @a   = 0_u8 # Remember that _u8 means it's an 8-bit UInt
6   @x   = 0_u8
7   @y   = 0_u8
8   @sp  = 0xFD_u8
9   @status = 0x24_u8
10  @pc   = 0_u16 # Same here, _u16 means a 16-bit UInt
11 end
```

Why `@sp = 0xFD_u8`? That's what the stack pointer is set to when the 6502 powers on.

And `@status = 0x24_u8`? In binary that's `00100100`. It turns on the U (Unused, always 1) and I (Interrupt Disable) flags. Same deal as `@sp`: this is just how the processor starts up.

Now the heart of everything, the `step` method:

```
1  # src/nes/cpu.cr
2
3  def step
4    opcode = fetch_byte
5
6    case opcode
7      when OpCodes::CODE_LDA_IMMEDIATE then op_lda_immediate
8      when OpCodes::CODE_INX           then op_inx
9    end
10
11    CYCLES[opcode]
12  end
```

It reads the next instruction (opcode), decodes it with the `case`, executes it, and returns how many cycles it took. That's it.

As we implement more instructions, this `case` will grow. For now it only has two.

Finally, the methods for reading from memory:

```
1  # src/nes/cpu.cr
2
3  def fetch_byte : UInt8
4    value = read_byte(@pc)
5    @pc &+= 1
6
7    value
8  end
9
10 def read_byte(address : UInt16) : UInt8
11   @bus.cpu_read(address)
12 end
```

```
13     end
14 end
```

`fetch_byte` reads the byte at the current Program Counter position and advances the PC.

Notice the `&+=`. That's wrapping addition. If the PC is at `0xFFFF` and we add 1, it wraps around to `0x0000` instead of throwing an error. That's how real hardware behaves.

OceanofPDF.com

The instructions

We're going to organize instructions into separate modules. Starting with LDA:

```
1  # src/nes/cpu/instructions/lda.cr
2
3  module NES
4    class CPU
5      module Instructions
6        module LDA
7          def lda(value)
8            @a = value
9
10           set_z_flag(@a)
11           set_n_flag(@a)
12         end
13
14         def op_lda_immediate
15           value = fetch_byte
16
17           lda(value)
18         end
19       end
20     end
21   end
22 end
```

LDA immediate reads the next byte from the program and stores it in the Accumulator. Then it updates the Z (Zero) and N (Negative) flags based on the loaded value.

How do I know which flags each instruction affects? The [NES Wiki](#) has it all. Every instruction is documented with exactly what it does and which flags it touches.

Now INX:

```
1  # src/nes/cpu/instructions/register_increments.cr
2
```

```
3 module NES
4   class CPU
5     module Instructions
6       module RegisterIncrements
7         def op_inx
8           @x &+= 1
9
10          set_z_flag(@x)
11          set_n_flag(@x)
12        end
13      end
14    end
15  end
16 end
```

INX increments X by 1. If X is at 255 and you increment it, it wraps back to 0 (because of &+). It also updates Z and N.

And the umbrella module that ties everything together, **Instructions**:

```
1 # src/nes/cpu/instructions.cr
2
3 require "../instructions/lda"
4 require "../instructions/register_increments"
5
6 module NES
7   class CPU
8     module Instructions
9       include LDA
10      include RegisterIncrements
11    end
12  end
13 end
```

This one-file-per-instruction-family pattern will be a lifesaver once we have all 56 instructions in place. Small files, easy to find.

OceanofPDF.com

The flags

Flags are individual bits inside the Status register. We need to be able to flip them on and off.

First we define the masks, one for each flag:

```
1 # src/nes/cpu/flags.cr
2
3 module NES
4   class CPU
5     module Flags
6       # Flag Masks (N V U B D I Z C)
7       FLAG_C = 0b0000_0001_u8 # Carry
8       FLAG_Z = 0b0000_0010_u8 # Zero
9       FLAG_I = 0b0000_0100_u8 # Interrupt Disable
10      FLAG_D = 0b0000_1000_u8 # Decimal
11      FLAG_B = 0b0001_0000_u8 # Break
12      FLAG_U = 0b0010_0000_u8 # Unused (always 1)
13      FLAG_V = 0b0100_0000_u8 # Overflow
14      FLAG_N = 0b1000_0000_u8 # Negative
```

Each constant has a single bit set in its corresponding position. FLAG_Z = 0b0000_0010 has bit 1 set.

Why “masks”? Because we use them to cover up all the bits we don’t care about and target just one specific bit. Think painter’s tape: it covers everything except the part you want to paint.

Now the methods that set the most common flags:

```
1 # src/nes/cpu/flags.cr
2
3   def set_z_flag(register_value : UInt8)
4     is_zero = register_value == 0
5     set_flag(FLAG_Z, is_zero)
6   end
7
8   def set_n_flag(register_value : UInt8)
9     seventh_bit = register_value.bit(7)
10    is_on = seventh_bit == 1
```

```
11         set_flag(FLAGS_N, is_on)
12     end
```

set_z_flag turns the flag on when the value is zero. set_n_flag turns the flag on when bit 7 (the most significant bit) is 1, which means the number is negative in two's complement.

And the helper that does the actual dirty work:

```
1  # src/nes/cpu/flags.cr
2
3      def set_flag(mask : UInt8, on : Bool)
4          if on
5              @status |= mask
6          else
7              @status &= (~mask).to_u8
8          end
9      end
10 end
11 end
12 end
```

The set_flag method uses bitwise operations:

- To turn a bit on: OR with the mask (|=)
- To turn a bit off: AND with the inverted mask (&= ~mask)

Let's walk through a concrete example. Say we want to turn on the Z (Zero) flag in a status register that's currently 0x00:

```
1 status = 0000 0000
2 FLAG_Z = 0000 0010
3
4 status | FLAG_Z = 0000 0010  # Bit 1 is now on
```

And now we want to turn it back off:

```
1 status = 0000 0010
2 FLAG_Z = 0000 0010
3 ~FLAG_Z = 1111 1101  # All bits inverted
```

4

5 `status & ~FLAG_Z = 0000 0000` # *Bit 1 is now off*

OR turns bits on, AND with an inverted mask turns them off. It clicks once you see it in action.

OceanofPDF.com

The Bus

The CPU doesn't access memory directly. Everything goes through the Bus. First we define the address constants:

```
1 # src/nes/bus/addresses.cr
2
3 module NES
4   class Bus
5     module Addresses
6       RAM_SIZE = 2048 # 2 KB
7
8       RAM_START = 0x0000_u16
9       RAM_END   = 0x07FF_u16
10      RAM_MIRROR_END = 0x1FFF_u16
11
12      RAM_MASK = 0x07FF_u16
13    end
14  end
15 end
```

What's mirroring? The NES only has 2KB of RAM (addresses 0x0000 to 0x07FF), but the address space reserved for RAM goes all the way up to 0x1FFF. Addresses 0x0800 through 0x1FFF are "mirrors" of the first 2KB.

Writing to 0x0800 actually writes to 0x0000. Writing to 0x1000 also hits 0x0000. It's a hardware trick to simplify the address decoder.

$\text{RAM_MASK} = 0x07FF$ is how we calculate the real address. $0x0800 \ \& \ 0x07FF = 0x0000$. Bit magic.

Now the Bus itself:

```
1 # src/nes/bus.cr
2
3 require "../bus/addresses"
4
5 module NES
6   class Bus
7     include Addresses
```

```
8
9     def initialize
10         @ram = Bytes.new(RAM_SIZE, 0_u8)
11     end
```

The RAM is simply a byte array initialized to zero. `Bytes.new(2048, 0_u8)` gives us 2KB of memory ready to use.

The read and write methods:

```
1  # src/nes/bus.cr
2
3     def cpu_read(address : UInt16) : UInt8
4         case address
5             when RAM_START..RAM_MIRROR_END
6                 read_ram(address)
7             else
8                 0_u8
9             end
10    end
11
12    def cpu_write(address : UInt16, value : UInt8)
13        case address
14            when RAM_START..RAM_MIRROR_END
15                write_ram(address, value)
16            end
17    end
```

Why `cpu_read` and not just `read`? Because later the PPU will also read and write through the Bus, but with different rules. We'll have `cpu_read`, `cpu_write`, `ppu_read`, and `ppu_write`. The prefix helps us distinguish who's accessing what.

The case determines which component handles that address. For now we only have RAM, but this is where we'll later add the PPU, APU, cartridges, etc.

And the internal methods that apply mirroring:

```
1  # src/nes/bus.cr
2
3     def read_ram(address : UInt16) : UInt8
```

```
4     mirrored_address = (address & RAM_MASK)
5     @ram[mirrored_address]
6 end
7
8 def write_ram(address : UInt16, value : UInt8)
9     mirrored_address = (address & RAM_MASK)
10    @ram[mirrored_address] = value
11 end
12 end
13 end
```

& RAM_MASK is where the mirroring magic happens. $0x0800 \& 0x07FF = 0x0000$. Any address in the mirror range automatically maps back to its real address.

OceanofPDF.com

The Emulator

The Emulator coordinates everything:

```
1 # src/nes/emulator.cr
2
3 require "./bus"
4 require "./cpu"
5 require "./emulator/debugging"
6
7 module NES
8   class Emulator
9     include Debugging
10
11     getter cpu : CPU
12     getter bus : Bus
13
14     def initialize
15       @bus = Bus.new
16       @cpu = CPU.new(@bus)
17     end
```

Nothing fancy: create a Bus, create a CPU connected to it.

The step method executes one step of the emulation:

```
1 # src/nes/emulator.cr
2
3   def step
4     cycles = @cpu.step
5     # TODO: PPU steps cycles * 3
6   end
```

For now it only ticks the CPU forward. When we have the PPU, this is where we'll run it 3 cycles for every CPU cycle.

And a helper for loading test programs:

```
1 # src/nes/emulator.cr
2
```

```
3     def load_program(program : Bytes, start_address : UInt16 =  
Bus::RAM_START)  
4         program.each_with_index do |byte, i|  
5             @bus.cpu_write((start_address &+ i), byte)  
6         end  
7     end  
8 end  
9 end
```

load_program takes a byte array and writes it to memory starting at a given address. That way we can load test programs without needing a cartridge yet.

OceanofPDF.com

Debugging

To peek at what's going on under the hood, we'll add a debugging module to the Emulator:

```
1 # src/nes/emulator/debugging.cr
2
3 module NES
4   class Emulator
5     module Debugging
6       def debug
7         puts "    A=#{hex8(@cpu.a)} X=#{hex8(@cpu.x)} Y=#{
hex8(@cpu.y)} SP=#{hex8(@cpu.sp)} PC=#{hex16(@cpu.pc)}    [#
{status_string}]"
8       end
9     end
10  end
11 end
```

debug prints the full CPU state in a single line. You'll use it after each step to watch the registers change.

The hex formatting helpers:

```
1 # src/nes/emulator/debugging.cr
2
3 def hex8(value : UInt8) : String
4   "$#{value.to_s(16).rjust(2, '0').upcase}"
5 end
6
7 def hex16(value : UInt16) : String
8   "$#{value.to_s(16).rjust(4, '0').upcase}"
9 end
```

hex8 formats a byte as \$0A, hex16 formats an address as \$8000. The \$ prefix is 6502 assembler convention for hexadecimal.

And for displaying the flags in a human-readable way:

```
1 # src/nes/emulator/debugging.cr
2
3 def status_string : String
```

```

4      s = @cpu.status
5      String.build do |str|
6          str << (s.bit(7) == 1 ? 'N' : 'n')
7          str << (s.bit(6) == 1 ? 'V' : 'v')
8          str << (s.bit(5) == 1 ? 'U' : 'u')
9          str << (s.bit(4) == 1 ? 'B' : 'b')
10         str << (s.bit(3) == 1 ? 'D' : 'd')
11         str << (s.bit(2) == 1 ? 'I' : 'i')
12         str << (s.bit(1) == 1 ? 'Z' : 'z')
13         str << (s.bit(0) == 1 ? 'C' : 'c')
14     end
15 end
16 end
17 end
18 end

```

Uppercase = flag active, lowercase = flag inactive. nvUbdIzc means only U and I are on (the CPU's initial state).

The module is included in the Emulator with `include Debugging`, so you can call `emu.debug` directly.

Tests

We'll write one test file per instruction. You can run them individually or all at once.

The main runner:

```
1 # bin/test_cpu.cr
2
3 require "./cpu/test_lda"
4
5 puts "\n"
6 puts "#" * 60
7 puts "\n"
8
9 require "./cpu/test_inx"
```

Each `require` runs the corresponding test file. Straightforward.

Let's look at the LDA test in `bin/cpu/test_lda.cr`. First the structure:

```
1 # bin/cpu/test_lda.cr
2
3 require "../../src/nes"
4
5 puts "LDA Immediate"
6
7 all_passed = true
```

We pull in the emulator and set up a variable to track whether all tests pass.

A typical test looks like this:

```
1 # Test 1: Load value
2 puts "\n 1. Load value"
3 program = Bytes[
4   0xA9, # LDA immediate
5   0x42  # value
6 ]
7
```

```

8  emu = NES::Emulator.new
9  emu.load_program(program)
10
11  puts "      LDA #$42"
12  emu.step
13  emu.debug
14
15  passed = emu.cpu.a == 0x42 && emu.cpu.status.bit(1) == 0 &&
emu.cpu.status.bit(7) == 0
16  puts passed ? "       " : "       "
17  all_passed &= passed

```

We create a tiny 2-byte program: the opcode `0xA9` (LDA immediate) and the value `0x42`. Load it up, execute one step, and check that register A has the right value and the flags are correct.

The full test also checks the Zero flag (loading `0x00`) and the Negative flag (loading `0x80`). Same structure, different values.

At the end:

```

1  puts all_passed ? "\n LDA passed" : "\n LDA failed"

```

The INX test in `bin/cpu/test_inx.cr` is similar. It checks:

1. Simple increment (X goes from 0 to 1)
2. Wrap around (X goes from 255 to 0, Z flag turns on)
3. Negative flag (X reaches 128, bit 7 set)

You can find the full test files in the [repository](#).

Run `crystal run bin/test_cpu.cr` and you should see everything passing. If not, you've got a bug. Welcome to emulation.

Final structure

Just to make sure you have everything, here's what your project should look like:

```
1 nes-emulator/
2 |─ bin/
3 |   |─ test_cpu.cr
4 |   |─ cpu/
5 |       |─ test_lda.cr
6 |       |─ test_inx.cr
7 |─ src/
8 |   |─ nes.cr
9 |   |─ nes/
10 |       |─ bus.cr
11 |       |─ cpu.cr
12 |       |─ emulator.cr
13 |       |─ bus/
14 |           |─ addresses.cr
15 |       |─ cpu/
16 |           |─ flags.cr
17 |           |─ instructions.cr
18 |           |─ op_codes.cr
19 |           |─ instructions/
20 |               |─ lda.cr
21 |               |─ register_increments.cr
22 |─ emulator/
23 |   |─ debugging.cr
```

If something isn't working, you can compare with the code in the repository:

[Github](#)

OceanofPDF.com

Wrap up

Congratulations, your CPU can do two whole things: load a number into A and add 1 to X. It's like a baby that just learned to say "mama" and "dada." Not much, but it's honest work.

Now comes the fun part (or the tedious part, depending on your perspective): implementing the other 54 instructions. I personally got bored halfway through, so I won't blame you if you do too.

We'll implement enough instructions together that you'll be able to handle the rest on your own. My recommendation: do at least 10 or 15 yourself before copying from the repo. That's where you really internalize how the CPU works, and after that it's all more of the same. Once you feel like you've got the pattern down, I'll give you the link to the repository with everything implemented so you can move on.

[OceanofPDF.com](https://oceanofpdf.com)

Chapter 3.2: Store Instructions and Addressing Modes

In the previous chapter we implemented LDA (load a value into A) and INX (increment X). Pretty cool, but if we can't write values to memory, we can't do much.

We're going to implement STA (Store Accumulator): it takes whatever's in register A and writes it to a memory address.

But here's where it gets interesting: *which* address? The 6502 has several ways of specifying where to read from or write to. They're called **addressing modes**, and we're going to look at two of them: **Zero Page** and **Absolute**.

OceanofPDF.com

Addressing Modes

Addressing modes are the different ways an instruction can figure out where its data lives. Think of it this way: STA always does the same thing (store A in memory), but *how* you tell it where to store can vary.

Why so many ways? Optimization. In the 80s, every byte and every cycle counted. Some modes are faster but limited, others are more flexible but slower. The programmer picked whichever fit the situation.

These are all the addressing modes of the 6502:

Mode	Example	Description
Immediate	LDA #\$42	The operand is the literal value
Zero Page	STA \$10	Address in the first 256 bytes
Zero Page,X	STA \$10,X	Zero Page + register X
Zero Page,Y	LDA \$10,Y	Zero Page + register Y
Absolute	STA \$1234	Full 16-bit address
Absolute,X	STA \$1234,X	Absolute + register X
Absolute,Y	STA \$1234,Y	Absolute + register Y
Indirect	JMP (\$1234)	The address contains another address
Indexed Indirect	LDA (\$10,X)	Zero Page + X, then indirect
Indirect Indexed	LDA (\$10),Y	Indirect, then + Y
Relative	BEQ \$10	Offset from PC (for branches)
Implied	INX	No operand needed
Accumulator	ASL A	Operates on register A

Don't panic. We'll implement all of these in section 3.3. For now we're just tackling Zero Page and Absolute.

Zero Page

The first 256 bytes of memory (addresses \$0000 to \$00FF) are called **Zero Page**. Why do they get a special name? Because accessing them is faster.

A Zero Page address only needs 1 byte to be specified. If you want to store something at \$0010, the instruction is:

```
1 85 10
```

Two bytes: the opcode (\$85 = STA Zero Page) and the address (\$10).

But if you want to store something at \$0200 (outside Zero Page), you need:

```
1 8D 00 02
```

Three bytes: the opcode (\$8D = STA Absolute) and the full address in two bytes.

Saving one byte doesn't sound like much, but:

- Fewer bytes = less memory used by the program
- Fewer bytes = fewer fetch cycles
- STA Zero Page takes 3 cycles, STA Absolute takes 4

Back in 1985, programmers put their most important variables in Zero Page: player position, loop counters, pointers. Anything that got accessed constantly lived there.

Absolute

Absolute mode uses a full 16-bit address. You can access any location in the 64KB of memory.

This is where we run into something important: **little-endian**. The 6502 stores addresses with the low byte first.

If you want to write to address \$0200, it shows up in memory like this:

1	8D	00	02	
2				
3				High byte (\$02)
4				Low byte (\$00)
5				Opcode (STA Absolute)

So \$0200 is stored as 00 02, not 02 00. It feels backwards at first, but you get used to it fast. And it's not some 6502 quirk: Intel x86 processors also use little-endian, so if you've ever done low-level debugging on a modern PC, you've seen this before.

Here's another example. To write to \$1234:

1	8D	34	12
---	----	----	----

First \$34 (low byte), then \$12 (high byte).

Implementation

Let's write some code. First we add the opcodes:

```
1  # src/nes/cpu/op_codes.cr
2
3  module NES
4    class CPU
5      module OpCodes
6        # LDA
7        CODE_LDA_IMMEDIATE = 0xA9_u8
8
9        # STA
10       CODE_STA_ZERO_PAGE = 0x85_u8
11       CODE_STA_ABSOLUTE = 0x8D_u8
12
13       # Register Increments
14       CODE_INX = 0xE8_u8
15
16       # Cycles per Operation Code
17       CYCLES = {
18         CODE_LDA_IMMEDIATE => 2,
19         CODE_STA_ZERO_PAGE => 3,
20         CODE_STA_ABSOLUTE => 4,
21         CODE_INX           => 2
22       }
23     end
24   end
25 end
```

Notice STA Zero Page takes 3 cycles while STA Absolute takes 4. That extra cycle is the cost of reading one more byte from memory.

Now we create a module for the addressing modes:

```
1  # src/nes/cpu/addressing_modes.cr
2
3  module NES
4    class CPU
5      module AddressingModes
6        # Zero Page: reads 1 byte, it's the direct address in page
```

```

zero
7      def address_zero_page : UInt16
8          fetch_byte.to_u16
9      end
10
11      # Absolute: reads 2 bytes (little-endian) to form a 16-bit
address
12      def address_absolute : UInt16
13          lo = fetch_byte
14          hi = fetch_byte
15          (hi.to_u16 << 8) | lo.to_u16
16      end
17  end
18 end
19 end

```

address_zero_page is dead simple: read a byte and widen it to a 16-bit address.

address_absolute reads two bytes (low first, then high) and combines them:

- `hi.to_u16 << 8` moves the high byte to its position (e.g.: `$02 => $0200`)
- `| lo.to_u16` adds the low byte (e.g.: `$0200 | $34 => $0234`)

Now the STA instruction:

```

1  # src/nes/cpu/instructions/sta.cr
2
3  module NES
4      class CPU
5          module Instructions
6              module STA
7                  # STA doesn't affect any flags
8
9                  def op_sta_zero_page
10                     address = address_zero_page
11                     write_byte(address, @a)
12                 end
13
14                 def op_sta_absolute
15                     address = address_absolute

```

```
16         write_byte(address, @a)
17     end
18 end
19 end
20 end
21 end
```

Both do the same thing: get the address (using the appropriate addressing mode) and write A's value there. The only difference is how they figure out the address.

Note that STA doesn't touch any flags. It just writes the value and moves on.

We add STA to the instructions module:

```
1  # src/nes/cpu/instructions.cr
2
3  require "./instructions/lda"
4  require "./instructions/sta"
5  require "./instructions/register_increments"
6
7  module NES
8    class CPU
9      module Instructions
10        include LDA
11        include STA
12        include RegisterIncrements
13      end
14    end
15  end
```

The CPU needs to include the addressing modes module and get a `write_byte` method:

```
1  # src/nes/cpu.cr
2
3  require "./cpu/op_codes"
4  require "./cpu/addressing_modes"
5  require "./cpu/instructions"
6  require "./cpu/flags"
7
8  module NES
```

```

9   class CPU
10      include Flags
11      include AddressingModes
12      include Instructions
13      include OpCodes
14
15      # ... getters and initialize same as before ...
16
17      def step
18          opcode = fetch_byte
19
20          case opcode
21              # LDA
22              when CODE_LDA_IMMEDIATE then op_lda_immediate
23              # STA
24              when CODE_STA_ZERO_PAGE then op_sta_zero_page
25              when CODE_STA_ABSOLUTE then op_sta_absolute
26              # Register Increments
27              when CODE_INX              then op_inx
28          end
29
30          CYCLES[opcode]
31      end
32
33      def fetch_byte : UInt8
34          value = read_byte(@pc)
35          @pc &+= 1
36          value
37      end
38
39      def read_byte(address : UInt16) : UInt8
40          @bus.cpu_read(address)
41      end
42
43      def write_byte(address : UInt16, value : UInt8)
44          @bus.cpu_write(address, value)
45      end
46  end
47 end

```

The case now has the two new STA opcodes. And we added write_byte, the counterpart to read_byte.

Tests

Let's create the test for STA in `bin/cpu/test_sta.cr`:

```
1 # bin/cpu/test_sta.cr
2
3 require "../src/nes"
4
5 puts "STA"
6
7 all_passed = true
```

Zero Page test:

```
1 # Test 1: STA Zero Page
2 puts "\n 1. Zero Page"
3 program = Bytes[
4   0xA9, # LDA immediate
5   0x42, # value
6   0x85, # STA zero page
7   0x10 # address $10
8 ]
9
10 emu = NES::Emulator.new
11 emu.load_program(program)
12
13 puts "    LDA #$42"
14 emu.step
15 emu.debug
16
17 puts "    STA $10"
18 emu.step
19 emu.debug
20
21 stored_value = emu.bus.cpu_read(0x10_u16)
22 passed = stored_value == 0x42
23 puts "    MEM[$10] = ${stored_value.to_s(16).rjust(2,
'0').uppercase}"
24 puts passed ? "     " : "     "
25 all_passed &= passed
```

We load \$42 into A, store it at \$10, and check that memory has the right value.

Absolute test:

```
1  # Test 2: STA Absolute
2  puts "\n 2. Absolute"
3  program = Bytes[
4      0xA9, # LDA immediate
5      0x7F, # value
6      0x8D, # STA absolute
7      0x00, # low byte
8      0x02 # high byte ($0200)
9  ]
10
11  emu = NES::Emulator.new
12  emu.load_program(program)
13
14  puts "    LDA #$7F"
15  emu.step
16  emu.debug
17
18  puts "    STA $0200"
19  emu.step
20  emu.debug
21
22  stored_value = emu.bus.cpu_read(0x0200_u16)
23  passed = stored_value == 0x7F
24  puts "    MEM[$0200] = ${stored_value.to_s(16).rjust(2,
'0').upcase}"
25  puts passed ? "    ✓" : "    ✗"
26  all_passed &= passed
```

Notice how the address \$0200 is written in the program as 0x00, 0x02 (little-endian).

And we make sure STA doesn't mess with the flags:

```
1  # Test 3: STA doesn't modify flags
2  puts "\n 3. Doesn't modify flags"
3  program = Bytes[
4      0xA9, # LDA immediate
5      0x00, # value (sets Z flag)
```

```

6    0x85, # STA zero page
7    0x10 # address
8 ]
9
10 emu = NES::Emulator.new
11 emu.load_program(program)
12
13 puts "    LDA #$00"
14 emu.step
15 emu.debug
16 status_before = emu.cpu.status
17
18 puts "    STA $10"
19 emu.step
20 emu.debug
21 status_after = emu.cpu.status
22
23 passed = status_before == status_after
24 puts "    Status unchanged: #{status_before == status_after}"
25 puts passed ? "    ✓ " : "    ✗ "
26 all_passed &= passed
27
28 puts all_passed ? "\n✓ STA passed" : "\n✗ STA failed"

```

Update the test runner:

```

1 # bin/test_cpu.cr
2
3 require "./cpu/test_lda"
4
5 puts "#" * 60
6 puts "\n"
7
8 require "./cpu/test_sta"
9
10 puts "#" * 60
11 puts "\n"
12
13 require "./cpu/test_inx"

```

Run `crystal run bin/test_cpu.cr` and all three tests should pass.

Updated structure

```
1 nes-emulator/
2 |─ bin/
3 |   |─ test_cpu.cr
4 |   |─ cpu/
5 |       |─ test_lda.cr
6 |       |─ test_sta.cr      <= New
7 |       |─ test_inx.cr
8 |─ src/
9 |   |─ nes/
10 |       |─ cpu.cr
11 |       |─ cpu/
12 |           |─ addressing_modes.cr  <= New
13 |           |─ flags.cr
14 |           |─ instructions.cr
15 |           |─ op_codes.cr
16 |           |─ instructions/
17 |               |─ lda.cr
18 |               |─ sta.cr          <= New
19 |               |─ register_increments.cr
```

If something isn't working: [Github](#)

[OceanofPDF.com](#)

Wrap up

Now you know what addressing modes are and why they exist. You've implemented Zero Page and Absolute, the two most common ones.

In the next section we'll implement the rest of the addressing modes and round out the Load and Store instruction families. Once you have all the modes working, adding new instructions becomes almost mechanical.

OceanofPDF.com

Chapter 3.3: All Addressing Modes

In the previous chapter we covered Zero Page and Absolute. Now we're going to implement the rest of the addressing modes. Once we're done, you'll be able to add any instruction without worrying about how to get the address.

Quick recap: addressing modes are the different ways an instruction can get its operand. The 6502 has 13 modes. We've already seen some, others are just variations, and a few are pretty convoluted.

OceanofPDF.com

The ones we already know

Immediate: The operand is the literal value right after the opcode. LDA #\$42 loads \$42 directly. No method needed in the module since it's just a fetch_byte.

Zero Page: An 8-bit address in the first 256 bytes of memory. Already implemented.

Absolute: A full 16-bit address. Already implemented.

Implied: The instruction doesn't need an operand. INX increments X, nothing to fetch. No method needed here either.

Accumulator: The instruction operates on the A register. For example, ASL A does a shift left on the accumulator. Handled inside the instruction itself, not in the addressing mode.

OceanofPDF.com

Zero Page with index

This is where things get interesting. What if you want to access an address that depends on a register?

Zero Page,X adds the X register to a Zero Page address:

```
1 LDA $80,X    ; If X = $10, loads from $0090
```

It's great for walking through arrays. You put the array's base address in the instruction and use X as the index.

```
1 # Zero Page + X offset
2 # E.g.: base=$80, X=$10 => $0090
3 def address_zero_page_x : UInt16
4     base = fetch_byte
5     (base &+ @x).to_u16
6 end
```

Read the base address, add X, done. The &+ is wrapping addition: \$FF + \$10 wraps around to \$0F instead of giving you \$10F. Zero Page always stays in Zero Page.

Zero Page,Y is the same thing but with Y:

```
1 # Zero Page + Y offset
2 # E.g.: base=$80, Y=$10 => $0090
3 def address_zero_page_y : UInt16
4     base = fetch_byte
5     (base &+ @y).to_u16
6 end
```

Some instructions use X, others Y. LDX uses Y (because indexing with X while you're loading X wouldn't make much sense). The NES Wiki tells you which register each instruction uses.

Absolute with index

Same idea but with 16-bit addresses:

Absolute,X:

```
1  # Absolute + X offset
2  def address_absolute_x : UInt16
3    (address_absolute &+ @x).to_u16
4  end
```

Absolute,Y:

```
1  # Absolute + Y offset
2  def address_absolute_y : UInt16
3    (address_absolute &+ @y).to_u16
4  end
```

We reuse `address_absolute` and add the index. Simple.

These modes are slower than Zero Page (longer addresses = more bytes to read), but they give you access to the entire memory.

OceanofPDF.com

Relative

This one is used exclusively by branch instructions (BEQ, BNE, BCS, etc.).

Instead of an absolute address, you use an **offset** relative to the current Program Counter. The offset is a signed byte: it can range from -128 to +127.

How does this work in practice? Imagine this code:

```
1 $8000: LDA #$05      ; Load 5 into A
2 $8002: SEC           ; Set the carry
3 $8003: SBC #$01      ; Subtract 1 (A = A - 1)
4 $8005: BNE $FC       ; If A != 0, jump -4 bytes (back to $8003)
5 $8007: BRK           ; Done
```

BNE \$FC has the offset \$FC, which is -4 (jump 4 bytes backward). Values from \$00 to \$7F are forward jumps (0 to 127), and \$80 to \$FF are backward jumps (-128 to -1). When the CPU takes the branch (assuming the condition is met), it does:

```
1 Current PC ($8007) + offset (-4) = $8003
```

And it goes back to SBC. It's a loop that counts down from 5 to 0.

You don't write the offset by hand, though. In assembly, you just write a label:

```
1 loop:
2   SBC #$01
3   BNE loop      ; The assembler figures out that "loop" is 4 bytes
back
```

The assembler handles the math and sticks \$FC in the machine code.

Why relative instead of absolute? Because branches are usually short jumps (go to the start of a loop, skip a few lines). A single byte is enough for that and saves space.

```
1 # Relative: used by branch instructions
2 # Reads a byte as a signed offset (-128 to +127) and adds it to
  the PC
3 def address_relative : UInt16
4   offset = fetch_byte.to_i8! # Interpret UInt8 as signed Int8
5   (@pc &+ offset.to_i16).to_u16!
6 end
```

The trick is `to_i8!`: it reinterprets the byte (0-255) as a signed number (-128 to +127). So `$F0` becomes -16, not 240.

OceanofPDF.com

Indirect

Here's where things get spicy. **Indirect** means the address you read isn't the final destination, it's a pointer to the final address.

Think of it like showing up at an address to pick something up, only to find a note that says "actually it's over here =>". You have to follow the note.

Only JMP uses the pure Indirect mode:

```
1  JMP ($1234)    ; Reads the address stored at $1234-$1235, and jumps
there
```

If \$1234 contains \$00 and \$1235 contains \$80, the JMP jumps to \$8000.

```
1  # Indirect: JMP ($nnnn)
2  # Reads a 16-bit address from the pointer location
3  def address_indirect : UInt16
4    pointer = address_absolute
5
6    lo = read_byte(pointer)
7
8    # Emulate 6502 page boundary bug
9    hi_addr = if (pointer & 0xFF) == 0xFF
10      pointer & 0xFF00 # Wrap to $xx00
11    else
12      pointer + 1
13    end
14
15    hi = read_byte(hi_addr)
16
17    (hi.to_u16 << 8) | lo.to_u16
18  end
```

What's that weird `if` about? A hardware bug in the original chip.

The 6502's memory is divided into "pages" of 256 bytes. The last byte of each page ends in \$FF:

1	\$00FF	<=	end of page 0 (Zero page)
2	\$01FF	<=	end of page 1
3	\$02FF	<=	end of page 2
4	...		
5	\$FEFF	<=	end of page 254
6	\$FFFF	<=	end of page 255

If the pointer lands on the last byte of a page (e.g. \$12FF), the 6502 doesn't read the high byte from \$1300 like you'd expect. It wraps around and reads from \$1200. The bug was in how the chip was wired: incrementing the low byte didn't carry into the high byte. Emulators replicate this to stay faithful to the hardware.

Does it matter in practice? Hardly ever. But if you don't emulate it, some game that happens to rely on this behavior will break.

OceanofPDF.com

Indexed Indirect: (\$nn,X)

This is the most confusing one. Deep breath.

Indexed Indirect (or “Indirect X”) works like this:

1. Read a byte (a Zero Page address)
2. Add X to it
3. At that Zero Page address, read a 16-bit pointer
4. That pointer is the final address

```
1 LDA ($20,X)    ; If X=$04, reads the pointer at $24-$25, and loads
from there
```

It's like having a table of pointers in Zero Page and using X to pick which one you want.

```
1 # Indexed Indirect: LDA ($nn,X)
2 # Adds X to the zero page address, then reads a 16-bit pointer
3 # E.g.: base=$20, X=$04 => reads pointer from $24 and $25
4 def address_indirect_x : UInt16
5     base = fetch_byte
6     ptr = base &+ @x
7
8     lo = read_byte(ptr.to_u16)
9     hi = read_byte((ptr &+ 1_u8).to_u16)
10
11     (hi.to_u16 << 8) | lo.to_u16
12 end
```

Step by step:

1. `base = fetch_byte` => we read \$20
2. `ptr = base &+ @x` => if X is \$04, ptr is \$24
3. `lo = read_byte(ptr)` => we read the pointer's low byte from \$24
4. `hi = read_byte(ptr + 1)` => we read the high byte from \$25
5. We combine them to get the final address

OceanofPDF.com

Indirect Indexed: (\$nn),Y

Similar but different. The order of operations changes:

1. Read a byte (a Zero Page address)
2. At that address, read a 16-bit pointer
3. Add Y to the pointer
4. That's the final address

```
1 LDA ($20),Y    ; Reads the pointer at $20-$21, adds Y, and loads
from there
```

It's meant for walking through data whose base address is stored in Zero Page.

```
1 # Indirect Indexed: LDA ($nn),Y
2 # Reads a 16-bit pointer from zero page, then adds Y
3 # E.g.: ($20) contains $0300, Y=$05 => final address $0305
4 def address_indirect_y : UInt16
5     base = fetch_byte
6
7     lo = read_byte(base.to_u16)
8     hi = read_byte((base &+ 1_u8).to_u16)
9
10    addr = (hi.to_u16 << 8) | lo.to_u16
11    addr &+ @y.to_u16
12 end
```

Step by step:

1. `base = fetch_byte` => we read \$20
2. `lo = read_byte(base)` => we read the pointer's low byte from \$20
3. `hi = read_byte(base + 1)` => we read the high byte from \$21
4. We combine them to get \$0300
5. `addr &+ @y` => if Y is \$05, the final address is \$0305

OceanofPDF.com

(\$nn,X) vs (\$nn),Y: which is which?

The difference is *when* the index gets added:

- **(\$nn,X)**: You add X *before* reading the pointer. You're choosing which pointer to use.
- **(\$nn),Y**: You add Y *after* reading the pointer. You use the pointer as a base and move from there.

Here's a trick to remember which is which: look at where the parentheses are. In (\$nn, X) the X is *inside* the parentheses (added before reading the pointer). In (\$nn), Y the Y is *outside* (added after).

Game example: (\$nn),Y for traversing a sprite

You have sprite data (8 bytes) stored somewhere in memory, with a Zero Page pointer telling you where:

```
1 ; $20-$21 contains $0300 (address where the sprite starts)
2 LDY #$00
3 loop:
4   LDA ($20),Y    ; Read byte Y of the sprite
5   STA $2004      ; Send it to the PPU
6   INY
7   CPY #$08      ; Have we read all 8 bytes?
8   BNE loop
```

The pointer (\$20) stays the same, but Y walks you through the sprite data.

Game example: (\$nn,X) for choosing between enemies

You have a table of pointers in Zero Page, each pointing to a different enemy's data:

```
1 ; $20-$21 => Goomba data
2 ; $22-$23 => Koopa data
3 ; $24-$25 => Lakitu data
4
```

```
5 LDX enemy_type    ; X = 0, 2, or 4 (each pointer takes 2 bytes)
6 LDA ($20,X)       ; Read the first byte of the selected enemy
```

X chooses *which* pointer to use. If enemy_type is 2, you read from the Koopa pointer.

OceanofPDF.com

The complete module

Here's the full `src/nes/cpu/addressing_modes.cr`:

```
1 module NES
2   class CPU
3     module AddressingModes
4       def address_zero_page : UInt16
5         fetch_byte.to_u16
6       end
7
8       def address_zero_page_x : UInt16
9         base = fetch_byte
10        (base &+ @x).to_u16
11      end
12
13      def address_zero_page_y : UInt16
14        base = fetch_byte
15        (base &+ @y).to_u16
16      end
17
18      def address_absolute : UInt16
19        lo = fetch_byte
20        hi = fetch_byte
21        (hi.to_u16 << 8) | lo.to_u16
22      end
23
24      def address_absolute_x : UInt16
25        (address_absolute &+ @x).to_u16
26      end
27
28      def address_absolute_y : UInt16
29        (address_absolute &+ @y).to_u16
30      end
31
32      def address_indirect_x : UInt16
33        base = fetch_byte
34        ptr = base &+ @x
35
36        lo = read_byte(ptr.to_u16)
37        hi = read_byte((ptr &+ 1_u8).to_u16)
38      end
39    end
40  end
41 end
```

```

39         (hi.to_u16 << 8) | lo.to_u16
40     end
41
42     def address_indirect_y : UInt16
43         base = fetch_byte
44
45         lo = read_byte(base.to_u16)
46         hi = read_byte((base &+ 1_u8).to_u16)
47
48         addr = (hi.to_u16 << 8) | lo.to_u16
49         addr &+ @y.to_u16
50     end
51
52     def address_indirect : UInt16
53         pointer = address_absolute
54
55         lo = read_byte(pointer)
56
57         hi_addr = if (pointer & 0xFF) == 0xFF
58             pointer & 0xFF00
59         else
60             pointer + 1
61         end
62
63         hi = read_byte(hi_addr)
64
65         (hi.to_u16 << 8) | lo.to_u16
66     end
67
68     def address_relative : UInt16
69         offset = fetch_byte.to_i8!
70         (@pc &+ offset.to_i16).to_u16!
71     end
72 end
73 end
74 end

```

Wrap up

With all addressing modes implemented, adding new instructions becomes mechanical. LDA Zero Page,X? Call `address_zero_page_x`. STA Absolute,Y? Call `address_absolute_y`. The heavy lifting is done.

In the next chapter we'll implement all the Load and Store operations using these modes. After that we'll tackle more complex instructions like JMP, and once we have enough, we'll move on to the cartridge reader and then the PPU so we can actually see something on screen.

OceanofPDF.com

Chapter 3.4: Completing Load and Store

In the previous chapters we implemented LDA immediate, STA zero page, STA absolute, and all the addressing modes. Now it's time to put it all together: every Load and Store variant that the 6502 supports.

That's 28 opcodes in total. Sounds like a lot, but once you have the pattern down, it's just mechanical work. We'll walk through the pattern with LDA, then apply it to the rest.

OceanofPDF.com

Completing LDA

Our `lda.cr` only has immediate mode. We need 7 more: Zero Page, Zero Page X, Absolute, Absolute X, Absolute Y, Indirect X, and Indirect Y.

But first, let's look at what we already have:

```
1  # src/nes/cpu/instructions/lda.cr
2
3  def lda(value)
4    @a = value
5
6    set_z_flag(@a)
7    set_n_flag(@a)
8  end
9
10 def op_lda_immediate
11   value = fetch_byte
12
13   lda(value)
14 end
```

Two methods here. `lda(value)` is the core logic: take a value, put it in A, update the flags. `op_lda_immediate` is the immediate mode variant: grab the value with `fetch_byte` and hand it to `lda`.

This separation is key. `lda(value)` doesn't know or care where the value came from. Could be a direct fetch, Zero Page, an absolute address, an indirect pointer... doesn't matter. It always does the same thing: load into A and update flags.

So adding a new mode is always the same:

1. Resolve the address (using the addressing mode)
2. Read the value from that address
3. Call `lda(value)`

Let's see it with Zero Page:

```
1 def op_lda_zero_page
2     address = address_zero_page
3     value = read_byte(address)
4
5     lda(value)
6 end
```

Three lines. `address_zero_page` (which we already implemented in chapter 3.3) gives us the address. `read_byte` reads the value at that address. `lda` does the rest.

What about Absolute X?

```
1 def op_lda_absolute_x
2     address = address_absolute_x
3     value = read_byte(address)
4
5     lda(value)
6 end
```

Same pattern, different addressing mode. Indirect Y?

```
1 def op_lda_indirect_y
2     address = address_indirect_y
3     value = read_byte(address)
4
5     lda(value)
6 end
```

See the pattern? With the addressing modes already implemented, adding variants is literally just swapping one word. Here's the complete file:

```
1 # src/nes/cpu/instructions/lda.cr
2
3 module NES
4     class CPU
5         module Instructions
6             module LDA
```

```
7      def lda(value)
8          @a = value
9
10         set_z_flag(@a)
11         set_n_flag(@a)
12     end
13
14     def op_lda_immediate
15         value = fetch_byte
16
17         lda(value)
18     end
19
20     def op_lda_zero_page
21         address = address_zero_page
22         value = read_byte(address)
23
24         lda(value)
25     end
26
27     def op_lda_zero_page_x
28         address = address_zero_page_x
29         value = read_byte(address)
30
31         lda(value)
32     end
33
34     def op_lda_absolute
35         address = address_absolute
36         value = read_byte(address)
37
38         lda(value)
39     end
40
41     def op_lda_absolute_x
42         address = address_absolute_x
43         value = read_byte(address)
44
45         lda(value)
46     end
47
48     def op_lda_absolute_y
49         address = address_absolute_y
50         value = read_byte(address)
51
52         lda(value)
```

```
53         end
54
55     def op_lda_indirect_x
56         address = address_indirect_x
57         value = read_byte(address)
58
59         lda(value)
60     end
61
62     def op_lda_indirect_y
63         address = address_indirect_y
64         value = read_byte(address)
65
66         lda(value)
67     end
68 end
69 end
70 end
71 end
```

8 variants of LDA, and the core method that does the real work. Each variant only differs in how it obtains the address.

OceanofPDF.com

LDX and LDY

LDX (Load X) and LDY (Load Y) are the exact same idea as LDA, just loading into X and Y instead.

LDX:

```
1  # src/nes/cpu/instructions/ldx.cr
2
3  module NES
4    class CPU
5      module Instructions
6        module LDX
7          def ldx(value)
8            @x = value
9
10           set_z_flag(@x)
11           set_n_flag(@x)
12         end
13
14         def op_ldx_immediate
15           value = fetch_byte
16
17           ldx(value)
18         end
19
20         def op_ldx_zero_page
21           address = address_zero_page
22           value = read_byte(address)
23
24           ldx(value)
25         end
26
27         # ... and so on for each mode
28       end
29     end
30   end
31 end
```

You swap @a for @x, lda for ldx, and that's it. Same pattern.

One thing worth noting: **LDX doesn't support the same modes as LDA.** LDA has 8 modes, LDX has 5, LDY has 5. This isn't arbitrary: X and Y were designed for indexing, so they don't need the indirect modes.

Instruction	Available modes
LDA	Immediate, Zero Page, Zero Page X, Absolute, Absolute X, Absolute Y, Indirect X, Indirect Y
LDX	Immediate, Zero Page, Zero Page Y, Absolute, Absolute Y
LDY	Immediate, Zero Page, Zero Page X, Absolute, Absolute X

Notice something interesting: **LDX uses Zero Page,Y and Absolute,Y** (not X). And **LDY uses Zero Page,X and Absolute,X** (not Y). Makes sense when you think about it: if you're loading X, it doesn't have the value you want yet, so you use Y as the index. And vice versa.

Feel free to copy the full LDX and LDY code from the repo. Same pattern as LDA: a core method (`ldx/ldy`) that sets the register and flags, plus one `op_` method per addressing mode.

OceanofPDF.com

The Store instructions

Store instructions are the opposite of Load: take the value from a register and write it to memory.

We already have STA Zero Page and STA Absolute from chapter 3.2. The pattern for store is even simpler than load:

1. Resolve the address
2. Write the register's value to that address

Just two steps. And one important difference: **Store instructions don't affect any flags**. They just write the value and move on.

Here's the complete STA:

```
1  # src/nas/cpu/instructions/sta.cr
2
3  module NES
4    class CPU
5      module Instructions
6        module STA
7          # STA doesn't affect flags
8
9          def op_sta_zero_page
10             address = address_zero_page
11
12             write_byte(address, @a)
13           end
14
15           def op_sta_zero_page_x
16             address = address_zero_page_x
17
18             write_byte(address, @a)
19           end
20
21           def op_sta_absolute
22             address = address_absolute
23
24             write_byte(address, @a)
25           end
```

```
26
27     def op_sta_absolute_x
28         address = address_absolute_x
29
30         write_byte(address, @a)
31     end
32
33     def op_sta_absolute_y
34         address = address_absolute_y
35
36         write_byte(address, @a)
37     end
38
39     def op_sta_indirect_x
40         address = address_indirect_x
41
42         write_byte(address, @a)
43     end
44
45     def op_sta_indirect_y
46         address = address_indirect_y
47
48         write_byte(address, @a)
49     end
50 end
51 end
52 end
53 end
```

Notice there's no core method for STA. It doesn't need one: there's no shared logic beyond `write_byte`. No flags to set, no value to transform. Just resolve the address and write.

STA has 7 modes, same as LDA minus immediate (you can't "store" a literal value, store it where?).

STX and STY

STX and STY are more limited than STA. They only have 3 modes each:

```
1  # src/nes/cpu/instructions/stx.cr
2
3  module NES
4    class CPU
5      module Instructions
6        module STX
7          # STX doesn't affect flags
8
9          def op_stx_zero_page
10             address = address_zero_page
11
12             write_byte(address, @x)
13           end
14
15           def op_stx_zero_page_y
16             address = address_zero_page_y
17
18             write_byte(address, @x)
19           end
20
21           def op_stx_absolute
22             address = address_absolute
23
24             write_byte(address, @x)
25           end
26         end
27       end
28     end
29 end
```

STY is identical but uses @y and Zero Page,X instead of Zero Page,Y. Same logic as the loads: if you're storing X, you use Y as the index.

Write STY yourself. It's 3 methods, takes 30 seconds.

Opcode table

Here are all the opcodes we've just implemented. These go in `op_codes.cr`:

Instruction	Mode	Opcode	Cycles
LDA	Immediate	0xA9	2
	Zero Page	0xA5	3
	Zero Page,X	0xB5	4
	Absolute	0xAD	4
	Absolute,X	0xBD	4
	Absolute,Y	0xB9	4
	Indirect,X	0xA1	6
	Indirect,Y	0xB1	5
LDX	Immediate	0xA2	2
	Zero Page	0xA6	3
	Zero Page,Y	0xB6	4
	Absolute	0xAE	4
	Absolute,Y	0xBE	4
LDY	Immediate	0xA0	2
	Zero Page	0xA4	3
	Zero Page,X	0xB4	4
	Absolute	0xAC	4
	Absolute,X	0xBC	4
STA	Zero Page	0x85	3
	Zero Page,X	0x95	4
	Absolute	0x8D	4
	Absolute,X	0x9D	5
	Absolute,Y	0x99	5
	Indirect,X	0x81	6
	Indirect,Y	0x91	6
STX	Zero Page	0x86	3
	Zero Page,Y	0x96	4
	Absolute	0x8E	4
STY	Zero Page	0x84	3
	Zero Page,X	0x94	4
	Absolute	0x8C	4

28 opcodes. Notice how the simpler modes (Immediate, Zero Page) take fewer cycles than the complex ones (Indirect). Every extra byte read from memory costs another cycle.

OceanofPDF.com

Wiring it all up

Now we need to connect the pieces. First, `instructions.cr` needs to include all the new modules:

```
1  # src/nes/cpu/instructions.cr
2
3  require "./instructions/lda"
4  require "./instructions/ldx"
5  require "./instructions/ldy"
6  require "./instructions/sta"
7  require "./instructions/stx"
8  require "./instructions/sty"
9  require "./instructions/register_increments"
10
11  module NES
12    class CPU
13      module Instructions
14        include LDA
15        include LDX
16        include LDY
17        include STA
18        include STX
19        include STY
20        include RegisterIncrements
21      end
22    end
23  end
```

The opcodes in `op_codes.cr`:

```
1  # src/nes/cpu/op_codes.cr
2
3  module NES
4    class CPU
5      module OpCodes
6        # LDA
7        CODE_LDA_IMMEDIATE    = 0xA9_u8
8        CODE_LDA_ZERO_PAGE    = 0xA5_u8
9        CODE_LDA_ZERO_PAGE_X  = 0xB5_u8
10       CODE_LDA_ABSOLUTE      = 0xAD_u8
```

```

11     CODE_LDA_ABSOLUTE_X = 0xBD_u8
12     CODE_LDA_ABSOLUTE_Y = 0xB9_u8
13     CODE_LDA_INDIRECT_X = 0xA1_u8
14     CODE_LDA_INDIRECT_Y = 0xB1_u8
15
16     # LDX
17     CODE_LDX_IMMEDIATE = 0xA2_u8
18     CODE_LDX_ZERO_PAGE = 0xA6_u8
19     CODE_LDX_ZERO_PAGE_Y = 0xB6_u8
20     CODE_LDX_ABSOLUTE = 0xAE_u8
21     CODE_LDX_ABSOLUTE_Y = 0xBE_u8
22
23     # LDY
24     CODE_LDY_IMMEDIATE = 0xA0_u8
25     CODE_LDY_ZERO_PAGE = 0xA4_u8
26     CODE_LDY_ZERO_PAGE_X = 0xB4_u8
27     CODE_LDY_ABSOLUTE = 0xAC_u8
28     CODE_LDY_ABSOLUTE_X = 0xBC_u8
29
30     # STA
31     CODE_STA_ZERO_PAGE = 0x85_u8
32     CODE_STA_ZERO_PAGE_X = 0x95_u8
33     CODE_STA_ABSOLUTE = 0x8D_u8
34     CODE_STA_ABSOLUTE_X = 0x9D_u8
35     CODE_STA_ABSOLUTE_Y = 0x99_u8
36     CODE_STA_INDIRECT_X = 0x81_u8
37     CODE_STA_INDIRECT_Y = 0x91_u8
38
39     # STX
40     CODE_STX_ZERO_PAGE = 0x86_u8
41     CODE_STX_ZERO_PAGE_Y = 0x96_u8
42     CODE_STX_ABSOLUTE = 0x8E_u8
43
44     # STY
45     CODE_STY_ZERO_PAGE = 0x84_u8
46     CODE_STY_ZERO_PAGE_X = 0x94_u8
47     CODE_STY_ABSOLUTE = 0x8C_u8
48
49     # Register Increments
50     CODE_INX = 0xE8_u8
51
52     # Cycles per Operation Code
53     CYCLES = {
54         CODE_LDA_IMMEDIATE => 2,
55         CODE_LDA_ZERO_PAGE => 3,
56         CODE_LDA_ZERO_PAGE_X => 4,

```

```

57         CODE_LDA_ABSOLUTE      => 4,
58         CODE_LDA_ABSOLUTE_X    => 4,
59         CODE_LDA_ABSOLUTE_Y    => 4,
60         CODE_LDA_INDIRECT_X    => 6,
61         CODE_LDA_INDIRECT_Y    => 5,
62         # ... and so on for the rest
63     }
64 end
65 end
66 end

```

The CYCLES hash follows the same pattern for all 28 opcodes. Just copy the values from the table above.

And finally, the case in CPU#step needs a when for each new opcode:

```

1  # src/nes/cpu.cr (step method)
2
3  def step
4      opcode = fetch_byte
5
6      case opcode
7          # LDA
8          when CODE_LDA_IMMEDIATE then op_lda_immediate
9          when CODE_LDA_ZERO_PAGE then op_lda_zero_page
10         when CODE_LDA_ZERO_PAGE_X then op_lda_zero_page_x
11         when CODE_LDA_ABSOLUTE then op_lda_absolute
12         when CODE_LDA_ABSOLUTE_X then op_lda_absolute_x
13         when CODE_LDA_ABSOLUTE_Y then op_lda_absolute_y
14         when CODE_LDA_INDIRECT_X then op_lda_indirect_x
15         when CODE_LDA_INDIRECT_Y then op_lda_indirect_y
16         # LDX
17         when CODE_LDX_IMMEDIATE then op_ldx_immediate
18         when CODE_LDX_ZERO_PAGE then op_ldx_zero_page
19         when CODE_LDX_ZERO_PAGE_Y then op_ldx_zero_page_y
20         when CODE_LDX_ABSOLUTE then op_ldx_absolute
21         when CODE_LDX_ABSOLUTE_Y then op_ldx_absolute_y
22         # LDY
23         when CODE_LDY_IMMEDIATE then op_ldy_immediate
24         when CODE_LDY_ZERO_PAGE then op_ldy_zero_page
25         when CODE_LDY_ZERO_PAGE_X then op_ldy_zero_page_x
26         when CODE_LDY_ABSOLUTE then op_ldy_absolute
27         when CODE_LDY_ABSOLUTE_X then op_ldy_absolute_x
28         # STA

```

```
29  when CODE_STA_ZERO_PAGE      then op_sta_zero_page
30  when CODE_STA_ZERO_PAGE_X    then op_sta_zero_page_x
31  when CODE_STA_ABSOLUTE       then op_sta_absolute
32  when CODE_STA_ABSOLUTE_X     then op_sta_absolute_x
33  when CODE_STA_ABSOLUTE_Y     then op_sta_absolute_y
34  when CODE_STA_INDIRECT_X     then op_sta_indirect_x
35  when CODE_STA_INDIRECT_Y     then op_sta_indirect_y
36  # STX
37  when CODE_STX_ZERO_PAGE      then op_stx_zero_page
38  when CODE_STX_ZERO_PAGE_Y    then op_stx_zero_page_y
39  when CODE_STX_ABSOLUTE       then op_stx_absolute
40  # STY
41  when CODE_STY_ZERO_PAGE      then op_sty_zero_page
42  when CODE_STY_ZERO_PAGE_X    then op_sty_zero_page_x
43  when CODE_STY_ABSOLUTE       then op_sty_absolute
44  # Register Increments
45  when CODE_INX                then op_inx
46  end
47
48  CYCLES[opcode]
49  end
```

Yeah, it's long. But look at it: purely mechanical. Each line connects an opcode to its method. No logic, no decisions. Just wiring.

OceanofPDF.com

Tests

For testing, same strategy as chapter 3.1: one file per instruction family.

Here's an LDA Zero Page test to show you the pattern:

```
1 # bin/cpu/test_lda.cr (excerpt)
2
3 # Test: LDA Zero Page
4 puts "\n 3. Zero Page"
5 program = Bytes[
6   0xA5, # LDA zero page
7   0x10 # address $10
8 ]
9
10 emu = NES::Emulator.new
11 emu.load_program(program)
12 emu.bus.cpu_write(0x10_u16, 0x42_u8)
13
14 puts "      LDA $10 (MEM[$10] = $42)"
15 emu.step
16 emu.debug
17
18 passed = emu.cpu.a == 0x42
19 puts passed ? "      ✓ " : "      ✗ "
20 all_passed &= passed
```

The key line is `emu.bus.cpu_write(0x10_u16, 0x42_u8)`: before executing, we plant the value 42 at address \$10. That way when LDA reads from \$10, it finds 42 waiting there.

For indexed modes, you also need to load the index register first. For example, to test LDA Zero Page,X:

```
1 # Load X = $05 first, then LDA zero page,X from $10
2 # Should read from $10 + $05 = $15
3 program = Bytes[
4   0xA2, # LDX immediate
5   0x05, # X = $05
6   0xB5, # LDA zero page,X
7   0x10 # base address $10
```



```
8 ]
9
10 emu = NES::Emulator.new
11 emu.load_program(program)
12 emu.bus.cpu_write(0x15_u16, 0x99_u8)
13
14 emu.step # LDX #$05
15 emu.step # LDA $10,X -> reads from $15
16
17 passed = emu.cpu.a == 0x99
```

For stores, you verify that the value arrived at the correct address:

```
1 # Test: STA writes to memory and doesn't modify flags
2 stored_value = emu.bus.cpu_read(address)
3 passed = stored_value == expected_value
4
5 # Verify that flags didn't change
6 flags_unchanged = emu.cpu.status == status_before
```

I'm not going to show all 47 tests here (check the repo for the full set). But the general idea is that each instruction needs:

- A basic test (did the correct value arrive at the correct place?)
- A Z flag test (load with value \$00)
- An N flag test (load with value >= \$80)
- A test for each addressing mode

For stores, also verify that the flags **don't change** after the store.

Create the test files and add them to the runner:

```
1 # bin/test_cpu.cr
2
3 require "../src/nes"
4
5 puts "#" * 60
6 puts "\n"
7 require "../cpu/test_lda"
8
9 puts "#" * 60
```

```
10 puts "\n"
11 require "./cpu/test_ldx"
12
13 puts "#" * 60
14 puts "\n"
15 require "./cpu/test_ldy"
16
17 puts "#" * 60
18 puts "\n"
19 require "./cpu/test_sta"
20
21 puts "#" * 60
22 puts "\n"
23 require "./cpu/test_stx"
24
25 puts "#" * 60
26 puts "\n"
27 require "./cpu/test_sty"
28
29 puts "#" * 60
30 puts "\n"
31 require "./cpu/test_inx"
```

Run `crystal run bin/test_cpu.cr` and make sure everything passes.

[OceanofPDF.com](https://oceanofpdf.com)

Updated structure

```
1 nes-emulator/
2 |─ bin/
3 |   |─ test_cpu.cr
4 |   |─ cpu/
5 |       |─ test_lda.cr
6 |       |─ test_ldx.cr
7 |       |─ test_ldy.cr
8 |       |─ test_sta.cr
9 |       |─ test_stx.cr
10 |      |─ test_sty.cr
11 |      |─ test_inx.cr
12 |─ src/
13 |   |─ nes.cr
14 |   |─ nes/
15 |       |─ bus.cr
16 |       |─ cpu.cr
17 |       |─ emulator.cr
18 |       |─ bus/
19 |           |─ addresses.cr
20 |           |─ cpu/
21 |               |─ addressing_modes.cr
22 |               |─ flags.cr
23 |               |─ instructions.cr
24 |               |─ op_codes.cr
25 |               |─ instructions/
26 |                   |─ lda.cr
27 |                   |─ ldx.cr
28 |                   |─ ldy.cr
29 |                   |─ sta.cr
30 |                   |─ stx.cr
31 |                   |─ sty.cr
32 |                   |─ register_increments.cr
```

If something isn't working, the full code is in the [repo](#).

[OceanofPDF.com](https://oceanofpdf.com)

Wrap up

We went from 4 opcodes to 32. And honestly, 90% of the work was copy-paste with a name change. That's not a bad thing, it's the design doing its job. The addressing modes handle the heavy lifting; the instructions just combine them.

Here are the patterns to remember:

Load (LDA, LDX, LDY): Has a core method that sets the register and updates the Z and N flags. Each addressing mode resolves the address, reads the value, and calls the core.

Store (STA, STX, STY): Resolves the address and writes the register's value. Doesn't affect flags.

With the Load and Store families complete, our emulator can move data between registers and any memory address, using every addressing mode the 6502 supports. It still can't make decisions or do math, but it's got hands now, it can grab things and put them down.

In the next chapter we'll give it a brain: jumps, comparisons, branches, and arithmetic.

OceanofPDF.com

Chapter 3.5: Complex Instructions

We left the emulator with 32 opcodes, all Load and Store. The CPU can move data around, but it can't make decisions or do math. It's like having a calculator that can only copy numbers from one place to another.

In this chapter we're giving it a brain. We'll implement one representative instruction from each new family: branches, jumps, comparisons, shifts, and arithmetic. That's 17 opcodes that cover the patterns for ~70 more. Once you get these, the rest are just variations on the same theme.

OceanofPDF.com

New flag helpers

Before jumping into new instructions, we need to expand our flags module. So far we have `set_z_flag` and `set_n_flag`. The new instructions need three more helpers:

```
1  # src/nes/cpu/flags.cr
2
3  def set_c_flag(on : Bool)
4    set_flag(FLAG_C, on)
5  end
6
7  def set_v_flag(on : Bool)
8    set_flag(FLAG_V, on)
9  end
10
11 def flag_set?(mask : UInt8) : Bool
12   (@status & mask) != 0
13 end
```

`set_c_flag` and `set_v_flag` are convenience helpers for setting the Carry and Overflow flags. `set_c_flag(sum > 0xFF)` reads a lot better than `set_flag(FLAG_C, sum > 0xFF)`.

Notice that unlike `set_z_flag` and `set_n_flag`, these take a `Bool` directly rather than a value they derive the condition from. Why? Z and N are always calculated the same way (Z = is it zero?, N = is bit 7 set?), but C and V mean different things depending on the instruction. C might mean “there was carry in an addition”, “there was no borrow in a subtraction”, or “the bit that fell off a shift”. There’s no universal formula, so each instruction works out the condition itself and passes it in pre-computed.

`flag_set?` is the opposite: instead of setting a flag, it reads one. Branch instructions need it to decide whether to jump or not. `flag_set?(FLAG_Z)` returns true if the Zero flag is set.

Branches, the 6502's if statements

If you've ever written assembly, you know there's no `if`. The closest thing is branches: "if a condition is met, jump somewhere else in the program".

The 6502 has 8 branch instructions, and they all work exactly the same way. The only thing that changes is which flag they check:

Branch	Condition	Plain English
BEQ	Z == 1	"the result was zero" (branch if e qual)
BNE	Z == 0	"the result was NOT zero" (branch if n ot equal)
BCS	C == 1	"there was carry" (branch if carry set)
BCC	C == 0	"there was no carry" (branch if carry clear)
BMI	N == 1	"the result was negative" (branch if m inus)
BPL	N == 0	"the result was positive" (branch if p lus)
BVS	V == 1	"there was signed overflow" (branch if overflow set)
BVC	V == 0	"there was no signed overflow" (branch if overflow clear)

They come in complementary pairs: BEQ/BNE, BCS/BCC, BMI/BPL, BVS/BVC. One checks if the flag is set, the other checks if it's clear.

They all use **relative addressing**, where the operand isn't an absolute address but an offset from -128 to +127 relative to where you are. "Skip forward 5 bytes" or "go back 10 bytes". We already implemented this as `address_relative` in chapter 3.3.

The code is almost trivial. First, a shared helper:

```
1 # src/nes/cpu/instructions/branches.cr
2
3 def branch_if(condition : Bool)
4   target = address_relative
5   @pc = target if condition
6 end
```

`address_relative` gets called **every time**, whether the branch is taken or not. If the condition is true, we overwrite PC with the target. If not, PC already points to the next instruction, so we just keep going.

Each branch is a one-liner:

```
1  # src/nes/cpu/instructions/branches.cr
2
3  # Jump if the result was zero
4  def op_beq
5    branch_if(flag_set?(FLAG_Z))
6  end
7
8  # Jump if the result was NOT zero
9  def op_bne
10   branch_if(!flag_set?(FLAG_Z))
11 end
12
13 # Jump if there was carry (unsigned overflow / no borrow)
14 def op_bcs
15   branch_if(flag_set?(FLAG_C))
16 end
17
18 # Jump if there was no carry (no unsigned overflow / there was
borrow)
19 def op_bcc
20   branch_if(!flag_set?(FLAG_C))
21 end
```

BMI, BPL, BVS, BVC follow the exact same pattern with `FLAG_N` and `FLAG_V`. Write those yourself.

Branches on their own aren't very useful. How does the Z flag get set so BEQ knows to jump? You need a previous instruction to set it. That's where comparisons come in, but first let's look at jumps.

Jumps, the 6502's goto

Jumps are simpler than branches: go to this address, no questions asked.

```
1 # src/nes/cpu/instructions/jumps.cr
2
3 # JMP is the 6502's "go-to"
4 def op_jump_absolute
5   @pc = address_absolute
6 end
7
8 def op_jump_indirect
9   @pc = address_indirect
10 end
```

JMP absolute reads a 16-bit address and sets PC to it. JMP indirect does the same but through a pointer: it reads the address where the real destination is stored. All the complexity (including the infamous 6502 page boundary bug) lives in `address_indirect`, which we already implemented.

Subroutines: JSR and RTS

JMP is a one-way trip. JSR (Jump to Subroutine) and RTS (Return from Subroutine) are like calling a function and then returning: they send you somewhere else but remember where you came from.

To “remember” where to come back to, they need to store the return address somewhere. That somewhere is the **stack**.

The Stack

The 6502's stack lives in **page 1** of memory: addresses \$0100 to \$01FF. Always there, fixed, hardcoded in the hardware. (Page 0, \$0000-\$00FF, is Zero Page, which we already know.)

The Stack Pointer (SP) is an 8-bit register that only stores the low byte of the address; the high byte is always \$01. To get the full address, you OR it with the base:

```
1 SP = $FD
2 $0100 | $FD = $01FD <= actual address in memory
```

The stack grows **downward**. When you push something, SP goes down. When you pull, SP goes up. SP always points to the next **free** position, not the last occupied one.



Figure 2. Stack Diagram

Let's create a Stack module with some constants and 4 helpers:

```
1 # src/nes/cpu/stack.cr
2
3 module NES
4   class CPU
5     module Stack
```

```

6      STACK_BASE = 0x0100_u16 # The stack lives at $0100-$01FF
(Page 1)
7      STACK_INITIAL = 0xFD_u8 # The 6502 decrements 3 times
during reset, so SP starts at $FD. You'll see where we use this
shortly.
8
9      def stack_push_byte(value : UInt8)
10         write_byte(STACK_BASE | @sp.to_u16, value)
11
12         @sp &-= 1
13     end
14
15     def stack_pull_byte : UInt8
16         @sp &+= 1
17
18         read_byte(STACK_BASE | @sp.to_u16)
19     end

```

stack_push_byte writes the value at the current stack address and decrements SP. stack_pull_byte increments SP first, then reads. Notice that push writes *before* decrementing, and pull increments *before* reading. They're mirror images of each other.

For 16-bit addresses (a “word”, 2 bytes), we need versions that push and pull two bytes at a time:

```

1  # src/nes/cpu/stack.cr
2
3      def stack_push_word(value : UInt16)
4         stack_push_byte((value >> 8).to_u8)    # high byte first
5         stack_push_byte((value & 0xFF).to_u8)  # low byte second
6     end
7
8      def stack_pull_word : UInt16
9         lo = stack_pull_byte
10        hi = stack_pull_byte
11
12        (hi.to_u16 << 8) | lo.to_u16 # Combine both bytes into a
word
13    end
14  end
15 end
16 end

```

Push saves the high byte first, then the low byte (the stack grows downward, so the low byte ends up at the lower address). Pull does the reverse: grabs the low byte first, then the high byte, and combines them.

Don't forget to include the module in the CPU:

```
1 # src/nes/cpu.cr
2
3 require "../cpu/stack"
4
5 module NES
6   class CPU
7     include Stack
8     # ...
9   end
10 end
```

JSR and RTS

With the stack ready, JSR and RTS are straightforward:

```
1 # src/nes/cpu/instructions/jumps.cr
2
3 # JSR: jump to a subroutine. Saves the return address on the
stack
4 def op_jsr
5   target = address_absolute
6   stack_push_word(@pc &- 1)
7
8   @pc = target
9 end
10
11 # RTS: return from a subroutine. Pulls the return address from
the stack
12 def op_rts
13   @pc = stack_pull_word &+ 1
14 end
```

JSR reads the destination address, pushes PC - 1 onto the stack, and jumps. RTS pulls from the stack and adds 1.

Why PC - 1 and then + 1? Just a 6502 quirk. JSR saves the address of the **last byte** of the JSR instruction (not the next instruction). RTS compensates by adding 1. Weird, but consistent, they cancel each other out.

The full flow:

1	\$0200: JSR \$0400	; pushes \$0202 (last byte of JSR), jumps to \$0400
2	\$0400: ...	; executes the subroutine
3	\$0410: RTS	; pulls \$0202, +1 = \$0203
4	\$0203: ...	; continues executing after the JSR

OceanofPDF.com

Comparisons, the invisible subtraction

Comparisons are subtractions where you throw away the result. All you care about is the flags.

```
1 # src/nes/cpu/instructions/comparisons.cr
2
3 def compare(register : UInt8, operand : UInt8)
4   result = register &- operand
5
6   set_c_flag(register >= operand)
7   set_z_flag(result)
8   set_n_flag(result)
9 end
```

It computes `register - operand` and extracts three flags from the result:

- C - was the register greater than or equal to the operand? $\$50 - \$30$ has no borrow, so C=1. $\$30 - \$50$ borrows, so C=0.
- Z - are they equal? If the subtraction gives zero, Z=1.
- N - bit 7 of the result. Useful for signed comparisons.

The subtraction result is discarded. The register **does not change**. Only the flags change.

CMP, CPX, and CPY are all the same logic, just targeting different registers:

```
1 # src/nes/cpu/instructions/comparisons.cr
2
3 def op_cmp_immediate
4   value = fetch_byte
5
6   compare(@a, value)
7 end
8
9 def op_cpx_immediate
10  value = fetch_byte
11
12  compare(@x, value)
```

```
13 end
14
15 def op_cpy_immediate
16   value = fetch_byte
17
18   compare(@y, value)
19 end
```

Comparisons + Branches = control flow

This is where branches and comparisons join forces. Here's the classic 6502 loop pattern:

```
1      LDY #$00      ; X = 0
2 loop: INX          ; X++
3      CPY #$0A      ; X == 10?
4      BNE loop      ; if not, repeat
```

CPY compares X with \$0A. When they're equal, the subtraction produces zero, so Z=1. BNE (branch if **not** equal) sees Z=1 and *doesn't* jump. The loop ends.

Another example: an if that skips code:

```
1      LDA score
2      CMP #$64      ; score >= 100?
3      BCC skip      ; if score < 100, skip ahead
4      ; ... code for score >= 100 ...
5 skip: ; ... continues here ...
```

CMP sets C=1 if A >= the operand. BCC (branch if carry **clear**) jumps if C=0, meaning the score was less than 100.

No if, no for, no while. Everything is built from compare + branch. Primitive, but enough for any program.

Shifts: multiply by two

ASL (Arithmetic Shift Left) moves all bits one position to the left:

```
1 Before:  [7] [6] [5] [4] [3] [2] [1] [0]
2 After:   [6] [5] [4] [3] [2] [1] [0] [0]
3           |                               |
4 bit 7 went to flag C      a 0 enters
```

The bit that “falls off” the left end goes into the Carry flag. A 0 enters from the right.

```
1 # src/nes/cpu/instructions/shifts.cr
2
3 def op_asl_accumulator
4   set_c_flag(@a.bit(7) == 1) # The bit that falls off goes to flag
C
5   @a = (@a << 1).to_u8
6
7   set_z_flag(@a)
8   set_n_flag(@a)
9 end
```

The `.to_u8` at the end is necessary because `<<` in Crystal returns `Int32`. Without it, the compiler would complain about assigning to `@a`, which is `UInt8`.

In practice, shifting left is the same as multiplying by 2. `$05 << 1 = $0A` ($5 \times 2 = 10$). Games used this constantly for fast multiplication: shift left twice = $\times 4$, three times = $\times 8$.

We only implement the accumulator version (operates on A directly). ASL can also operate on memory with different addressing modes, but the pattern is the same.

Arithmetic: ADC

ADC (Add with Carry) is the 6502's addition instruction. It does:

```
1 A = A + value + Carry
```

Why include the Carry? So you can chain additions larger than 8 bits. If you want to add two 16-bit numbers, you add the low bytes with ADC, then the high bytes with another ADC. The carry from the first addition flows into the second. It's like when you add by hand and "carry the one".

```
1 # src/nes/cpu/instructions/arithmetic.cr
2
3 MAX_8_BIT_UINT = 0xFF # Maximum number that fits in a byte (8
bits): 255
4
5 def adc(value : UInt8)
6   carry_in = flag_set?(FLAG_C) ? 1_u16 : 0_u16
7   sum = @a.to_u16 + value.to_u16 + carry_in
8
9   result = sum.to_u8!
10
11   set_c_flag(sum > MAX_8_BIT_UINT) # Unsigned overflow
12   set_v_flag((~(@a ^ value) & (@a ^ result) & 0x80) != 0) #
Signed overflow
13
14   @a = result
15
16   set_z_flag(@a)
17   set_n_flag(@a)
18 end
```

Let's break it down.

Why add in 16 bits?

Because we need to detect carry. $\$FF + \$01$ in 8 bits gives $\$00$, and the fact that it overflowed is lost. In 16 bits it gives $\$0100$, and $\$0100 > \$FF$

tells us there was carry.

Why `.to_u8!` with the exclamation mark?

`.to_u8` (without `!`) checks that the value fits in 8 bits and throws an error if it doesn't. `.to_u8!` does wrapping: `$100.to_u8!` gives `$00`. The hardware wraps, so we wrap too.

The easy flags

- **C** (Carry) - `sum > 0xFF`. Didn't fit in 8 bits? That's carry.
- **Z** (Zero) - is the truncated result zero?
- **N** (Negative) - is bit 7 of the result set?

The hard flag: **V** (oVerflow)

A byte can represent two things:

- **Unsigned**: 0 to 255
- **Signed**: -128 to +127

The CPU doesn't know which interpretation you're using. So it gives you **two** flags and lets you check whichever one applies:

- **C** - "it overflowed if you're using unsigned"
- **V** - "it overflowed if you're using signed"

Concrete example: `$7F + $01 = $80`

- As unsigned: `127 + 1 = 128`. Fits in 0-255. No problem. **C=0**.
- As signed: `127 + 1 = -128`. That makes no sense. **V=1**.

The CPU sets both flags. Your program decides which one to check.

The formula to detect signed overflow:

```
1 signed_overflow = (~(@a ^ value) & (@a ^ result) & 0x80) != 0
2 set_v_flag(signed_overflow)
```

Looks like black magic, but it's really just three questions chained with AND:

1. $\sim(@a \wedge \text{value}) \Rightarrow$ do A and value have the **same** sign?
2. $(@a \wedge \text{result}) \Rightarrow$ does the result have a **different** sign than A?
3. $\& 0x80 \Rightarrow$ only look at the sign bit (bit 7)

If all three are true, the inputs had the same sign but the result flipped sign. That's overflow. Two positives gave a negative, or two negatives gave a positive.

If the inputs had different signs (one positive, one negative), overflow is **impossible** because the result always stays in range. That's why the first check filters out that case.

The opcode method just reads the value and calls the core:

```
1 # src/nes/cpu/instructions/arithmetic.cr
2
3 def op_adc_immediate
4   value = fetch_byte
5
6   adc(value)
7 end
```

Cleaning up the initialize

Now that we have constants in the Stack and Flags modules, let's clean up the CPU's initialize. Before, it had magic numbers:

```
1 @sp      = 0xFD_u8 # Stack initial value
2 @status = 0x24_u8 # 0010_0100 -> nvUbdIzc
```

Now it uses the constants we defined:

```
1 # src/nes/cpu.cr
2
3 def initialize(bus : Bus)
4   @bus = bus
5   @a   = 0_u8
6   @x   = 0_u8
7   @y   = 0_u8
8   @sp  = STACK_INITIAL
9   @status = STATUS_INITIAL
10  @pc   = 0_u16
11 end
```

STACK_INITIAL lives in the Stack module. STATUS_INITIAL lives in Flags and is defined as:

```
1 STATUS_INITIAL = FLAG_U | FLAG_I # U and I start set when the CPU boots
```

Instead of having to remember that \$24 is 0010_0100 is U + I, we express it directly with the flags we already have. Anyone reading the code can instantly see which flags start set.

Wiring it all up

As always, we need to wire everything together. In `instructions.cr` we add the 5 new modules:

```
1  # src/nes/cpu/instructions.cr
2
3  require "./instructions/jumps"
4  require "./instructions/comparisons"
5  require "./instructions/branches"
6  require "./instructions/shifts"
7  require "./instructions/arithmetic"
8
9  module NES
10    class CPU
11      module Instructions
12        # ... previous includes ...
13        include Jumps
14        include Comparisons
15        include Branches
16        include Shifts
17        include Arithmetic
18      end
19    end
20  end
```

In `op_codes.cr`, the 17 new constants and their cycles:

Instruction	Opcode	Cycles
JMP absolute	0x4C	3
JMP indirect	0x6C	5
JSR	0x20	6
RTS	0x60	6
CMP immediate	0xC9	2
CPX immediate	0xE0	2
CPY immediate	0xC0	2
BEQ	0xF0	2
BNE	0xD0	2
BCS	0xB0	2
BCC	0x90	2
BMI	0x30	2
BPL	0x10	2
BVS	0x70	2
BVC	0x50	2
ASL accumulator	0x0A	2
ADC immediate	0x69	2

And in `cpu.cr`, the 17 new when clauses in the step case. By now you know the drill: one when per opcode, purely mechanical.

OceanofPDF.com

Updated structure

```
1 nes-emulator/
2 |─ bin/
3 |   └─ test_cpu.cr
4 |   └─ cpu/
5 |       └─ test_lda.cr
6 |       └─ ... (previous tests)
7 |       └─ test_jumps.cr
8 |       └─ test_comparisons.cr
9 |       └─ test_branches.cr
10 |      └─ test_shifts.cr
11 |      └─ test_arithmetic.cr
12 └─ src/
13     └─ nes/
14         └─ cpu/
15             └─ stack.cr                <= NEW
16             └─ flags.cr                (+ flag_set?,
set_c_flag, set_v_flag)
17         └─ instructions/
18             └─ jumps.cr                <= NEW
19             └─ comparisons.cr         <= NEW
20             └─ branches.cr           <= NEW
21             └─ shifts.cr             <= NEW
22             └─ arithmetic.cr         <= NEW
```

Wrap up

We went from 32 to 49 opcodes. But the real win isn't the count, it's the capabilities we picked up:

- **Branches** give the CPU the ability to make decisions. They're the 6502's `if`.
- **Jumps** let you go anywhere in the program. `JMP` is a `goto`, `JSR/RTS` are function calls.
- **Comparisons** set the flags that branches read. Together they form control flow.
- **Shifts** let you multiply and divide by powers of 2.
- **ADC** is the 6502's only addition instruction. With it (and `SBC` which follows the same pattern) all arithmetic is done.

Each of these families has more variants we haven't implemented yet. `CMP` has 8 addressing modes, not just immediate. Same for `ADC`. Branches only have one mode (relative), so those are already done. In the next chapter we'll list all the remaining variants so you can fill them in.

The important thing is you already have the patterns. Adding `CMP` zero page is the same as adding `LDA` zero page in the previous chapter: swap the addressing mode, and the core method does the rest.

Our CPU can now move data, make decisions, call subroutines, and do math. That's enough to run a real program. We're not far from running an actual game.

Check the [repo](#) for guidance on the instructions you need to implement yourself in this chapter.

Chapter 3.6: Interrupts

So far our CPU only does what the program tells it, instruction by instruction, in order. But on a real NES, stuff happens outside the CPU: the PPU finishes drawing a frame, a timer fires, the programmer wants to pause execution. To handle all of that, there's a mechanism called **interrupts**.

An interrupt is exactly what it sounds like: something interrupts what the CPU is doing, sends it off to execute some other code, and when it's done the CPU goes back to what it was doing as if nothing happened.

OceanofPDF.com

Interrupt vectors

Where does the CPU jump to when an interrupt fires? It can't be a fixed address, because every game puts its code in different places. The 6502's solution is simple: the game writes the target address at a known memory location. The CPU looks it up, reads the address, and jumps.

Those “known locations” are called **interrupt vectors**. They're 3 pairs of bytes at the very end of the address space:

Vector	Address	When it's used
NMI	\$FFFA-\$FFFB	The PPU finished drawing a frame (Non-Maskable Interrupt)
RESET	\$FFFC-\$FFFD	The console powers on or gets reset
IRQ/BRK	\$FFFE-\$FFFF	Software interrupt (BRK) or hardware interrupt (IRQ)

Each vector is 2 bytes (a 16-bit address, little-endian). They're hardcoded in the 6502 hardware, always at those addresses. You can't change them.

On the NES, addresses \$FFFA-\$FFFF fall within the cartridge range (PRG-ROM). So it's the game that decides where each handler lives. For example, a cartridge might have:

1	\$FFFA: \$00 \$80	=> NMI handler at \$8000
2	\$FFFC: \$00 \$C0	=> RESET: start at \$C000
3	\$FFFE: \$50 \$80	=> IRQ/BRK handler at \$8050

When you turn on the console, the CPU reads \$FFFC-\$FFFD, assembles the address, and sets the PC there. That's the first instruction the game ever executes. Right now we set the PC manually with `load_program`, but once we implement the cartridge reader in chapter 4 we'll use the RESET vector for real.

We'll use the NMI vector when we implement the PPU. Every time the PPU finishes drawing a frame (an event called VBlank), it fires an NMI. The game

puts all its state-update code in the NMI handler: read input, move sprites, update scroll. It's the heartbeat of the game loop.

The one we care about right now is IRQ/BRK.

OceanofPDF.com

The Interrupts module

The vectors are hardware constants, so we define them in their own module:

```
1 # src/nes/cpu/interrupts.cr
2
3 module NES
4   class CPU
5     module Interrupts
6       NMI_VECTOR = 0xFFFA_u16 # Non-Maskable Interrupt (PPU
VBlank)
7       RESET_VECTOR = 0xFFFFC_u16 # Power-on / Reset
8       IRQ_VECTOR = 0xFFFFE_u16 # Interrupt Request / BRK
9     end
10  end
11 end
```

Three constants, three fixed addresses. Now our instructions can refer to `IRQ_VECTOR` instead of hardcoding `0xFFFFE` everywhere. When we implement the cartridge we'll read from `RESET_VECTOR` to find out where the game starts, and when we implement the PPU we'll read from `NMI_VECTOR` to jump to the VBlank handler.

Don't forget to wire the module into `cpu.cr`:

```
1 # src/nes/cpu.cr
2
3 require "../cpu/interrupts"
4
5 # ... inside the CPU class:
6 include Interrupts
```

BRK, Break

BRK is the instruction that triggers a software interrupt. The difference from JMP or JSR is that BRK **saves the full CPU state** (PC and flags) so it can be restored later. It's like a JSR that also remembers the flags.

Here's what happens:

1. Push PC+1 to the stack (the return address)
2. Push the status register to the stack (with the B and U bits forced to 1)
3. Set the I flag to disable further interrupts (you don't want to be interrupted while handling an interrupt)
4. Read the address from the IRQ/BRK vector (\$FFFE-\$FFFF) and jump there

```
1  # src/nes/cpu/instructions/jumps.cr
2
3  def op_brk
4    # Save return address
5    stack_push_word(@pc &+ 1)
6
7    # Save flags (with B and U forced to 1)
8    stack_push_byte(@status | FLAG_B | FLAG_U)
9
10   # Disable interrupts
11   set_flag(FLAG_I, true)
12
13   # Jump to IRQ/BRK handler
14   lo = read_byte(IRQ_VECTOR)
15   hi = read_byte(IRQ_VECTOR &+ 1)
16   @pc = (hi.to_u16 << 8) | lo.to_u16
17 end
```

Why PC+1 and not PC-1 like JSR? JSR saves the address of the last byte of its own instruction. BRK saves the address of the next instruction + 1, leaving a spare byte between the BRK and the following instruction. Some programs used that byte as a “signature” to tell the handler which BRK triggered it. Weird, but that's how they designed it.

Why force B and U to 1 when pushing? So the handler can tell the difference between a BRK (software, B=1) and an IRQ (hardware, B=0). When the handler pulls the status from the stack, it checks the B bit: if it's 1, it was a BRK. If it's 0, it was a hardware IRQ.

Notice that the last step reads the interrupt vector using our IRQ_VECTOR constant. It reads two consecutive bytes (\$FFFE and \$FFFF), combines them into a 16-bit address (little-endian, as always on the 6502), and sets the PC there. Without a cartridge plugged in, the bus returns \$00 for those addresses, so BRK jumps to \$0000. Once we have the cartridge, it'll jump wherever the game says.

OceanofPDF.com

RTI, Return from Interrupt

RTI undoes what BRK did. It restores the flags from the stack and then restores the PC:

```
1 # src/nes/cpu/instructions/jumps.cr
2
3 def op_rti
4   # Restore flags (clear B, force U)
5   @status = (stack_pull_byte & ~FLAG_B.to_u8) | FLAG_U
6
7   # Restore the PC
8   @pc = stack_pull_word
9 end
```

Two things to notice:

It doesn't add 1 to the PC (unlike RTS). BRK already pushed PC+1, so the value on the stack already points to the right instruction. RTI doesn't need to compensate for anything.

It clears B and forces U when restoring the status. B isn't really a flag that lives in the register; it only exists when the status gets pushed to the stack. In the actual register it's always 0. U is always 1.

Here's the full picture:

```
1 $0200: BRK           ; pushes $0202 and status, jumps to handler
2 ...
3 ; interrupt handler at whatever address $FFFE-$FFFF points to
4 $8050: ...           ; handler does its thing
5 $8060: RTI           ; restores status and PC, returns to $0202
6 ...
7 $0202: ...           ; CPU picks up right where it left off
```

Wiring it up

First, the opcodes and cycles in `op_codes.cr`:

```
1 # src/nes/cpu/op_codes.cr
2
3 # Jumps (add to existing constants)
4 CODE_BRK = 0x00_u8
5 CODE_RTI = 0x40_u8
6
7 # Cycles (add to existing hash)
8 CODE_BRK => 7,
9 CODE_RTI => 6,
```

Instruction	Opcode	Cycles	Flags
BRK	0x00	7	I = 1
RTI	0x40	6	all (restored from stack)

And the matching when clauses in the case in `cpu.cr`:

```
1 # src/nes/cpu.cr (inside the step case)
2
3 # Jumps
4 when CODE_JMP_ABSOLUTE then op_jmp_absolute
5 when CODE_JMP_INDIRECT then op_jmp_indirect
6 when CODE_JSR           then op_jsr
7 when CODE_RTS           then op_rts
8 when CODE_BRK           then op_brk
9 when CODE_RTI           then op_rti
```

BRK and RTI live in `jumps.cr` because they're conceptually jumps: BRK jumps to the handler, RTI jumps back. Same idea as JSR/RTS, but with the full CPU context saved and restored.

Wrap up

With BRK and RTI, we've got the interrupt mechanism in place. Our CPU can now:

- Jump to a handler when something happens (BRK)
- Return exactly where it was, with flags restored (RTI)

The interrupt vectors will become essential later. The RESET vector (\$FFFC-\$FFFD) will tell us where the game starts once we can read cartridges. The NMI vector (\$FFFA-\$FFFB) will be the heartbeat of the game loop once we implement the PPU.

For now we've gone from 49 to 51 opcodes. In the next chapter we'll knock out the rest.

Check the [repo](#) to copy the tests and/or look at the code for this chapter.

OceanofPDF.com

Chapter 3.7: Completing the CPU, all remaining instructions

In the previous chapters we implemented representative instructions from each family. We learned the patterns: how branches work, how jumps work, comparisons, shifts, arithmetic, and interrupts. Now it's time for the mechanical part: completing the 100 remaining variants.

Heads up: this chapter is long and repetitive on purpose. It's a reference. As soon as you feel like you've got the pattern down and you're getting bored, just copy the missing opcodes from the repo and skip ahead. Seriously. Chapter 4 is where we start reading real cartridges and chapter 5 is where we draw pixels on screen. Don't get stuck here repeating variations of the same thing. If you can implement one instruction from each family, you can implement all of them. You don't need to prove it.

NOTE: In the repo you'll find both the implementation and the tests for each family. If you copy the code, copy the tests too and run them to verify everything works.

With that out of the way, here's everything that's left. First the complete table, then the explanations family by family.

OceanofPDF.com

Complete table

These are the 100 remaining instructions. Each row has the opcode, the addressing mode, the cycles, the flags it affects, and the repo commit where it's implemented.

Instruction	Mode	Opcode	Cycles	Flags	Commit
Register					
increments/decrements					Link
INY	Implied	0xC8	2	Z, N	
DEX	Implied	0xCA	2	Z, N	
DEY	Implied	0x88	2	Z, N	
Register transfers					Link
TAX	Implied	0xAA	2	Z, N	
TAY	Implied	0xA8	2	Z, N	
TXA	Implied	0x8A	2	Z, N	
TYA	Implied	0x98	2	Z, N	
TSX	Implied	0xBA	2	Z, N	
TXS	Implied	0x9A	2	none	
Flag operations					Link
CLC	Implied	0x18	2	C = 0	
SEC	Implied	0x38	2	C = 1	
CLI	Implied	0x58	2	I = 0	
SEI	Implied	0x78	2	I = 1	
CLV	Implied	0xB8	2	V = 0	
CLD	Implied	0xD8	2	D = 0	
SED	Implied	0xF8	2	D = 1	
Stack operations					Link
PHA	Implied	0x48	3	none	
PLA	Implied	0x68	4	Z, N	
PHP	Implied	0x08	3	none	
PLP	Implied	0x28	4	all	
Logical: AND, ORA, EOR					Link
AND	Immediate	0x29	2	Z, N	
AND	Zero Page	0x25	3	Z, N	
AND	Zero Page,X	0x35	4	Z, N	
AND	Absolute	0x2D	4	Z, N	
AND	Absolute,X	0x3D	4	Z, N	
AND	Absolute,Y	0x39	4	Z, N	

Instruction	Mode	Opcode	Cycles	Flags	Commit
AND	Indirect,X	0x21	6	Z, N	
AND	Indirect,Y	0x31	5	Z, N	
ORA	Immediate	0x09	2	Z, N	
ORA	Zero Page	0x05	3	Z, N	
ORA	Zero Page,X	0x15	4	Z, N	
ORA	Absolute	0x0D	4	Z, N	
ORA	Absolute,X	0x1D	4	Z, N	
ORA	Absolute,Y	0x19	4	Z, N	
ORA	Indirect,X	0x01	6	Z, N	
ORA	Indirect,Y	0x11	5	Z, N	
EOR	Immediate	0x49	2	Z, N	
EOR	Zero Page	0x45	3	Z, N	
EOR	Zero Page,X	0x55	4	Z, N	
EOR	Absolute	0x4D	4	Z, N	
EOR	Absolute,X	0x5D	4	Z, N	
EOR	Absolute,Y	0x59	4	Z, N	
EOR	Indirect,X	0x41	6	Z, N	
EOR	Indirect,Y	0x51	5	Z, N	

**Comparisons
(remaining modes)**

[Link](#)

CMP	Zero Page	0xC5	3	C, Z, N	
CMP	Zero Page,X	0xD5	4	C, Z, N	
CMP	Absolute	0xCD	4	C, Z, N	
CMP	Absolute,X	0xDD	4	C, Z, N	
CMP	Absolute,Y	0xD9	4	C, Z, N	
CMP	Indirect,X	0xC1	6	C, Z, N	
CMP	Indirect,Y	0xD1	5	C, Z, N	

Instruction	Mode	Opcode	Cycles	Flags	Commit
CPX	Zero Page	0xE4	3	C, Z, N	
CPX	Absolute	0xEC	4	C, Z, N	
CPY	Zero Page	0xC4	3	C, Z, N	
CPY	Absolute	0xCC	4	C, Z, N	
Arithmetic: remaining					
ADC + SBC					Link
ADC	Zero Page	0x65	3	C, Z, N, V	
ADC	Zero Page,X	0x75	4	C, Z, N, V	
ADC	Absolute	0x6D	4	C, Z, N, V	
ADC	Absolute,X	0x7D	4	C, Z, N, V	
ADC	Absolute,Y	0x79	4	C, Z, N, V	
ADC	Indirect,X	0x61	6	C, Z, N, V	
ADC	Indirect,Y	0x71	5	C, Z, N, V	
SBC	Immediate	0xE9	2	C, Z, N, V	
SBC	Zero Page	0xE5	3	C, Z, N, V	
SBC	Zero Page,X	0xF5	4	C, Z, N, V	
SBC	Absolute	0xED	4	C, Z, N, V	
SBC	Absolute,X	0xFD	4	C, Z, N, V	

Instruction	Mode	Opcode	Cycles	Flags	Commit
SBC	Absolute,Y	0xF9	4	C, Z, N, V	
SBC	Indirect,X	0xE1	6	C, Z, N, V	
SBC	Indirect,Y	0xF1	5	C, Z, N, V	
Shifts: ASL, LSR, ROL, ROR					Link
ASL	Zero Page	0x06	5	C, Z, N	
ASL	Zero Page,X	0x16	6	C, Z, N	
ASL	Absolute	0x0E	6	C, Z, N	
ASL	Absolute,X	0x1E	7	C, Z, N	
LSR	Accumulator	0x4A	2	C, Z, N	
LSR	Zero Page	0x46	5	C, Z, N	
LSR	Zero Page,X	0x56	6	C, Z, N	
LSR	Absolute	0x4E	6	C, Z, N	
LSR	Absolute,X	0x5E	7	C, Z, N	
ROL	Accumulator	0x2A	2	C, Z, N	
ROL	Zero Page	0x26	5	C, Z, N	
ROL	Zero Page,X	0x36	6	C, Z, N	
ROL	Absolute	0x2E	6	C, Z, N	

Instruction	Mode	Opcode	Cycles	Flags	Commit
ROL	Absolute,X	0x3E	7	C, Z, N	
ROR	Accumulator	0x6A	2	C, Z, N	
ROR	Zero Page	0x66	5	C, Z, N	
ROR	Zero Page,X	0x76	6	C, Z, N	
ROR	Absolute	0x6E	6	C, Z, N	
ROR	Absolute,X	0x7E	7	C, Z, N	
Misc: INC, DEC, BIT, NOP					Link
INC	Zero Page	0xE6	5	Z, N	
INC	Zero Page,X	0xF6	6	Z, N	
INC	Absolute	0xEE	6	Z, N	
INC	Absolute,X	0xFE	7	Z, N	
DEC	Zero Page	0xC6	5	Z, N	
DEC	Zero Page,X	0xD6	6	Z, N	
DEC	Absolute	0xCE	6	Z, N	
DEC	Absolute,X	0xDE	7	Z, N	
BIT	Zero Page	0x24	3	Z, N, V	
BIT	Absolute	0x2C	4	Z, N, V	
NOP	Implied	0xEA	2	none	

With those 100 opcodes plus the 51 we already had (49 from chapter 3.5 + BRK and RTI from 3.6), we reach **151 official opcodes**. That's 100% of the 6502 CPU (without illegal/undocumented instructions, which we don't need).

Alright, let's go family by family.

OceanofPDF.com

Register increments and decrements

We already have INX. INY, DEX, and DEY are the same idea, just with a different register and operation:

```
1  # src/nes/cpu/instructions/register_increments.cr
2
3  def op_iny
4    @y &+= 1
5    set_z_flag(@y)
6    set_n_flag(@y)
7  end
8
9  def op_dex
10   @x &-= 1
11   set_z_flag(@x)
12   set_n_flag(@x)
13 end
14
15 def op_dey
16   @y &-= 1
17   set_z_flag(@y)
18   set_n_flag(@y)
19 end
```

No surprises here. &+= and &-= so it wraps around (0x00 - 1 = 0xFF, 0xFF + 1 = 0x00).

Register transfers

Copy the value from one register to another. A new file:

```
1  # src/nes/cpu/instructions/register_transfers.cr
2
3  # Transfer A to X
4  def op_tax
5    @x = @a
6    set_z_flag(@x)
7    set_n_flag(@x)
8  end
9
10 # Transfer A to Y
11 def op_tay
12   @y = @a
13   set_z_flag(@y)
14   set_n_flag(@y)
15 end
16
17 # Transfer X to A
18 def op_txa
19   @a = @x
20   set_z_flag(@a)
21   set_n_flag(@a)
22 end
23
24 # Transfer Y to A
25 def op_tya
26   @a = @y
27   set_z_flag(@a)
28   set_n_flag(@a)
29 end
30
31 # Transfer Stack Pointer to X
32 def op_tsx
33   @x = @sp
34   set_z_flag(@x)
35   set_n_flag(@x)
36 end
```

TXS is the odd one out: it doesn't set any flags. It's the only way to set the Stack Pointer to an arbitrary value.

```
1 # Transfer X to Stack Pointer
2 def op_txs
3   @sp = @x
4 end
```

OceanofPDF.com

Flag operations

The simplest instructions on the entire 6502. Each one just flips a flag on or off:

```
1 # src/nes/cpu/instructions/flag_ops.cr
2
3 def op_clc
4   set_flag(FLAG_C, false)
5 end
6
7 def op_sec
8   set_flag(FLAG_C, true)
9 end
```

CLI, SEI, CLV, CLD, and SED follow the exact same pattern with their respective flags. Write them yourself or copy them from the repo.

When would you use them? CLC before an ADC to make sure there's no stale carry. SEC before an SBC because subtraction needs carry=1 to work correctly. SEI/CLI to enable/disable interrupts. CLD/SED for decimal mode, which the NES doesn't use but the 6502 supports.

There's no instruction to clear N or Z directly. Those flags only get set as a side effect of other instructions.

Stack operations

Push and pull between registers and the stack. They use the `stack_push_byte` and `stack_pull_byte` helpers we already have:

```
1  # src/nes/cpu/instructions/stack_ops.cr
2
3  # Push accumulator to stack
4  def op pha
5    stack_push_byte(@a)
6  end
7
8  # Pull from stack to accumulator
9  def op pla
10   @a = stack_pull_byte
11   set_z_flag(@a)
12   set_n_flag(@a)
13 end
```

PHA doesn't affect flags. PLA does, because it changes A.

PHP and PLP are more interesting. PHP pushes the status register (flags) to the stack, PLP restores it. But they have a hardware quirk:

```
1  # Push status to stack (with B and U bits forced to 1)
2  def op php
3    stack_push_byte(@status | FLAG_B | FLAG_U)
4  end
5
6  # Pull status from stack (ignoring B, forcing U)
7  def op plp
8    value = stack_pull_byte
9    @status = (value & ~FLAG_B.to_u8) | FLAG_U
10 end
```

When PHP pushes the status, it forces the B (Break) and U (Unused) bits to 1. That's just a hardware quirk. When PLP restores, we make sure B goes back to 0 (only BRK sets it) and U stays at 1 (it's always 1).

OceanofPDF.com

Logical: AND, ORA, EOR

Three bitwise operations on the accumulator. Identical structure, just a different operator:

```
1  # src/nes/cpu/instructions/logical.cr
2
3  def and_op(value : UInt8)
4    @a &= value
5    set_z_flag(@a)
6    set_n_flag(@a)
7  end
8
9  def ora(value : UInt8)
10   @a |= value
11   set_z_flag(@a)
12   set_n_flag(@a)
13 end
14
15 def eor(value : UInt8)
16   @a ^= value
17   set_z_flag(@a)
18   set_n_flag(@a)
19 end
```

The name `and_op` is because `and` is a reserved word in Crystal.

Each one has 8 addressing modes, same as LDA. The pattern is the same as always: resolve address, read value, call the core method:

```
1  # AND Immediate
2  def op_and_immediate
3    value = fetch_byte
4    and_op(value)
5  end
6
7  # AND Zero Page
8  def op_and_zero_page
9    address = address_zero_page
10   value = read_byte(address)
11   and_op(value)
```



```
12 end
13
14 # ... and so on for each mode
```

ORA and EOR are identical except they call `ora(value)` and `eor(value)`. That's 24 opcodes total, but they're all mechanical. Implement AND fully, then copy the pattern for ORA and EOR.

OceanofPDF.com

Comparisons, remaining modes

`compare(register, value)` already exists from the previous chapter. We just need to add the missing addressing modes. Same pattern as LDA, nothing new:

```
1  # src/nes/cpu/instructions/comparisons.cr
2
3  # CMP, the 7 remaining modes
4  def op_cmp_zero_page
5    address = address_zero_page
6    value = read_byte(address)
7    compare(@a, value)
8  end
9
10 def op_cmp_absolute
11   address = address_absolute
12   value = read_byte(address)
13   compare(@a, value)
14 end
15
16 # ... and so on for Zero Page,X, Absolute,X, Absolute,Y,
    Indirect,X, Indirect,Y
```

CPX and CPY only support Zero Page and Absolute (on top of the Immediate mode we already have):

```
1  def op_cpx_zero_page
2    address = address_zero_page
3    value = read_byte(address)
4    compare(@x, value)
5  end
6
7  def op_cpx_absolute
8    address = address_absolute
9    value = read_byte(address)
10   compare(@x, value)
11 end
12
13 # CPY is the same but with @y
```

OceanofPDF.com

Arithmetic: remaining ADC + SBC

ADC, remaining modes

`adc(value)` already exists. Adding the other 7 modes is purely mechanical:

```
1 # src/nes/cpu/instructions/arithmetic.cr
2
3 def op_adc_zero_page
4   address = address_zero_page
5   value = read_byte(address)
6   adc(value)
7 end
8
9 def op_adc_absolute
10  address = address_absolute
11  value = read_byte(address)
12  adc(value)
13 end
14
15 # ... and so on for each mode
```

SBC, Subtract with Carry

The counterpart to ADC. It computes $A = A - \text{value} - (1 - \text{Carry})$. The trick is that you can implement it internally as $A + \sim\text{value} + \text{Carry}$ (flip the value's bits and add):

```
1 # src/nes/cpu/instructions/arithmetic.cr
2
3 def sbc(value : UInt8)
4   # Invert the value and reuse ADC logic (one's complement)
5   adc(~value)
6 end
```

Yep, it's one line. `~value` inverts all the bits (one's complement), and the addition with carry does the rest. The flags are set correctly because the ADC formula already handles them.

Then the 8 opcodes follow the usual pattern:

```
1 def op_sbc_immediate
2     value = fetch_byte
3     sbc(value)
4 end
5
6 def op_sbc_zero_page
7     address = address_zero_page
8     value = read_byte(address)
9     sbc(value)
10 end
11
12 # ... and so on for each mode
```

OceanofPDF.com

Full shifts: ASL, LSR, ROL, ROR

We already have ASL accumulator. What's missing is the memory modes for ASL, plus the three sibling families.

ASL on memory

The memory modes are different from accumulator: they read the value, do the shift, and write the result back.

```
1  # src/nes/cpu/instructions/shifts.cr
2
3  def asl_memory(address : UInt16)
4    value = read_byte(address)
5    set_c_flag(value.bit(7) == 1)
6    result = (value << 1).to_u8
7    write_byte(address, result)
8    set_z_flag(result)
9    set_n_flag(result)
10 end
11
12 def op_asl_zero_page
13   address = address_zero_page
14   asl_memory(address)
15 end
16
17 # ... and so on for each mode
```

LSR, Logical Shift Right

The opposite of ASL. Shifts all bits one position to the right. Bit 0 goes to the C flag, and a 0 enters from the left.

```
1 Before:  [7][6][5][4][3][2][1][0]
2 After:   [0][7][6][5][4][3][2][1]
```

3		
4	a 0 enters	bit 0 went to C flag

```

1 def op_lsr_accumulator
2   set_c_flag(@a.bit(0) == 1)
3   @a = (@a >> 1).to_u8
4   set_z_flag(@a)
5   set_n_flag(@a)
6 end

```

N always ends up 0 after an LSR (because bit 7 is always 0 after the shift), but we set N anyway for consistency.

ROL, Rotate Left

Like ASL, but instead of a 0 entering from the right, the current C flag value enters. The bit 7 that falls off becomes the new C.

```

1 C(old) => [bit 0]  and  [bit 7] => C(new)

```

```

1 def op_rol_accumulator
2   old_carry = flag_set?(FLAG_C) ? 1_u8 : 0_u8
3   set_c_flag(@a.bit(7) == 1)
4   @a = ((@a << 1) | old_carry).to_u8
5   set_z_flag(@a)
6   set_n_flag(@a)
7 end

```

ROR, Rotate Right

Like LSR, but instead of a 0 entering from the left, the current C flag value enters. The bit 0 that falls off becomes the new C.

```

1 def op_ror_accumulator
2   old_carry = flag_set?(FLAG_C) ? 0x80_u8 : 0_u8
3   set_c_flag(@a.bit(0) == 1)
4   @a = ((@a >> 1) | old_carry).to_u8
5   set_z_flag(@a)

```

```
6    set_n_flag(@a)
7  end
```

Notice that in ROL the old carry enters as bit 0 (value 1), and in ROR it enters as bit 7 (value 0x80).

All four shift instructions (ASL, LSR, ROL, ROR) have the same modes: accumulator, zero page, zero page X, absolute, absolute X. The memory modes follow the read-modify-write pattern: read, operate, write back.

OceanofPDF.com

Misc: INC, DEC, BIT, NOP

INC and DEC

Increment and decrement, but on memory. Similar to INX/DEX except they operate on a memory address instead of a register:

```
1  # src/nes/cpu/instructions/misc.cr
2
3  def inc(address : UInt16)
4    value = read_byte(address)
5    result = value &+ 1
6    write_byte(address, result)
7    set_z_flag(result)
8    set_n_flag(result)
9  end
10
11 def dec(address : UInt16)
12   value = read_byte(address)
13   result = value &- 1
14   write_byte(address, result)
15   set_z_flag(result)
16   set_n_flag(result)
17 end
```

They don't use or affect the Carry (unlike ADC/SBC). They have 4 modes each: Zero Page, Zero Page X, Absolute, Absolute X.

BIT, Bit Test

ANDs the accumulator with a memory value, but **doesn't store the result**. Think of it like CMP but with AND instead of subtraction. It sets three flags in an unusual way:

```
1  def bit(address : UInt16)
2    value = read_byte(address)
3    set_z_flag(@a & value)           # Z from the AND result
4    set_n_flag(value)                # N from bit 7 of the MEMORY
```

```
value
5   set_v_flag(value.bit(6) == 1)   # V from bit 6 of the MEMORY
value
6   end
```

Notice that N and V come straight from the memory value's bits, not from the AND result. It's the only instruction that sets V this way (ADC/SBC calculate it with the signed overflow formula). It only has 2 modes: Zero Page and Absolute.

NOP

Does absolutely nothing. Literally an empty method:

```
1   def op_nop
2   end
```

Games used it for timing or to reserve space in the code.

OceanofPDF.com

Wiring everything up

For each new family, you need to:

1. Add the opcode constants in `op_codes.cr`
2. Add the cycles in the CYCLES hash
3. Add the when clauses in the case statement in `cpu.cr`
4. If it's a new module, add the `require` and `include` in `instructions.cr`

By this point it's all mechanical. Don't forget to compile every now and then to make sure things still build:

```
1 crystal build bin/test_cpu.cr --no-codegen
```

OceanofPDF.com

Wrap up

With these 100 opcodes plus the 51 we already had, that's **151 official opcodes**. The full 6502 CPU, done. We skipped the illegal/undocumented instructions because no commercial game uses them (or at least none of the ones we care about).

Our CPU now knows how to do everything: move data, make decisions, call subroutines, do arithmetic, operate at the bit level, manage the stack, and respond to interrupts.

But a CPU with nothing to run is pretty boring. In the next chapter we're going to feed it: we'll read an actual .nes file, parse its contents, and load a real program into memory. That's the first step toward running a game.

OceanofPDF.com

Chapter 4: The Cartridge

Our CPU knows how to do everything: move data, add, compare, jump, handle interrupts. But so far we've been hand-feeding it little test programs:

```
1 program = Bytes[0xA9, 0x05, 0x85, 0x10]
2 emu.load_program(program)
```

That's not a game. A NES game comes on a cartridge, and that cartridge is more than a glorified USB stick. There's real hardware in there. In this chapter we're going to understand what's inside a cartridge, how it connects to the NES, and what we need to build so our emulator can read real .nes files.

No code yet. First we build the mental model, then we get our hands on the keyboard.

OceanofPDF.com

What's inside a cartridge?

If you crack open a NES cartridge (don't do it with a good one), you'll find a circuit board with soldered chips. In the simplest case, there are just two:

- **PRG-ROM (Program ROM)** - A read-only memory chip containing the **game code**. The instructions the CPU executes. It's the same kind of thing we've been writing in our tests (0xA9, 0x05 . . .), but we're talking anywhere from 16KB to 512KB of real code.
- **CHR-ROM (Character ROM)** - Another read-only memory chip, but this one holds the **graphics**. Mario, the blocks, the clouds, the Goombas. All stored as 8x8 pixel tiles.



Figure 3. Super Mario Bros Cartridge

That's the bare minimum. Donkey Kong, Mario Bros (the arcade one, not Super Mario), Balloon Fight... small games that fit on two chips.

But as games got more ambitious, cartridges started including extras:

- **Mapper** - An extra logic chip that lets the cartridge hold more memory than the NES can address at once. We'll get into the details shortly.

- **Battery + RAM** - A RAM chip powered by a coin cell battery, for save games. This is how Zelda lets you save your progress.
- **Other goodies** - Some cartridges had scanline counters, extra sound generators, or even accelerometers (the Kirby Tilt 'n' Tumble cartridge for Game Boy had one). The NES wasn't quite that exotic, but the point is that cartridges could pack in pretty much any hardware.



Figure 4. Super Mario Bros 3 Cartridge

OceanofPDF.com

PRG-ROM: the code

The PRG-ROM contains the instructions our CPU executes. Every byte the CPU reads from the cartridge region comes from this chip.

What “cartridge region” am I talking about? Remember the memory map from the first chapters. The CPU sees a flat 64KB address space, and each range is assigned to a different component:

1	\$0000-\$07FF	RAM (2KB, with mirrors up to \$1FFF)
2	\$2000-\$3FFF	PPU registers (future)
3	\$4000-\$401F	APU and I/O (future)
4	\$4020-\$5FFF	Expansion ROM (almost never used)
5	\$6000-\$7FFF	Cartridge SRAM (if it has a battery)
6	\$8000-\$FFFF	PRG-ROM <= THE GAME CODE LIVES HERE

Addresses \$8000 through \$FFFF give us 32KB. When the CPU does a read in that range, it’s not hitting the console’s RAM. It’s reading **directly from the cartridge’s PRG-ROM chip**. Our Bus needs to know how to route those reads to the right place.

And remember the interrupt vectors from chapter 3.6:

1	RESET_VECTOR	= \$FFFC
2	NMI_VECTOR	= \$FFFA
3	IRQ_VECTOR	= \$FFFE

All three live in the cartridge range. When the NES powers on, the CPU reads \$FFFC-\$FFFD to figure out where to start executing. Those bytes come from the PRG-ROM. **Without a cartridge, the CPU has no idea where to start.**

So once we wire the cartridge into the Bus, everything falls into place: the CPU reads the RESET vector from the PRG-ROM and starts executing the actual game.

CHR-ROM: the graphics

The CHR-ROM contains the game's graphics, organized as **tiles** of 8x8 pixels. Every image you see on the NES is built from these tiles: backgrounds, sprites, score numbers, menu text. It's all just combinations of 8x8 tiles.

Here's something important: **the CPU never touches the CHR-ROM**. The CHR-ROM is connected to the PPU's bus, which is completely separate from the CPU's bus:

1 CPU bus (64KB):	PPU bus (16KB):
2 \$0000 RAM	\$0000 CHR-ROM <= the graphics
3 \$2000 PPU registers	\$2000 Nametables (backgrounds)
4 \$8000 PRG-ROM (cartridge)	\$3F00 Color palettes

They're two separate worlds. If the CPU reads address \$0000, it reads its RAM. If the PPU reads address \$0000, it reads the cartridge's CHR-ROM. Same address, different bus, different result.

For us, this means the cartridge needs two interfaces: one for the CPU (PRG-ROM) and one for the PPU (CHR-ROM). For now we'll only use the CPU side, since we don't have a PPU yet. But we'll set up the structure ahead of time.

CHR-ROM vs CHR-RAM

Some cartridges don't have a dedicated graphics chip. Instead, they have an empty **RAM** chip in that slot. The game (running on the CPU) writes graphics into that RAM during execution.

Why? Flexibility. A game with CHR-ROM has fixed graphics, burned in at the factory. A game with CHR-RAM can generate and modify graphics in real time. It's slower (the CPU has to write the tiles), but more versatile.

How do we tell which one a game uses? The `.nes` file header tells us how many CHR-ROM blocks the cartridge has. If it says "0 blocks", the cartridge

uses CHR-RAM. In that case, instead of loading data from the file, we just allocate 8KB of empty RAM and call it a day.

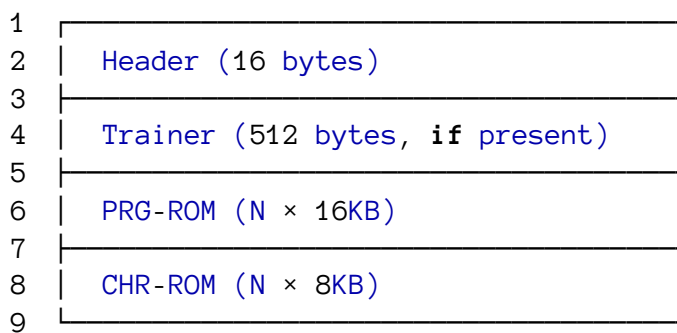
Don't worry about the CHR details yet. We'll dig into all of that when we get to the PPU.

OceanofPDF.com

The iNES format

When you download a NES ROM, it comes as a `.nes` file in iNES format. This format was created by a guy named Marat Fayzullin in 1996 for his emulator (creatively named “iNES”), and it became the de facto standard. Virtually every ROM out there uses it.

It’s simple: a 16-byte header followed by the raw cartridge data.



The header

The first 16 bytes tell you everything you need to know about the cartridge:

-
- | | | |
|---|------------|--|
| 1 | Byte 0-3: | "NES" followed by byte 0x1A |
| 2 | Byte 4: | Number of PRG-ROM blocks (each block = 16KB) |
| 3 | Byte 5: | Number of CHR-ROM blocks (each block = 8KB) |
| 4 | Byte 6: | Flags 6 |
| 5 | Byte 7: | Flags 7 |
| 6 | Byte 8-15: | Padding (generally zeros, we ignore them) |
-

Bytes 0 through 3 are the **magic number**: a signature that identifies the file as a NES ROM. If you open a `.nes` file in a hex editor, the first 4 bytes always read 4E 45 53 1A, which is “NES” in ASCII followed by a control byte. If those bytes don’t match, it’s not a valid iNES file.

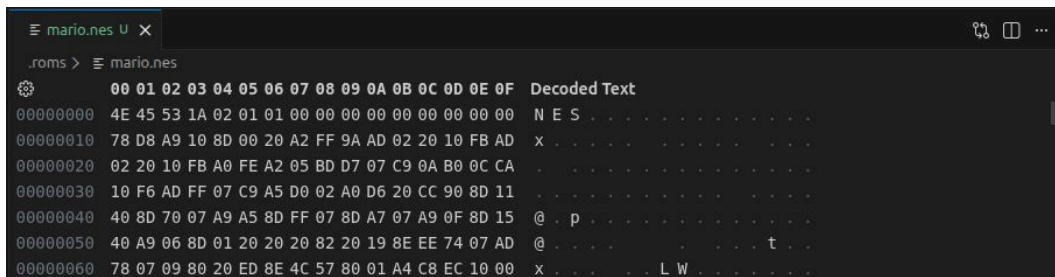


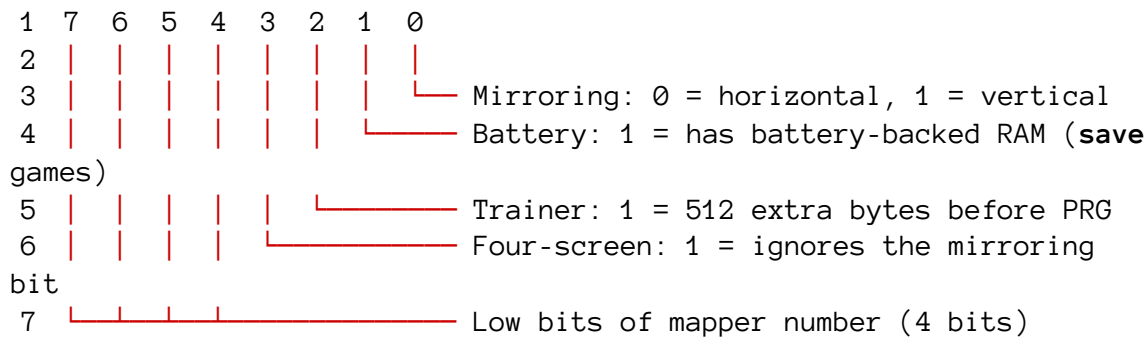
Figure 5. Super Mario Bros ROM Header

Bytes 4 and 5 tell you the sizes. If byte 4 is `0x02`, you've got 2 PRG-ROM blocks = 32KB. If byte 5 is `0x01`, you've got 1 CHR-ROM block = 8KB. If byte 5 is `0x00`, the cartridge uses CHR-RAM.

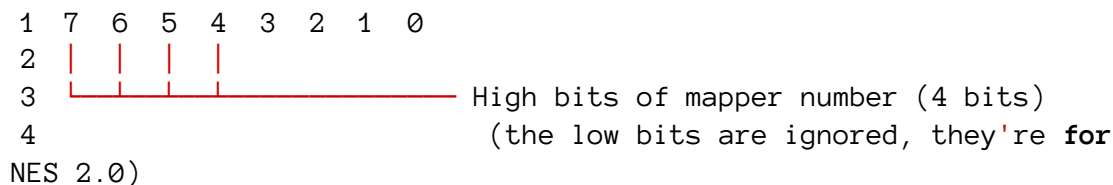
Flags 6 and 7

This is where things get packed bit by bit. Each bit means something different.

Flags 6:



Flags 7:



The **mapper number** is assembled by combining the upper 4 bits of Flags 6 (low nibble) with the upper 4 bits of Flags 7 (high nibble). This gives a number from 0 to 255 that identifies which mapper the cartridge uses.

A concrete example

Donkey Kong. Let's open the header in hexadecimal:

1	4E	45	53	1A	01	01	01	00	00	00	00	00	00	00	00
2															
3															
4															
5															
6															
7															
8															

Flags 6: 0x01 = 0000 0001
Vertical mirroring, no battery, no trainer
Mapper low = 0000 => mapper 0
CHR: 1 block = 8KB of graphics
PRG: 1 block = 16KB of code
Magic number "NES\x1A"

Total file size: 16 (header) + 16384 (PRG) + 8192 (CHR) = **24,592 bytes**. The entire Donkey Kong game, graphics included, is smaller than a favicon.

The trainer

You might have noticed that Flags 6 has a bit for “trainer”. A trainer is a 512-byte block injected between the header and the PRG-ROM.

Where did it come from? The copier era: devices that dumped cartridges to floppy disks in the 80s and 90s. Some copiers could inject extra code before the game to add cheats: infinite lives, level select, invincibility. Those 512 bytes of cheat code were called a “trainer”.

In practice, 99% of ROMs don't have a trainer. But if one does, we need to account for those extra 512 bytes when calculating where the PRG-ROM starts in the file:

1	Without trainer:	[Header 16 bytes]	[PRG-ROM...]
2	With trainer:	[Header 16 bytes]	[Trainer 512 bytes] [PRG-ROM...]

If we don't, we'll read the PRG-ROM offset by 512 bytes and everything blows up.

OceanofPDF.com

Mappers: why they exist and what they do

This is the single most important cartridge concept, and probably the most confusing one at first.

The problem is simple math: the 6502 CPU has a 16-bit address bus. That gives it access to $2^{16} = 64\text{KB}$ of address space. Of those 64KB, the range reserved for the cartridge is $\$8000 - \$FFFF = 32\text{KB}$.

Donkey Kong uses 16KB of code. Fits just fine. But Super Mario Bros uses 32KB. Barely squeaks in. And Zelda? 128KB. Mega Man 2? 256KB. No way those fit.

The solution is clever: the cartridge includes an extra logic chip (the **mapper**) that can swap which chunk of the ROM is visible at any given moment.

Think of it like a 200-page book on a lectern that only shows 2 pages at a time. You can't see the whole book at once, but you can flip pages. The mapper is the hand that flips them. The NES sees 32KB at a time, but the mapper can change *which* 32KB are showing.

This mechanism is called **bank switching**: swapping memory banks. The game tells the mapper "I need bank 3 now" by writing a value to a special address, and the mapper changes which portion of the ROM appears in the $\$8000 - \$FFFF$ space.

Over 200 different mappers were made, each with its own logic. Some games even had custom-designed mappers. We're going to implement two:

Mapper 0 (NROM)

The simplest. No bank switching. What you see is what you get.

It supports two configurations:

16KB of PRG-ROM: The code occupies $\$8000 - \$BFFF$ (16KB). And $\$C000 - \$FFFF$? It's a **mirror**: it shows exactly the same thing as

\$8000-\$BFFF. If you read \$C000, it's the same as reading \$8000. If you read \$FFFC (the RESET vector), it's the same as reading \$BFFC.

-
- 1 \$8000-\$BFFF => PRG-ROM (16KB)
 - 2 \$C000-\$FFFF => Mirror (same data as \$8000-\$BFFF)
-

32KB of PRG-ROM: The code occupies the entire \$8000-\$FFFF range. No mirrors, no tricks.

-
- 1 \$8000-\$FFFF => PRG-ROM (32KB)
-

How do we know if it's 16 or 32KB? The header tells us how many PRG blocks there are. If it says 1 (= 16KB), it's the first variant. If it says 2 (= 32KB), it's the second.

Mapper 0 games are the most primitive: Donkey Kong, Mario Bros (the arcade one), Balloon Fight, Ice Climber, Excitebike. Games that fit in 32KB or less.

Mapper 1 (MMC1)

Way more complex. This one has actual bank switching. The game writes bits one at a time (yes, one bit per write, 5 writes to send a single command) to mapper registers to choose which banks are visible.

It's used by Zelda, Metroid, Mega Man 2, Castlevania, and many more. We'll implement it in a later chapter, not now. But I wanted you to know it exists and what it's for.

Mirroring: the story of the mirrored screens

This is a PPU concept, but it's configured by the cartridge, so we'll cover it now. It can be confusing at first, so let's take it step by step.

The problem

The PPU builds the game's backgrounds using **nametables**: tables that say which tile goes in each screen position. The PPU's address space has room for 4 nametables, arranged in a 2x2 grid:

1		
2	Nametable 0	Nametable 1
3	(\$2000)	(\$2400)
4		
5	Nametable 2	Nametable 3
6	(\$2800)	(\$2C00)
7		

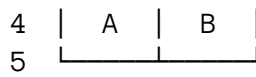
Each nametable is 1KB and describes a 256x240 pixel screen. Four nametables = 4KB. But the NES only has **2KB of Video RAM**. That's enough for two real nametables; the other two addresses just point to the same RAM as the first two. That's **mirroring**.

The question is: which pairs share RAM? That depends on the mirroring mode defined by the cartridge.

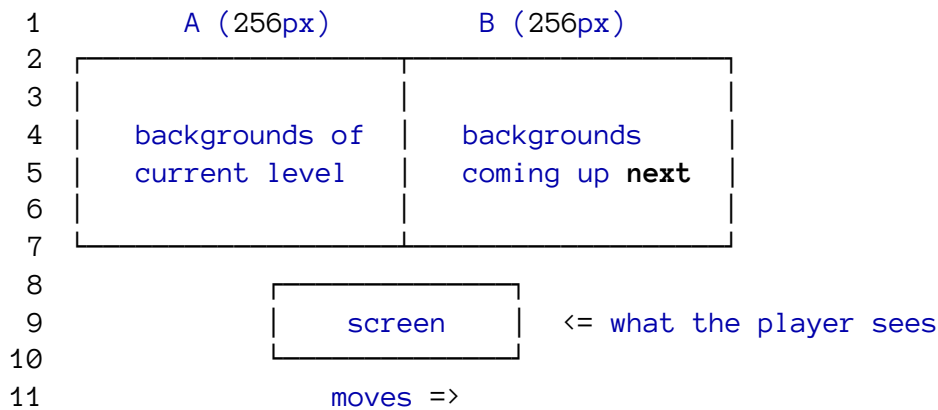
Vertical mirroring (horizontal scroll)

The two real nametables sit side by side:

1		
2	A	B
3		



This gives the PPU a 512-pixel-wide “canvas” (two screens side by side). The player sees a 256-pixel window that slides across it:



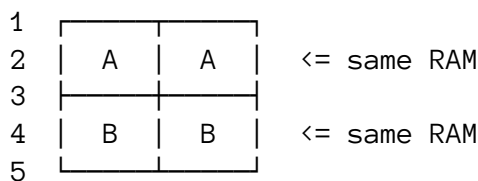
Here’s the trick: once the screen has scrolled far enough that it’s past A and showing B, **the game overwrites A with whatever comes next**. While the player is looking at B, A gets filled with new blocks, enemies, and clouds. Then when the viewport wraps back to A (which now has new content), the game overwrites B. And so on, forever.

It’s like a double-buffered conveyor belt: while the player is looking at one section, you prepare the other. That’s how Mario can scroll forever with just 2KB of VRAM. It doesn’t need the whole level in memory, just the two screens surrounding the player.

Super Mario Bros uses this mode.

Horizontal mirroring (vertical scroll)

The two real nametables are stacked:



Same idea, but vertical. The canvas is 480 pixels tall (two stacked screens), and the window slides up or down. The game overwrites whichever nametable isn't currently visible.

Kid Icarus and Ice Climber use this mode.

Why doesn't it work the other way around?

If Mario (which scrolls horizontally) used horizontal mirroring, it would only have one real nametable's worth of width (256px). To scroll, it would have to rewrite the nametable *while the PPU is drawing it on screen*. That causes glitches: flickering columns, half-drawn tiles, visual garbage.

With vertical mirroring, there's always a whole nametable "hidden" off to the side that the game can write to freely without the player noticing. The mirroring gives it a buffer in the right direction.

The names are confusing: "vertical mirroring" enables **horizontal** scrolling, and "horizontal mirroring" enables **vertical** scrolling. The names describe how the VRAM is laid out, not which direction you scroll. Yeah, I know.

Four-screen

Some cartridges include an extra 2KB of RAM on the board itself. Combined with the console's 2KB, you get 4KB = 4 real nametables, no mirroring needed. Full scrolling in any direction. Very few games use this.

Who decides the mode?

With Mapper 0, the mirroring is fixed: it's hardwired into the cartridge's circuitry. With fancier mappers (like Mapper 1), the game can change mirroring at runtime, which enables tricks like Metroid's multi-directional scrolling.

For now, it's just a value we parse from the header and stash away. When we build the PPU, we'll need it to figure out how to route nametable reads.

What do we need to build?

Now that you understand what a cartridge is and how it works, here's what we're going to build in the next two chapters:

Chapter 4.1 - Reading a ROM

We'll create the `Cartridge` class that takes a path to a `.nes` file and parses it:

- Validate the magic number
- Extract the PRG and CHR sizes from the header
- Parse the flags (mapper, mirroring, battery, trainer)
- Slice the file into the right pieces: PRG-ROM on one side, CHR-ROM on the other
- Test: open a real ROM and print what's inside

No emulation yet. Just pure parsing.

Chapter 4.2 - Mapper 0 and the first boot

We'll connect the cartridge to the emulator:

- Create Mapper 0 with the read logic (address => byte from PRG-ROM)
- Modify the Bus to route `$8000-$FFFF` to the cartridge's mapper
- Modify the Emulator to load `.nes` files
- Use the RESET vector so the CPU starts where the game says
- Test: load a real ROM and watch the CPU executing instructions from an actual game

If everything goes well, by the end of 4.2 you'll be able to do something like this:

```
1 emu = NES::Emulator.new
2 emu.load_rom("donkey_kong.nes")
3
```

```
4 100.times do
5   emu.step
6   emu.debug
7 end
```

And watch the CPU chew through the real Donkey Kong code. You won't see anything on screen yet (we need the PPU for that), but you'll know the emulator's brain is running a real game. That's the foundation everything else gets built on.

Let's get to it.

OceanofPDF.com

Chapter 4.1: Reading a ROM

In the previous chapter we learned what's inside a cartridge and what each part means. Now we're going to get our hands dirty: we'll write code that opens a `.nes` file, parses it, and tells us what's inside.

By the end of this chapter you'll be able to do this:

```
1 cart = NES::Cartridge.new("mario.nes")
2
3 cart.prg_rom    # => Bytes (32768 bytes)
4 cart.chr_rom    # => Bytes (8192 bytes)
5 cart.mapper_id  # => 0
6 cart.mirroring  # => Vertical
7 cart.battery    # => false
```

We're not going to connect anything to the emulator yet. Just pure parsing: open the file, validate it, and extract the data. Connecting the cartridge to the Bus and getting the CPU to execute real code is a problem for the next chapter.

OceanofPDF.com

The Cartridge class

Let's start from the end: what data do we want to extract from a .nes file?

```
1  # src/nes/cartridge.cr
2
3  module NES
4    class Cartridge
5      getter prg_rom    : Bytes      # Game code (executed by the
CPU)
6      getter chr_rom    : Bytes      # Graphics (read by the PPU)
7      getter mapper_id : UInt8      # Which mapper the cartridge
uses (0, 1, 2, etc.)
8      getter mirroring : Mirroring  # Enum representing the
mirroring mode (for the PPU)
9      getter battery   : Bool       # Has battery-backed RAM to
save progress?
10   end
11 end
```

Five pieces of data. That's all we need from the cartridge for now. PRG and CHR are the raw data blocks, and the other three are metadata from the header.

The format constants

Before parsing, we need to name each part of the file. Remember the header structure from the previous chapter:

1	Byte 0-3:	"NES" + 0x1A (magic number)
2	Byte 4:	PRG-ROM blocks (x 16KB)
3	Byte 5:	CHR-ROM blocks (x 8KB)
4	Byte 6:	Flags 6
5	Byte 7:	Flags 7
6	Byte 8-15:	Padding

All these positions and sizes go into a `Format` module, just like memory addresses live in `Bus : : Addresses`:

```
1  # src/nes/cartridge/format.cr
2
3  module NES
4    class Cartridge
5      module Format
6        # iNES header
7        HEADER_SIZE = 16
8        MAGIC        = Bytes[0x4E, 0x45, 0x53, 0x1A] # "NES\x1A"
9
10       # ROM block sizes
11       PRG_BLOCK_SIZE = 0x4000_u32 # 16 KB
12       CHR_BLOCK_SIZE = 0x2000_u32 # 8 KB
13
14       # Trainer
15       TRAINER_SIZE = 512
16
17       # Which header byte holds each piece of information
18       PRG_SIZE_BYTE = 4
19       CHR_SIZE_BYTE = 5
20       FLAGS_6_BYTE  = 6
21       FLAGS_7_BYTE  = 7
22     end
23   end
24 end
```

MAGIC is the signature that identifies a .nes file: the characters “NES” followed by byte 0x1A. If the first 4 bytes of the file don’t match, it’s not a valid ROM.

PRG_BLOCK_SIZE and CHR_BLOCK_SIZE are the size of each block. The header says “2 PRG blocks”, we multiply: $2 * 16\text{KB} = 32\text{KB}$.

The flag masks

Flags 6 and 7 are bytes where each bit means something different. To extract individual bits we use masks, exactly like we do with the CPU flags (FLAG_Z, FLAG_N, etc.):

```
1  # src/nes/cartridge/format.cr (continued inside module Format)
2
3      # Flags 6 masks
4      MIRROR_MASK      = 0b0000_0001_u8 # 0 = horizontal, 1 =
vertical
5      BATTERY_MASK     = 0b0000_0010_u8 # 1 = battery-backed RAM
6      TRAINER_MASK     = 0b0000_0100_u8 # 1 = 512-byte trainer
before PRG
7      FOUR_SCREEN_MASK = 0b0000_1000_u8 # 1 = four-screen VRAM
8      MAPPER_LOW_MASK  = 0b1111_0000_u8 # Mapper bits 0-3
9
10     # Flags 7 masks
11     MAPPER_HIGH_MASK  = 0b1111_0000_u8 # Mapper bits 4-7
```

If you’ve been following along, this should look familiar. flags_6 & BATTERY_MASK is the same thing as status & FLAG_Z. You apply the mask, and the AND tells you whether that bit was 1 or 0.

The Mirroring enum

The mirroring mode has three possible values. Instead of using a string or a number, we use an enum:

```
1 # src/nes/cartridge/mirroring.cr
2
3 module NES
4   class Cartridge
5     enum Mirroring
6       Horizontal
7       Vertical
8       FourScreen
9     end
10  end
11 end
```

An enum is a type that can only hold certain values. If you later write a case/switch on the mirroring and forget a case, the compiler will let you know. With a string, you're on your own.

We parse it now, but the PPU won't actually use it until we get to that chapter.

OceanofPDF.com

The parsing

With the constants in place, let's get to the logic. First, the full class structure and the initialize:

```
1  # src/nes/cartridge.cr
2
3  require "./cartridge/format"
4  require "./cartridge/mirroring"
5
6  module NES
7    class Cartridge
8      include Format
9
10     getter prg_rom    : Bytes
11     getter chr_rom    : Bytes
12     getter mapper_id  : UInt8
13     getter mirroring  : Mirroring
14     getter battery    : Bool
15
16     def initialize(path : String)
17       data = File.read(path).to_slice
18
19       @prg_rom    = Bytes.empty
20       @chr_rom    = Bytes.empty
21       @mapper_id  = 0_u8
22       @mirroring  = Mirroring::Horizontal
23       @battery    = false
24
25       parse(data)
26     end
```

`File.read` returns a `String` (text). `.to_slice` converts it to `Bytes` (a raw byte array). A `.nes` file isn't text, it's bytes, so we need `Bytes` to access by position (`data[4]`) and cut slices (`data[prg_start, prg_size]`).

Crystal requires all instance variables to be initialized before calling any method, which is why we set those default values before `parse`. They're just placeholders that get overwritten immediately.

Validation

The first thing `parse` does is check that the file is actually a legitimate `.nes` file, using a helper method `validate!`:

```
1 # src/nes/cartridge.cr
2
3 def validate!(data : Bytes)
4   if data.size < HEADER_SIZE
5     raise "Invalid iNES file: too small ({data.size} bytes,
minimum #{HEADER_SIZE})"
6   end
7
8   unless data[0, 4] == MAGIC
9     raise "Invalid iNES file: wrong magic number (expected
NES\\x1A)"
10  end
11 end
```

Two checks: is the file at least 16 bytes long (the header size), and do the first 4 bytes match the `NES\\x1A` signature? If either fails, we blow up with a clear message.

Parsing the mapper

The mapper number is built by combining bits from two different bytes:

```
1 # src/nes/cartridge.cr
2
3 def parse_mapper_id(flags_6 : UInt8, flags_7 : UInt8) : UInt8
4   low  = (flags_6 & MAPPER_LOW_MASK) >> 4
5   high = flags_7 & MAPPER_HIGH_MASK
6
7   high | low
8 end
```

The top 4 bits of `flags_6` hold the low nibble of the mapper number. The top 4 bits of `flags_7` hold the high nibble. We combine them with OR.

Concrete example with Super Mario Bros:

```
1 flags_6 = 0x01 = 0000 0001
2 flags_7 = 0x00 = 0000 0000
3
4 low  = (0000 0001 & 1111 0000) >> 4 = 0000 0000 >> 4 = 0000
0000
5 high = (0000 0000 & 1111 0000)      = 0000 0000
6
7 high | low = 0000 0000 = mapper 0
```

Mapper 0. NROM. The simplest one.

Parsing the mirroring

```
1 # src/nes/cartridge.cr
2
3 def parse_mirroring(flags_6 : UInt8) : Mirroring
4   if (flags_6 & FOUR_SCREEN_MASK) != 0
5     Mirroring::FourScreen
6   elsif (flags_6 & MIRROR_MASK) != 0
7     Mirroring::Vertical
8   else
9     Mirroring::Horizontal
10  end
11 end
```

Four-screen takes priority: if that bit is set, the other bits don't matter. Otherwise, we check the mirror bit: 1 = vertical, 0 = horizontal.

The parse method

Now we tie it all together:

```
1 # src/nes/cartridge.cr
2
3 def parse(data : Bytes)
4   validate!(data)
5
6   flags_6 = data[FLAGS_6_BYTE]
```

```

7      flags_7 = data[FLAGS_7_BYTE]
8
9      prg_size = data[PRG_SIZE_BYTE].to_u32 * PRG_BLOCK_SIZE
10     chr_size = data[CHR_SIZE_BYTE].to_u32 * CHR_BLOCK_SIZE
11
12     @mapper_id = parse_mapper_id(flags_6, flags_7)
13     @mirroring = parse_mirroring(flags_6)
14     @battery   = (flags_6 & BATTERY_MASK) != 0
15
16     has_trainer = (flags_6 & TRAINER_MASK) != 0
17     prg_start   = HEADER_SIZE + (has_trainer ? TRAINER_SIZE :
0)
18     chr_start   = prg_start + prg_size
19
20     @prg_rom = data[prg_start, prg_size]
21     @chr_rom = chr_size > 0 ? data[chr_start, chr_size] :
Bytes.new(CHR_BLOCK_SIZE, 0_u8)
22     end

```

The flow is linear, top to bottom:

1. Validate it's a .nes file
2. Read the two flag bytes
3. Calculate sizes (blocks x block size)
4. Parse metadata (mapper, mirroring, battery)
5. Calculate offsets (where PRG starts in the file, where CHR starts)
6. Cut the slices

The most interesting line is the last one. If `chr_size` is 0, the cartridge has no CHR-ROM and uses CHR-RAM instead. In that case we create 8KB of zeros: `Bytes.new(CHR_BLOCK_SIZE, 0_u8)`. It's empty RAM where the PPU will write graphics at runtime.

The `prg_start` calculation accounts for the trainer: if there's a trainer (512 bytes), PRG starts 512 bytes after the header. If not, it starts right after the header. Remember that the trainer is a relic from the copier era. 99% of ROMs don't have one, but if we don't account for those 512 bytes, all our offsets will be wrong.

Wiring it up

We add the require in the root module:

```
1  # src/nest.cr
2
3  require "./nes/emulator"
4  require "./nes/cartridge"
5
6  module NES
7    VERSION = "1.0"
8  end
```

[OceanofPDF.com](https://oceanofpdf.com)

Test

Let's test with Super Mario Bros. You'll need a .nes file for some game. You can find ROMs at:

- [Emu-Land](#)
- [Retrostic](#)
- [RomsFun](#)

The book's repo already includes mario.nes in the .roms/ folder. If you want to test with other games, put them there too.

```
1  # bin/test_cartridge.cr
2
3  require "../src/nas"
4
5  puts "CARTRIDGE"
6
7  all_passed = true
8
9  # =====
10 # Test 1: Parse a real ROM
11 # =====
12 puts "\n 1. Parse ROM header"
13
14 cart = NES::Cartridge.new(".roms/mario.nes")
15
16 puts "    PRG-ROM: #{cart.prg_rom.size / 1024}KB (#
{cart.prg_rom.size} bytes)"
17 puts "    CHR-ROM: #{cart.chr_rom.size / 1024}KB (#
{cart.chr_rom.size} bytes)"
18 puts "    Mapper:  #{cart.mapper_id}"
19 puts "    Mirror:   #{cart.mirroring}"
20 puts "    Battery:  #{cart.battery}"
21
22 # Super Mario Bros: 32KB PRG, 8KB CHR, mapper 0, vertical
mirroring, no battery
23 passed = cart.prg_rom.size == 32768
24 puts passed ? "    PRG size correct  " : "    PRG size wrong
X (expected 32768)"
25 all_passed &= passed
```



```

26
27 passed = cart.chr_rom.size == 8192
28 puts passed ? "      CHR size correct ✓ " : "      CHR size wrong
✗ (expected 8192)"
29 all_passed &= passed
30
31 passed = cart.mapper_id == 0
32 puts passed ? "      Mapper correct ✓ " : "      Mapper wrong ✗
(expected 0)"
33 all_passed &= passed
34
35 passed = cart.mirroring == NES::Cartridge::Mirroring::Vertical
36 puts passed ? "      Mirroring correct ✓ " : "      Mirroring wrong
✗ (expected Vertical)"
37 all_passed &= passed
38
39 passed = cart.battery == false
40 puts passed ? "      Battery correct ✓ " : "      Battery wrong ✗
(expected false)"
41 all_passed &= passed
42
43 # =====
44 # Test 2: Invalid file
45 # =====
46 puts "\n 2. Reject invalid file"
47
48 begin
49   NES::Cartridge.new("src/nas.cr")
50   puts "      Should have raised ✗"
51   all_passed &= false
52 rescue e
53   passed = e.message.to_s.includes?("magic number")
54   puts passed ? "      Rejected with correct error ✓ " : "
Wrong error: #{e.message} ✗"
55   all_passed &= passed
56 end
57
58 # =====
59 puts all_passed ? "\n✓ CARTRIDGE passed" : "\n✗ CARTRIDGE
failed"

```

Run it:

```
1 crystal run bin/test_cartridge.cr
```

You should see:

```
1 CARTRIDGE
2
3   1. Parse ROM header
4     PRG-ROM: 32.0KB (32768 bytes)
5     CHR-ROM: 8.0KB (8192 bytes)
6     Mapper: 0
7     Mirror: Vertical
8     Battery: false
9     PRG size correct ✓
10    CHR size correct ✓
11    Mapper correct ✓
12    Mirroring correct ✓
13    Battery correct ✓
14
15   2. Reject invalid file
16     Rejected with correct error ✓
17
18 ✓ CARTRIDGE passed
```

If you have other ROMs, try them out. Zelda should give mapper 1 and battery true. Donkey Kong should give 16KB of PRG instead of 32KB.

OceanofPDF.com

Updated structure

```
1 nes-emulator/
2 |─ bin/
3 |   |─ test_cpu.cr
4 |   |─ test_cartridge.cr      <= NEW
5 |   |─ cpu/
6 |   |   └─ ... (CPU tests)
7 |─ src/
8 |   |─ nes.cr                (+ require cartridge)
9 |   |─ nes/
10 |      |─ bus.cr
11 |      |─ cpu.cr
12 |      |─ emulator.cr
13 |      |─ cartridge.cr      <= NEW
14 |      |─ cartridge/
15 |      |   |─ format.cr      <= NEW
16 |      |   └─ mirroring.cr   <= NEW
```

Wrap up

With 3 files and under 80 lines of code, our emulator can read real .nes files. It parses the header, extracts the PRG-ROM and CHR-ROM, and knows which mapper and mirroring mode the cartridge uses.

But the Cartridge is still floating in space. It's not wired to anything. If you ask the CPU to read \$8000, the Bus still returns 0x00 because it has no idea a cartridge exists.

In the next chapter we'll implement Mapper 0, connect the cartridge to the Bus, and get the CPU to start executing real game code. That's when everything comes together.

You can check out the full code at the [repo](#).

OceanofPDF.com

Chapter 4.2: Mapper 0 and the First Boot

In the previous chapter we parsed a .nes file and extracted the PRG-ROM, CHR-ROM, and metadata. But the Cartridge is just floating in the void. If the CPU asks to read address \$8000, the Bus still returns 0x00 because it has no idea a cartridge exists.

In this chapter we're going to wire everything up: create the mapper that translates addresses, connect the cartridge to the Bus, and get the CPU executing real code from a ROM. By the end, we'll run the most important test suite in the NES ecosystem and verify that all official instructions work correctly.

OceanofPDF.com

The translation problem

The CPU asks for address \$8000. But the PRG-ROM is a byte array starting at index 0. Someone has to translate \$8000 to index 0, \$8001 to index 1, and so on. That someone is the **mapper**.

For a 32KB cartridge, the translation is straightforward: subtract \$8000 from the address and you get the index. Or even better, just AND it with a mask.

1	Address:	\$8000	=	1000 0000 0000 0000	
2	Mask:	\$7FFF	=	0111 1111 1111 1111	
3		AND	=	0000 0000 0000 0000	= index 0
4					
5	Address:	\$FFFC	=	1111 1111 1111 1100	
6	Mask:	\$7FFF	=	0111 1111 1111 1111	
7		AND	=	0111 1111 1111 1100	= index \$7FFC

\$8000 & \$7FFF = \$0000. \$FFFC & \$7FFF = \$7FFC. The mask strips bit 15 and gives you the offset within the PRG-ROM. Works for the entire \$8000-\$FFFF range.

Now, what happens with a 16KB cartridge? The PRG-ROM has 16384 bytes (indices 0 to \$3FFF). But the address range is still 32KB (\$8000-\$FFFF). So \$8000-\$BFFF maps directly to the PRG-ROM, and \$C000-\$FFFF is a **mirror**: it maps back to the same data.

The mask for that is \$3FFF:

1	\$8000 & \$3FFF	= \$0000	(first byte of PRG)
2	\$BFFF & \$3FFF	= \$3FFF	(last byte of PRG)
3	\$C000 & \$3FFF	= \$0000	(first byte of PRG, again)
4	\$FFFC & \$3FFF	= \$3FFC	(RESET vector, falls within PRG)

\$C000 and \$8000 give the same index. That's the mirror.

The Mapper class

We need a base class that defines the interface every mapper has to implement:

```
1  # src/nes/cartridge/mapper.cr
2
3  module NES
4    class Cartridge
5      class Mapper
6        def self.create(id : UInt8, prg_rom : Bytes, chr_rom :
Bytes) : Mapper
7          case id
8            when 0 then Mapper000.new(prg_rom, chr_rom)
9            else raise "Mapper #{id} not implemented"
10         end
11       end
12
13       def prg_read(address : UInt16) : UInt8
14         raise "Not implemented"
15       end
16
17       def chr_read(address : UInt16) : UInt8
18         raise "Not implemented"
19       end
20
21       def chr_write(address : UInt16, value : UInt8)
22         raise "Not implemented"
23       end
24     end
25   end
26 end
```

`self.create` is a class method (the `self.` means it's called on the class, not on an instance: `Mapper.create(id, prg, chr)` instead of `Mapper.new.create(...)`). It takes the mapper id we parsed from the header in the previous chapter and returns the right instance. For now only Mapper 0 exists. When we add Mapper 1 later, it's just one more when 1 in the case.

The base class methods just `raise`. If a mapper subclass forgets to override them, it blows up with a clear error instead of returning garbage data. `prg_read` is what the CPU will use (through the Bus) to read code from the cartridge. `chr_read` and `chr_write` are for the PPU; we're defining them now but won't use them yet.

OceanofPDF.com

Mapper 0: NROM

The simplest mapper there is. No bank switching, no registers, no tricks. Pure address translation with a mask:

```
1  # src/nes/cartridge/mappers/mapper_000.cr
2
3  module NES
4    class Cartridge
5      class Mapper000 < Mapper
6        PRG_MASK_16K = 0x3FFF_u16
7        PRG_MASK_32K = 0x7FFF_u16
8        CHR_MASK     = 0x1FFF_u16
9
10       def initialize(@prg_rom : Bytes, @chr_rom : Bytes)
11         @prg_mask = @prg_rom.size > Format::PRG_BLOCK_SIZE ?
PRG_MASK_32K : PRG_MASK_16K
12       end
13
14       def prg_read(address : UInt16) : UInt8
15         @prg_rom[address & @prg_mask]
16       end
17
18       def chr_read(address : UInt16) : UInt8
19         @chr_rom[address & CHR_MASK]
20       end
21     end
22   end
23 end
```

The entire mapper is three constants and one `if` in the constructor.

The constructor checks the size of the PRG-ROM. If it's more than 16KB (one block), it uses the 32KB mask (\$7FFF). Otherwise, it uses the 16KB mask (\$3FFF), which automatically mirrors the second half of the address space.

`prg_read` is one line: the AND with the mask gives you the index, and you access the array. `chr_read` is the same but with the CHR mask (8KB = \$1FFF).

That's all Donkey Kong, Excitebike, and Mario Bros (the arcade game) need to run. Mapper 1 (MMC1) is quite a bit more complex: it has real bank switching with a 5-bit shift register. We'll tackle that later, once we have the PPU working. Right now the priority is getting the full emulator wired up end-to-end.

OceanofPDF.com

Connecting the mapper to the Cartridge

We just need `Cartridge` to expose its mapper. In `cartridge.cr`, we add the `require`, the `getter`, and one line at the end of `parse`:

```
1 # src/nes/cartridge.cr
2
3 require "./cartridge/mapper"
4 require "./cartridge/mappers/mapper_000"
5
6 # ... inside the class, next to the other getters:
7 getter mapper : Mapper
8
9 # ... at the end of parse():
10 @mapper = Mapper.create(@mapper_id, @prg_rom, @chr_rom)
```

Plumbing. Nothing more.

OceanofPDF.com

The Bus learns to read from the cartridge

Up until now, the Bus only knew about RAM. If the CPU asked for any address outside \$0000-\$1FFF, it silently got back 0x00. Now we need reads in the \$8000-\$FFFF range to go through the cartridge's mapper.

First, the constants. In addresses.cr we add the PRG-ROM range:

```
1  # src/nes/bus/addresses.cr
2
3  module NES
4    class Bus
5      module Addresses
6        # ===== CPU RAM =====
7        RAM_SIZE = 2048
8
9        RAM_START      = 0x0000_u16
10       RAM_END        = 0x07FF_u16
11       RAM_MIRROR_END = 0x1FFF_u16
12
13       RAM_MASK = 0x07FF_u16
14
15       # ===== PRG-ROM (Cartridge) =====
16       PRG_ROM_START = 0x8000_u16
17       PRG_ROM_END   = 0xFFFF_u16
18     end
19   end
20 end
```

The two new constants are PRG_ROM_START and PRG_ROM_END. The \$8000-\$FFFF range covers the 32KB of address space where the cartridge code lives.

Now the Bus. Before it looked like this:

```
1  # Bus BEFORE (no cartridge)
2
3  def initialize
4    @ram = Bytes.new(RAM_SIZE, 0_u8)
5  end
```

```
6
7 def cpu_read(address : UInt16) : UInt8
8   case address
9     when RAM_START..RAM_MIRROR_END
10      read_ram(address)
11   else
12     0_u8
13   end
14 end
```

Now:

```
1 # src/nes/bus.cr
2
3 module NES
4   class Bus
5     include Addresses
6
7     def initialize(cartridge : Cartridge)
8       @cartridge = cartridge
9       @ram = Bytes.new(RAM_SIZE, 0_u8)
10    end
11
12    def cpu_read(address : UInt16) : UInt8
13      case address
14        when RAM_START..RAM_MIRROR_END
15          read_ram(address)
16        when PRG_ROM_START..PRG_ROM_END
17          @cartridge.mapper.prg_read(address)
18        else
19          0_u8
20        end
21      end
22
23    def cpu_write(address : UInt16, value : UInt8)
24      case address
25        when RAM_START..RAM_MIRROR_END
26          write_ram(address, value)
27        end
28      end
29    end
30  end
```

Two changes. The constructor now receives a `Cartridge` and stores it. And `cpu_read` has a new case: if the address falls in `$8000-$FFFF`, it delegates to the cartridge's mapper.

Notice that `cpu_write` does **not** route anything to the cartridge. It's ROM: Read Only Memory. The CPU can read from the cartridge but can't write to it. (Technically some mappers do intercept writes for bank switching, but that's a Mapper 1 topic.)

OceanofPDF.com

The boot: from file to Program Counter

The Emulator is the class that coordinates everything. Before, it created an empty Bus and let you load small programs with `load_program`. Now it takes a path to a `.nes` file and does a full boot:

```
1  # src/nes/emulator.cr
2
3  require "./bus"
4  require "./cpu"
5  require "./emulator/debugging"
6
7  module NES
8    class Emulator
9      include Debugging
10
11      getter cartridge : Cartridge
12      getter cpu       : CPU
13      getter bus       : Bus
14
15      def initialize(path : String)
16        @cartridge = Cartridge.new(path)
17        @bus = Bus.new(@cartridge)
18        @cpu = CPU.new(@bus)
19
20        @cpu.reset
21      end
22
23      def step
24        cycles = @cpu.step
25        # PPU steps cycles * 3
26      end
27    end
28  end
```

Four lines in the constructor. Four lines that build the complete chain:

1. `Cartridge.new(path)`: parses the `.nes`, extracts the PRG-ROM, creates Mapper 0
2. `Bus.new(@cartridge)`: creates the Bus and connects the cartridge

3. `CPU.new(@bus)`: creates the CPU and connects it to the Bus
4. `@cpu.reset`: reads the RESET vector and sets the PC

That last line is the key moment. Let's see what `reset` does:

```
1 # src/nes/cpu.cr (inside the CPU class)
2
3 def reset
4   lo = read_byte(RESET_VECTOR)
5   hi = read_byte(RESET_VECTOR &+ 1)
6
7   @pc = (hi.to_u16 << 8) | lo.to_u16
8 end
```

Remember chapter 3.6, where we defined the interrupt vectors and said “when we implement the cartridge we’ll use the RESET vector properly”? That time is now.

`reset` reads two bytes from `$FFFC` and `$FFFD` (the RESET vector), combines them into a 16-bit address (little-endian, as always on the 6502), and sets the PC there. That’s the first instruction the game will execute.

Let’s trace the full chain with Super Mario Bros:

```
1 1. Emulator.new("mario.nes")
2 2. Cartridge parses the header: PRG 32KB, Mapper 0
3 3. Mapper000 is created with mask $7FFF (32KB, no mirror)
4 4. Bus is created with the cartridge connected
5 5. CPU is created, reset() runs:
6   - read_byte($FFFC) => Bus.cpu_read($FFFC)
7                       => mapper.prg_read($FFFC)
8                       => prg_rom[$FFFC & $7FFF]
9                       => prg_rom[$7FFC]
10                      => $00 (low byte)
11   - read_byte($FFFD) => ... => prg_rom[$7FFD] => $80
    (high byte)
12   - PC = $8000
13 6. step() => fetch_byte() reads $8000 => first instruction of the
game
```

From file to Program Counter. The CPU is sitting at `$8000`, ready to execute Mario’s first instruction.

OceanofPDF.com

UnknownOpcode

Before running the test suite, there's one small addition that will make our lives much easier: a custom exception for unknown opcodes.

```
1 # src/nes/cpu/exceptions.cr
2
3 module NES
4   class UnknownOpcode < Exception
5     def initialize(opcode : UInt8)
6       super("Unknown opcode: 0x%02X" % opcode)
7     end
8   end
9 end
```

And in `cpu.cr`, at the end of the case in `step`:

```
1 # src/nes/cpu.cr (inside step, at the end of the case)
2
3   else raise UnknownOpcode.new(opcode)
4   end
```

Why not just `raise` with a generic string? Because a custom exception lets you do `rescue NES::UnknownOpcode` and handle that specific case. This matters for the test suite: our emulator implements the 151 official instructions, but the test ROM covers all 256 (including the unofficial ones). When it hits an unofficial instruction, we want to stop cleanly instead of blowing up with a generic error.

The message also includes the opcode in hex. When something fails, you'll know exactly which instruction the CPU choked on. That's the difference between a 30-second debug session and a 30-minute one.

nestest: trial by fire

There's a special ROM called **nestest** that's the de facto standard for verifying a 6502 CPU implementation. It was written by Kevin Horton and tests all official opcodes (plus the unofficial ones, but we don't care about those). You can find it at [christopherpow/nes-test-roms](https://github.com/christopherpow/nes-test-roms) (look for `nestest.nes` and `nestest.log` in the `other/` folder). The book's repo already has them in `.roms/test/`.

The ROM has two modes:

- **Visual:** needs the PPU to display results on screen. Starts at `$C004`.
- **Automated:** doesn't need the PPU, just executes instructions and leaves the result in memory. Starts at `$C000`.

The RESET vector of nestest points to `$C004` (visual mode). Since we don't have a PPU yet, we force the PC to `$C000` to use automated mode.

To do that, we need to be able to write to the PC from outside. In `cpu.cr`, the PC was declared as `getter` (read-only). We change it to `property` (read and write):

```
1 # src/nes/cpu.cr
2
3 property pc : UInt16 # Was getter, now property so we can force
  PC from tests
```

The ROM ships with a log file (`nestest.log`) that contains the expected CPU state **before** each instruction. Each line looks like this:

```
1 C000 4C F5 C5 JMP $C5F5 A:00 X:00 Y:00
P:24 SP:FD PPU: 0, 21 CYC:7
```

Our test reads line by line, compares the CPU state against what's expected, executes a step, and continues. If anything doesn't match, it fails with details

about what it expected and what it got. When it reaches an unofficial opcode (which we don't implement), UnknownOpcode stops it cleanly.

```
1  # bin/test_nestest.cr
2
3  require "../src/nes"
4
5  emulator = NES::Emulator.new(".roms/test/nestest.nes")
6
7  # Override: automated mode starts at $C000
8  emulator.cpu.pc = 0xC000_u16
9
10 log_lines = File.readlines(".roms/test/nestest.log")
11
12 log_lines.each_with_index do |line, i|
13   expected_pc = line[0, 4].to_u16(16)
14   expected_a  = line[50, 2].to_u8(16)
15   expected_x  = line[55, 2].to_u8(16)
16   expected_y  = line[60, 2].to_u8(16)
17   expected_p  = line[65, 2].to_u8(16)
18   expected_sp = line[71, 2].to_u8(16)
19
20   cpu = emulator.cpu
21   matches = cpu.pc == expected_pc && cpu.a == expected_a &&
22     cpu.x == expected_x && cpu.y == expected_y &&
23     cpu.status == expected_p && cpu.sp == expected_sp
24
25   unless matches
26     puts "FAIL at line #{i + 1}: #{line}"
27     puts "  got PC:%04X A:%02X X:%02X Y:%02X P:%02X SP:%02X" % {
28       cpu.pc, cpu.a, cpu.x, cpu.y, cpu.status, cpu.sp
29     }
30     exit 1
31   end
32
33   begin
34     emulator.step
35   rescue e : NES::UnknownOpcode
36     break
37   end
38 end
39
40 puts "PASS"
```

The log parsing is mechanical: the values sit at fixed positions within each line (PC at characters 0-3, A at 50-51, etc.). Feel free to just copy this.

This is where the custom exception pays off: when nestest reaches an unofficial opcode, the `rescue NES::UnknownOpcode` catches it and the `break` ends the loop cleanly. It's not an error. It's just where our implementation says "this is as far as I go."

Run it:

```
1 crystal run bin/test_nestest.cr
```

```
1 PASS
```

All official instructions executed correctly. If something fails for you, the error tells you exactly which line of the log has the mismatch and what value is off. From there you can look at the log, identify the instruction, and review your implementation.

OceanofPDF.com

Updated file structure

```
1 nes-emulator/
2 |─ bin/
3 |   |─ test_cartridge.cr
4 |   |─ test_nestest.cr           <= NEW
5 |─ src/
6 |   |─ nes.cr
7 |   |─ nes/
8 |       |─ bus.cr               (+ cartridge in
constructor, PRG routing)
9 |       |─ cpu.cr               (+ reset, +
UnknownOpcode in step, getter pc => property pc)
10 |      |─ emulator.cr          (rewritten: receives
path, does boot)
11 |      |─ cartridge.cr         (+ mapper getter)
12 |      |─ bus/
13 |      |   |─ addresses.cr     (+
PRG_ROM_START/END)
14 |      |─ cpu/
15 |      |   |─ addressing_modes.cr
16 |      |   |─ flags.cr
17 |      |   |─ stack.cr
18 |      |   |─ interrupts.cr
19 |      |   |─ exceptions.cr    <= NEW
20 |      |   |─ instructions.cr
21 |      |   |─ op_codes.cr
22 |      |   |─ instructions/
23 |      |       |─ ... (all instruction files)
24 |      |─ cartridge/
25 |      |   |─ format.cr
26 |      |   |─ mirroring.cr
27 |      |   |─ mapper.cr        <= NEW
28 |      |   |─ mappers/
29 |      |       |─ mapper_000.cr <= NEW
30 |      |─ emulator/
31 |      |   |─ debugging.cr
```

Wrap up

Three new files (`mapper.cr`, `mapper_000.cr`, `exceptions.cr`) and three modified files (`bus.cr`, `emulator.cr`, `cartridge.cr`). With that, the complete chain works:

```
1 .nes file => Cartridge (parsing) => Mapper (translation) =>
  Bus (routing) => CPU (execution)
```

The CPU reads the RESET vector, knows where to start, and executes real instructions from an actual game. nestest confirms that all 151 official instructions work correctly.

There's no screen yet. If you run Mario, the CPU is executing the game's code, but you can't see anything. It's like a car engine running on a workbench: it works, it makes noise, but it's not going anywhere. For that, we need the PPU.

That's what's coming in the next chapter.

You can check out the changes in the [repo](#).

[OceanofPDF.com](https://oceanofpdf.com)

Chapter 5: The PPU

Our CPU runs real code, but if you boot Super Mario Bros right now, you see nothing. The game is running internally (Mario is in there, jumping, stomping Goombas) but nothing is drawing the screen.

The chip responsible for drawing is the **PPU** (Picture Processing Unit). It's the second most important component in the NES, and in many ways the most interesting one. It has its own memory, its own bus, and runs at 3 times the speed of the CPU.

This chapter is pure theory. We're not writing any code yet, but you need to understand how the PPU works before implementing it. In the CPU chapters the question was "how does it execute instructions?" Here the question is "how does it draw a screen?".



Figure 6. PPU Diagram

The NES screen

The NES generates an image of **256x240 pixels**, 60 times per second. Each frame is drawn line by line, top to bottom, left to right. The PPU doesn't "render a full scene" all at once like a modern GPU would. It draws one pixel, moves on, draws another, moves on, and repeats 61,440 times (256 x 240) until it completes a frame.

The PPU has **64 possible colors**. Not 16 million like your monitor. Sixty-four. And out of those 64, at any given moment it can only show 25 on screen.

That's what they had to work with when building the worlds of Mario, the dungeons of Zelda, and the space vistas of Metroid. Creative limitations pushed to the absolute extreme.

OceanofPDF.com

Tiles: the building blocks of everything

Absolutely everything you see on the NES is made of **tiles**: 8x8 pixel blocks. Mario is made of tiles. The floor is made of tiles. The clouds, the question blocks, the pipes, the text “WORLD 1-1” - all 8x8 tiles.

Why 8x8? Because the PPU can't draw arbitrary pixels. There's no “paint pixel (142, 87) red” command. Instead, the PPU sweeps across the screen and assembles each pixel by looking up tables that say “at this position, use tile number 37 with this color palette”.

OceanofPDF.com

Pattern tables: the tile catalog

Tiles are stored in the **pattern tables**. Think of them as a catalog: 256 tiles numbered 0 to 255, each containing a little 8x8 pixel drawing. There are two pattern tables, each with its own set of 256 tiles:

1	\$0000-\$0FFF	Pattern Table 0 (256 tiles)
2	\$1000-\$1FFF	Pattern Table 1 (256 tiles)

Typically one pattern table is used for the background and the other for sprites, though the game chooses which is which through a PPU register.

In Mario, the background pattern table contains the tiles for ground blocks, clouds, pipes, question blocks, score numbers. The sprite pattern table contains the tiles for Mario walking, jumping, the Goombas, the Koopa Troopas, the coins, the mushrooms.

The pattern table data comes from the **CHR-ROM** on the cartridge - the graphics data we parsed in chapter 4 and set aside for “when we get to the PPU.” Well, here we are.

How a tile is encoded

Each tile takes up 16 bytes. An 8x8 tile where each pixel has 2 bits of color works out to 8 rows x 2 bits x 8 columns... but instead of storing the 2 bits together, they’re split across two **bitplanes**.

Let’s say we want to represent the letter “A” in a tile:

1	Tile visual (2-bit color per pixel):															
2	.	.	1	1	1	1	.	.								
3	.	1	.	.	.	.	1	.								
4	.	1	.	.	.	.	1	.								
5	.	1	1	1	1	1	1	.								
6	.	1	.	.	.	.	1	.								
7	.	1	.	.	.	.	1	.								
8	.	1	.	.	.	.	1	.								
9	.	.	.	.	.	.	.	.								

Each pixel can have a value of 0, 1, 2, or 3. Those 2 bits come from combining two planes:

1	Bitplane 0 (low bit):	Bitplane 1 (high bit):
2	0 0 1 1 1 1 0 0 = \$3C	0 0 0 0 0 0 0 0 = \$00
3	0 1 0 0 0 0 1 0 = \$42	0 0 0 0 0 0 0 0 = \$00
4	0 1 0 0 0 0 1 0 = \$42	0 0 0 0 0 0 0 0 = \$00
5	0 1 1 1 1 1 1 0 = \$7E	0 0 0 0 0 0 0 0 = \$00
6	0 1 0 0 0 0 1 0 = \$42	0 0 0 0 0 0 0 0 = \$00
7	0 1 0 0 0 0 1 0 = \$42	0 0 0 0 0 0 0 0 = \$00
8	0 1 0 0 0 0 1 0 = \$42	0 0 0 0 0 0 0 0 = \$00
9	0 0 0 0 0 0 0 0 = \$00	0 0 0 0 0 0 0 0 = \$00

In memory it's stored: first the 8 rows of bitplane 0, then the 8 rows of bitplane 1. Total: 16 bytes.

To get the color of a pixel: you take the corresponding bit from bitplane 0 (low bit) and bitplane 1 (high bit), combine them, and you get a value from 0 to 3. In this example all the "lit" pixels have color 1 (only the low bitplane is active). If the high bitplane were also active on those pixels, the color would be 3.

A tile of Mario's head with more colors might look like this:

1	Pixel color:	Bitplane 0:	Bitplane 1:
2	. . . 3 3 3 . .	0 0 0 1 1 1 0 0	0 0 0 1 1 1 0 0
3	. . 3 3 3 3 3 .	0 0 1 1 1 1 1 0	0 0 1 1 1 1 1 0
4	. . 1 1 2 1 2 .	0 0 1 1 0 1 0 0	0 0 0 0 1 0 1 0
5	. 1 2 1 2 2 2 .	0 1 0 1 0 0 0 0	0 0 1 0 1 1 1 0
6	. 1 2 1 1 2 2 1	0 1 0 1 1 0 0 1	0 0 1 0 0 1 1 0
7	. . 1 2 2 2 . .	0 0 1 0 0 0 0 0	0 0 0 1 1 1 0 0
8	. . 1 1 1 1 . .	0 0 1 1 1 1 0 0	0 0 0 0 0 0 0 0
9	. . 1 1 1 1 . .	0 0 1 1 1 1 0 0	0 0 0 0 0 0 0 0

Here we have 3 visible colors: 1 = skin, 2 = hair/eyes, 3 = cap. Only 3? Yep. With 2 bits per pixel you can represent 4 values (0, 1, 2, 3), but value 0 is reserved as **transparent**. That's what lets you see the background behind the sprite. So each tile gets only 3 real colors, plus transparency. This applies to both background and sprite tiles.

Why this convoluted format instead of just storing 2 bits per pixel? Because that's how the PPU hardware is wired. It reads one byte from each plane in parallel and combines them with minimal logic. 80s-era efficiency at its finest.

OceanofPDF.com

Nametables: the background map

Tiles are the building blocks. The **nametable** is the blueprint that says where each block goes.

The NES screen is 256x240 pixels. Each tile is 8x8 pixels. That gives a grid of 32 columns x 30 rows = **960 tiles** per screen. The nametable is exactly that: 960 bytes where each byte is the *index* of the tile in the pattern table that goes at that position.

When Mario's NMI handler sets up World 1-1, it writes something like this to the nametable:

```
1 Position (0,0): tile $24 (empty sky)
2 Position (1,0): tile $24 (empty sky)
3 ...
4 Position (0,25): tile $B4 (ground block)
5 Position (1,25): tile $B4 (ground block)
6 ...
7 Position (16,3): tile $C0 (question block, top-left part)
```

The PPU sweeps through the nametable position by position, looks up the corresponding tile in the pattern table, and draws it. The result is the game's background.

Four nametables, two real

The PPU's memory space has room for 4 nametables:

1		
2	Nametable 0	Nametable 1
3	(\$2000)	(\$2400)
4		
5	Nametable 2	Nametable 3
6	(\$2800)	(\$2C00)
7		

Each nametable takes up 1KB (960 bytes of tiles + 64 bytes of attributes, more on that soon). Four nametables = 4KB. But the NES only has **2KB of Video RAM**. That's enough for two real nametables; the other two are **mirrors** pointing to the same memory.

Which ones are real and which are mirrors? That depends on the **mirroring** mode we parsed from the cartridge header in chapter 4.

With **vertical mirroring** (what Mario uses), the two real nametables sit side by side:

1		
2	Real A	Real B
3		
4	Real A	Real B
5		

<-- mirrors of above

This gives the PPU a canvas 512 pixels wide (two screens). Mario scrolls horizontally across this canvas. While the player sees nametable A, the game writes the next section of the level into nametable B. When the camera moves to B, the game overwrites A with what comes next. Like a two-panel conveyor belt: you display one while preparing the other. That's how Mario can scroll infinitely with just 2KB of VRAM.

With **horizontal mirroring**, the nametables are stacked (for games that scroll vertically, like Kid Icarus).

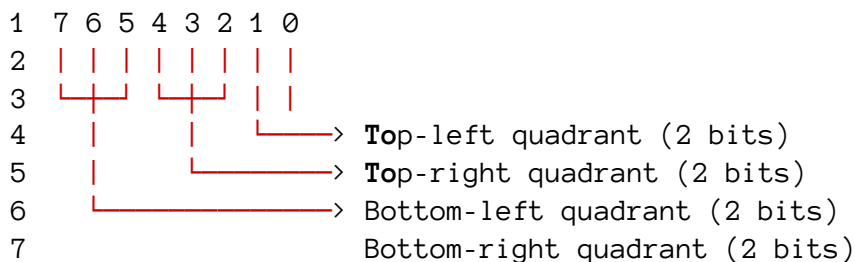
We already covered this in chapter 4. The point is that this mirroring choice now has real consequences for rendering.

Attribute tables: palettes by block

Each tile has 2 bits of color (0-3). But 4 colors for the entire screen would be pretty dull: the sky and the ground in Mario need different colors. The **attribute table** solves this by assigning different **palettes** to different regions of the screen.

The attribute table occupies the last 64 bytes of each nametable (positions 960-1023). Each byte controls a block of **4x4 tiles** (32x32 pixels). Since the screen is 32x30 tiles, there are $8 \times 8 = 64$ blocks. One byte per block.

But here's the twist. Each attribute byte is further subdivided into 4 quadrants of **2x2 tiles** (16x16 pixels). Each quadrant gets 2 bits that select one of the 4 background palettes. The bits are packed into the byte like this:



This means the smallest unit you can change palettes at is a 2x2 tile block (16x16 pixels). You can't give two adjacent tiles different palettes if they're in the same quadrant. It's the harshest visual limitation of the NES, and if you look carefully at any game, you can spot those 16x16 boundaries where the palette switches.

For example, in Mario the sky is blue with white clouds and the ground is brown with blocks. Those wildly different colors don't come from the tiles themselves. The sky and ground tiles could have the exact same pixel values (1, 2, 3). What makes them look different is the palette the attribute table assigns to each region. Palette 0 might be blue/light-blue/white (sky), palette 2 might be brown/orange/beige (ground).

Palettes: the final colors

The PPU stores palettes in the range \$3F00-\$3F1F (32 bytes):

1	\$3F00	Universal background color (backdrop)
2	\$3F01-\$3F03	Background palette 0 (3 colors)
3	\$3F05-\$3F07	Background palette 1 (3 colors)
4	\$3F09-\$3F0B	Background palette 2 (3 colors)
5	\$3F0D-\$3F0F	Background palette 3 (3 colors)
6		
7	\$3F11-\$3F13	Sprite palette 0 (3 colors)
8	\$3F15-\$3F17	Sprite palette 1 (3 colors)
9	\$3F19-\$3F1B	Sprite palette 2 (3 colors)
10	\$3F1D-\$3F1F	Sprite palette 3 (3 colors)

Each palette has 3 colors + color 0 (transparent, which is always the backdrop from \$3F00). Each color is a 6-bit index (0-63) that points to the PPU chip's **master palette**: 64 fixed RGB colors in hardware.

Let's trace the full path from a pixel on screen to an actual color:

1. The PPU reads the nametable: "at position (10, 5) goes tile number \$3C"
2. It reads tile \$3C from the pattern table: pixel (2, 3) within the tile has value 2 (2 bits, from the bitplanes)
3. It reads the attribute table: this block uses **palette 1**
4. It reads palette 1, position 2: the value is **\$16** (a master palette index)
5. It looks up color \$16 in the master palette: **red** (152, 34, 32 in RGB)

Five levels of indirection for a single pixel. Sounds complicated, but it's how the NES gets so much visual variety out of so little memory.

Sprites and OAM: the characters

Everything that moves independently of the background is a **sprite**: Mario, the Goombas, the Koopa Troopas, the spinning coins, the mushrooms, the fireballs. Sprites are drawn *on top of* the background (or behind it, depending on a priority flag).

The PPU can handle up to **64 sprites** at once, with a limit of **8 per scanline** (per horizontal line). If there are more than 8 sprites on the same line, the extras just don't get drawn. That's why enemies flicker in some games when there are too many bunched together: the game rotates which sprites are visible each frame so they all appear intermittently.

Sprite data is stored in the **OAM** (Object Attribute Memory): 256 bytes of internal PPU memory, 4 bytes per sprite:

-
- 1 Byte 0: Y position (vertical coordinate on screen)
 - 2 Byte 1: tile index (which tile from the pattern table to use)
 - 3 Byte 2: attributes (palette, priority, horizontal/vertical flip)
 - 4 Byte 3: X position (horizontal coordinate on screen)
-

Mario standing at position (80, 160) on screen might look like this in OAM:

-
- 1 Sprite 0: Y=159, tile=\$32, attr=\$00, X=80 (upper half of body)
 - 2 Sprite 1: Y=167, tile=\$33, attr=\$00, X=80 (lower half of body)
-

Mario is 8x16 pixels tall (two 8x8 tiles stacked). Each sprite is one tile, so it takes two sprites to draw him completely. And why is Y 159 instead of 160? The NES has a quirk where the Y position in OAM is offset by 1. Y=0 in OAM means "scanline 1", not "scanline 0". So a Goomba at position (120, 200) would have Y=199 in OAM.

The sprite attributes pack several things into one byte:

-
- 1 7 6 5 4 3 2 1 0
 - 2 | | | | | | |
 - 3 | | | | | | | > Sprite palette (0-3)

4 | | | _____> Priority: 0 = in front of background, 1 =
behind
5 | | _____> Flip horizontal
6 | | _____> Flip vertical

Flipping is one of the NES's most elegant tricks. Instead of storing one tile for Mario facing left and another for Mario facing right, you store ONE tile and flip it horizontally with a single bit. Same deal with symmetric sprites or anything that can be inverted: reuse the same tile, flip one bit.

OAM DMA: the fast copy

The OAM lives inside the PPU, and the CPU can only write to it through register \$2004 (OAMDATA), one byte at a time. Updating all 64 sprites that way would take 256 pairs of LDA + STA. Fortunately, the NES has a much better shortcut: **OAM DMA** (Direct Memory Access).

The idea is simple: the game keeps a copy of sprite data in CPU RAM (typically at page \$02, meaning \$0200-\$02FF). During the frame, it updates that copy: “Mario moved to X=90”, “the Goomba dropped to Y=180”, etc. When VBlank arrives (more on that shortly), it transfers all 256 bytes to the OAM in one shot:

```
1 LDA #$02
2 STA $4014      ; Copy $0200-$02FF to the PPU's OAM
```

Those two instructions replace 256 individual writes. The hardware reads all 256 bytes from the specified page and automatically copies them to OAM. The CPU is “frozen” during the transfer (~513 cycles), but it's way faster and more compact than doing it manually.

PPU registers (\$2000-\$2007)

The CPU can't access VRAM directly. To communicate with the PPU, it uses 8 registers mapped at addresses \$2000 through \$2007:

Address	Name	R/W	What Mario uses it for
\$2000	PPUCTRL	W	Configure the PPU at startup: which pattern table for background and sprites, enable NMI for VBlank sync (we'll get to that), choose base nametable
\$2001	PPUMASK	W	Turn rendering on or off: background on/off, sprites on/off. Mario turns them off during level loading and back on when ready
\$2002	PPUSTATUS	R	Check if we're in VBlank (we'll get to that) or if sprite 0 hit occurred (for the HUD trick)
\$2003	OAMADDR	W	OAM address for direct writing (rarely used, OAM DMA is better)
\$2004	OAMDATA	R/W	Read/write a byte of OAM (rarely used, OAM DMA is better)
\$2005	PPUSCROLL	W x2	Set the level's scroll position. Mario writes X=level_position, Y=0 after each frame so the camera follows the player
\$2006	PPUADDR	W x2	Point to a VRAM address. Mario uses it to write the new level tiles that appear when scrolling
\$2007	PPUDATA	R/W	Read or write a byte of VRAM at the address set by \$2006. Each write auto-increments the address

Notice that PPUSCROLL and PPUADDR say "W x2". They need 16 bits of data but the bus is only 8 bits wide, so you write to them twice. The PPU has

an internal latch that tracks whether it's the first or second write. Reading PPUSTATUS resets the latch. We'll dig into this when we implement them.

OceanofPDF.com

Timing: scanlines and cycles

The PPU draws the screen line by line. Each horizontal line is called a **scanline**. A complete frame has **262 scanlines**, but only 240 are visible:

1	Scanlines 0-239:	Visible (the PPU draws pixels)
2	Scanline 240:	Post-render (idle)
3	Scanlines 241-260:	VBlank (the PPU doesn't draw, the game updates VRAM)
4	Scanline 261:	Pre-render (the PPU prepares for the next frame)

Each scanline has **341 PPU cycles** (dots). Of those, only the first 256 produce a visible pixel. The rest are for internal PPU “housekeeping” (tile fetching, sprite evaluation, etc.).

VBlank: the window of opportunity

When the PPU finishes drawing the last visible scanline (239), it enters **VBlank** (Vertical Blank). During VBlank, the PPU doesn't access VRAM for drawing. It's the only safe moment for the game to modify VRAM.

Why? Because during visible scanlines, the PPU is actively reading nametables, pattern tables, and palettes to draw each pixel. If the CPU tries to write to VRAM while the PPU is reading from it, you get glitches: corrupted tiles, wrong colors, visual garbage.

VBlank lasts 20 scanlines (6,600 PPU cycles, roughly 2,200 CPU cycles). In that window, the game needs to:

- Update sprite positions (OAM DMA)
- Write new tiles to nametables (for scrolling)
- Update palettes if they changed
- Set the scroll position

It's a tight window. Games that do heavy visual work (like changing palettes mid-frame for extra colors) have to carefully time every instruction to avoid

running over.

OceanofPDF.com

NMI: the VBlank alarm

So how does the game know VBlank has started? Through an **NMI** (Non-Maskable Interrupt). When the PPU enters VBlank (scanline 241), if bit 7 of PPUCTRL (\$2000) is set, the PPU fires an NMI to the CPU.

The NMI works like the BRK we already implemented, but with differences:

- It reads the handler from **NMI_VECTOR** (\$FFFA) instead of **IRQ_VECTOR** (\$FFFE)
- It doesn't set the B flag in the status it pushes to the stack
- It can't be disabled: normal interrupts (IRQ) can be turned off with the SEI instruction we saw in chapter 3.7. The NMI can't. It fires no matter what (hence "Non-Maskable")

Pretty much every NES game follows the same pattern. Here's how a typical Mario frame works:

```
1 Main loop (runs between NMIs):
2   - Read the controller: did the player press right? jump?
3   - Update logic: move Mario, move Goombas, detect collisions
4   - Calculate the new camera position (scroll X)
5   - Prepare sprites in RAM ($0200-$02FF):
6       Mario at (80, 160), Goomba at (120, 200), coin at (95,
48)...
7   - Wait. Don't touch VRAM. The PPU is drawing.
```

The CPU just spins in a loop, waiting. When the PPU reaches scanline 241, it fires the NMI:

```
1 NMI handler (runs at VBlank start):
2   - OAM DMA: copy the 256 bytes from $0200 to OAM (updated
sprites)
3   - Write to nametables: if Mario moved forward, draw the new
level column
4   - Set scroll: $2005 with the camera's X position
5   - RTI: return to the main loop
```

The CPU calculates, the PPU draws, and the NMI synchronizes them. The entire game is a dance between these two chips, synchronized 60 times per second by the NMI.

OceanofPDF.com

What we're going to build

In the next chapters we'll implement all of this piece by piece:

Chapter 5.1 - Building the PPU

We build the PPU skeleton: registers, VRAM, timing (scanlines and cycles), VBlank, NMI. We add the `nmi` method to the CPU and connect the PPU to the Bus. No rendering yet - the PPU counts cycles and fires the NMI, but doesn't draw anything.

Chapter 5.2 - Opening the window

We hook up an SDL2 window so we can actually see something. The emulator runs internally but the screen is black.

Chapter 5.3 - Rendering the background

We implement background rendering: nametables, pattern tables, bitplanes, attribute tables, palettes. We also implement the automatic scroll increments (loopy). By the end of this chapter you'll see Mario's title screen, but without sprites (Mario, coins, and menu text are all sprites).

Chapter 5.4 - Controls

We implement controllers, map the keyboard, and hook up OAM DMA and sprite 0 hit. There's a bit of a detective story here: the controls work but Mario doesn't respond, because the NMI handler gets stuck waiting for sprite 0 hit. We track down the problem and fix it, and the game starts responding, though we still can't see sprites.

Chapter 5.5 - Sprites

We add sprite rendering and priority logic. Mario shows up, the Goombas show up, the coins spin. By the end of this chapter, you can play Super

Mario Bros.

Let's get started.

OceanofPDF.com

Chapter 5.1: Building the PPU

We have the theory. We know what tiles, nametables, palettes, sprites, registers, and scanline timing are. Now let's write some code.

In this chapter we build the PPU skeleton: registers, VRAM, timing (cycles and scanlines), VBlank, and NMI. We won't draw anything yet, the screen stays black. But by the end of this chapter, the PPU will be counting cycles, firing VBlank on scanline 241, and notifying the CPU via NMI. This is the infrastructure we'll build the rendering on in the next chapters.

OceanofPDF.com

The master palette

Let's start with the simplest thing. The NES has 64 fixed colors hardwired into the PPU chip. They're not configurable, they don't change, they're burned into the silicon. Games pick from these 64 colors to build their palettes.

We need these colors in RGB format so we can later convert them to pixels on screen:

```
1  # src/nes/palette.cr
2
3  module NES
4    PALETTE = [
5      # $00-$0F: Dark
6      [84, 84, 84], [0, 30, 116], [8, 16, 144], [48, 0, 136],
7      [68, 0, 100], [92, 0, 48], [84, 4, 0], [60, 24, 0],
8      [32, 42, 0], [8, 58, 0], [0, 64, 0], [0, 60, 0],
9      [0, 50, 60], [0, 0, 0], [0, 0, 0], [0, 0, 0],
10
11     # $10-$1F: Medium
12     [152, 150, 152], [8, 76, 196], [48, 50, 236], [92, 30,
13 228],
14     [136, 20, 176], [160, 20, 100], [152, 34, 32], [120, 60,
15 0],
16     [84, 90, 0], [40, 114, 0], [8, 124, 0], [0, 118,
17 40],
18     [0, 102, 120], [0, 0, 0], [0, 0, 0], [0, 0, 0],
19
20     # $20-$2F: Bright
21     [236, 238, 236], [76, 154, 236], [120, 124, 236], [176, 98,
22 236],
23     [228, 84, 236], [236, 88, 180], [236, 106, 100], [212, 136,
24 32],
25     [160, 170, 0], [116, 196, 0], [76, 208, 32], [56, 204,
26 108],
27     [56, 180, 204], [60, 60, 60], [0, 0, 0], [0, 0, 0],
28
29     # $30-$3F: Pastel
30     [236, 238, 236], [168, 204, 236], [188, 188, 236], [212, 178,
31 236],
32     [236, 174, 236], [236, 174, 212], [236, 180, 176], [228, 196,
```

```
144],  
26      [204, 210, 120], [180, 222, 120], [168, 226, 144], [152, 226,  
180],  
27      [160, 214, 228], [160, 162, 160], [0, 0, 0],      [0, 0, 0],  
28      ]  
29  end
```

An array of 64 entries, each with [R, G, B]. When the PPU needs to draw color \$16, it looks up PALETTE[0x16] and gets [152, 34, 32], a dark red. These values come from the NES Wiki and are the standard palette most emulators use.

OceanofPDF.com

The PPU skeleton

The PPU is organized the same way as the CPU: a main class that includes specialized modules. Each module handles a different responsibility:

```
1 PPU
2   includes Addresses      # PPU memory map constants
3   includes Display        # Screen dimensions and scanline timing
4   includes Loopy          # Internal registers v, t, fine_x, w
5   includes Registers      # The 8 registers ($2000-$2007)
6   includes Ctrl
7   includes Mask
8   includes Status
9   includes Scroll
10  includes Addr
11  includes Data
12  includes OAMAddr
13  includes OAMData
14  includes VRAM           # Memory routing (pattern tables,
                             nametables, palettes)
```

Same pattern we used for the CPU and its instruction modules. Same advantage too: each register lives in its own file, easy to find and easy to read. The main class stays clean.

Let's look at `ppu.cr`:

```
1 # src/nes/ppu.cr
2
3 require "../ppu/addresses"
4 require "../ppu/display"
5 require "../ppu/loopy"
6 require "../ppu/registers"
7 require "../ppu/vram"
8
9 module NES
10   class PPU
11     include Addresses
12     include Display
13     include Loopy
```



```
14     include Registers
15     include VRAM
```

The requires and includes. Now the instance variables, grouped by function:

```
1     # Timing
2     getter cycle      : Int32
3     getter scanline  : Int32
4
5     # Frame output
6     getter frame_buffer : Bytes
7     getter frame_ready : Bool
8
9     # Registers ($2000-$2007)
10    getter ctrl       : UInt8
11    getter mask       : UInt8
12    getter status     : UInt8
13    getter oam_addr   : UInt8
14
15    # Loopy scroll registers
16    getter v          : UInt16
17    getter t          : UInt16
18    getter fine_x     : UInt8
19    getter w          : Bool
20
21    # Internal
22    getter vram_buffer : UInt8
23
24    # Memory
25    getter nametable_ram : Bytes
26    getter palette_ram   : Bytes
27    getter oam            : Bytes
28
29    property nmi_callback : Proc(Nil)?
```

That’s a lot of variables, but each one has a clear purpose. The registers `v`, `t`, `fine_x` and `w` are the famous “loopy registers”, the internal registers shared by PPUSCROLL and PPUADDR. We’ll look at them in detail in a moment.

`nmi_callback` is a `Proc(Nil)?`. In Crystal (same as Ruby), you can store a block of code in a variable. `Proc(Nil)` is a block that returns nothing, and the `?` means it can be nil (nobody assigns it at first). The Emulator will

assign a block that calls `@cpu.nmi`, so the PPU can notify the CPU when VBlank starts. It's property (not getter) because we need to write it from the outside.

The constructor initializes everything to zero:

```
1  def initialize(@cartridge : Cartridge)
2    @nmi_callback = nil
3    @cycle         = 0
4    @scanline      = 0
5    @frame_ready   = false
6    @frame_buffer  = Bytes.new(WIDTH * HEIGHT * 4, 0_u8)
7    @ctrl          = 0_u8
8    @mask          = 0_u8
9    @status        = 0_u8
10   @oam_addr       = 0_u8
11   @v              = 0_u16
12   @t              = 0_u16
13   @fine_x         = 0_u8
14   @w              = false
15   @vram_buffer    = 0_u8
16   @nametable_ram  = Bytes.new(NAMETABLE_RAM_SIZE, 0_u8)
17   @palette_ram    = Bytes.new(PALETTE_RAM_SIZE, 0_u8)
18   @oam            = Bytes.new(OAM_SIZE, 0_u8)
19  end
```

`@frame_buffer` is `WIDTH * HEIGHT * 4` bytes: 256 x 240 pixels, 4 bytes each (RGBA). This is the buffer we'll later pass to SDL2 (the library we'll use to open a window and draw to the screen) in the next chapter. For now it stays filled with zeros.

The PPU receives the `Cartridge` because it needs access to the CHR-ROM to read tiles and to the mirroring mode to resolve nametables.

Constants: addresses and display

Before diving into the registers, let's define the constants. Two short modules.

The PPU memory map

The PPU has its own 16KB address space, separate from the CPU's:

```
1  # src/nnes/ppu/addresses.cr
2
3  module NES
4    class PPU
5      module Addresses
6        PATTERN_TABLE_START = 0x0000_u16
7        PATTERN_TABLE_END   = 0x1FFF_u16
8        NAMETABLE_START     = 0x2000_u16
9        NAMETABLE_MIRROR_END = 0x3EFF_u16
10       PALETTE_START        = 0x3F00_u16
11       VRAM_MASK            = 0x3FFF_u16
12
13       NAMETABLE_RAM_SIZE   = 2048
14       PALETTE_RAM_SIZE    = 32
15       OAM_SIZE             = 256
16
17       NAMETABLE_SIZE       = 1024_u16
18       NAMETABLE_AREA_MASK = 0x0FFF_u16
19
20       PALETTE_MASK         = 0x1F_u16
21       PALETTE_SPRITE_BASE  = 0x10_u16
22       PALETTE_COLOR_MASK  = 0x03_u16
23       COLOR_MASK          = 0x3F_u8
24     end
25   end
26 end
```

The map is the one we saw in chapter 5.0:

- \$0000-\$1FFF: pattern tables (live in the cartridge's CHR-ROM)
- \$2000-\$3EFF: nametables (2KB of internal VRAM, with mirroring)

- \$3F00-\$3FFF: palettes (32 internal bytes)

VRAM_MASK (\$3FFF) wraps any address into the 16KB range. The palette constants will be used for the mirroring between sprite and background palettes.

Dimensions and timing

```
1  # src/nes/ppu/display.cr
2
3  module NES
4    class PPU
5      module Display
6        WIDTH = 256
7        HEIGHT = 240
8
9        CYCLES_PER_SCANLINE = 341
10       SCANLINES_PER_FRAME = 262
11       VBLANK_SCANLINE     = 241
12       PRE_RENDER_SCANLINE = 261
13     end
14   end
15 end
```

This comes straight from the theory: 341 cycles per scanline, 262 scanlines per frame, VBlank starts at scanline 241, and scanline 261 is the pre-render (where the PPU prepares for the next frame).

The loopy registers

This is the densest section of the chapter. Stick with me though, because it's the most important part for understanding how the PPU works internally.

As we saw in chapter 5.0, PPUSCROLL (\$2005) and PPUADDR (\$2006) each need two writes to set a complete value through an 8-bit bus. Internally, the PPU doesn't have two separate "scroll" and "address" variables. It has a **single set of registers** that serves both purposes:

- **v** (15 bits): the "current" register. During rendering it's the scroll position; during VBlank it's the VRAM pointer.
- **t** (15 bits): the "temporary" register. Accumulates the CPU's partial writes. Think of it as a draft: the CPU writes here, and when it's done, **t** gets copied to **v** to activate the value.
- **fine_x** (3 bits): the pixel offset within the first visible tile. It lives separately because it doesn't fit in the 15 bits of **v/t**. I'll explain why in a moment.
- **w** (1 bit): the write latch. A latch is a single bit that remembers state. In this case, it tracks whether the next write to SCROLL or ADDR is the first or the second. It starts at 0 ("first write"), and after writing it flips to 1 ("second write"). Reading PPUSTATUS resets it to 0.

Why do scroll and address share registers? Because the scroll position IS a VRAM address. If the camera points at tile (5, 3) of nametable 1, the PPU needs to read that tile's data from VRAM, which is at the address corresponding to nametable 1, row 3, column 5.

The layout of **v** and **t** is:

1	yyy NN YYYYY XXXXX	
2		coarse X scroll (0-31 tiles)
3	LLLLL	coarse Y scroll (0-29 tiles)
4	LL	nametable select (0-3)
5	LLL	fine Y scroll (0-7 pixels)

15 bits packed together. Let's see what each part means with a concrete example.

Suppose Mario has moved 43 pixels in World 1-1. The game writes 43 to PPUSCROLL. The PPU decomposes that value into:

- **coarse X = 5**: the camera starts at **tile number 5** of the nametable (counting from the left: tile 0, 1, 2, 3, 4, 5). On Mario's screen, the first 5 tiles of sky/ground have already scrolled off to the left.
- **fine X = 3**: within that tile 5, we don't start from the left edge but from **pixel 3** (out of the tile's 8 pixels). This is the fine adjustment that gives smooth pixel-by-pixel scrolling.

The math: a tile is 8 pixels wide. 43 pixels of scroll = 5 full tiles (40 pixels) + 3 extra pixels into the sixth tile.

Now, `fine_x` is stored in a separate variable (`@fine_x`), while `fine_y` is inside the 15 bits of `@t`. Why the difference? Because the PPU uses `v` as a VRAM address to fetch tiles. To know which **row** of the tile to draw, it needs fine Y as part of the address (each row of a tile is a different byte in the pattern table). But fine X doesn't affect which bytes to fetch from VRAM. The full tile byte (8 pixels) has already been read, and fine X simply says "start reading from pixel 3 instead of 0". It's a purely visual offset applied after the read, not during it.

Think about it in Mario: if the camera is 3 pixels to the right within a tile, the PPU reads the full tile from VRAM and then discards the first 3 pixels. It doesn't need fine X to calculate the VRAM address. But if the camera is 3 pixels down within a tile, the PPU needs to know it should read row 3 of the tile (not row 0), and that *does* affect the address.

Let's look at `loopy.cr`:

```
1 # src/nes/ppu/loopy.cr
2
3 module NES
4   class PPU
5     module Loopy
6       MASK_COARSE_X = 0x001F_u16
7       MASK_COARSE_Y = 0x03E0_u16
8       MASK_NAMETABLE = 0x0C00_u16
```

```
9      MASK_FINE_Y      = 0x7000_u16
10     CLEAR_NAMETABLE = 0xF3FF_u16
11     CLEAR_COARSE_X   = 0xFFE0_u16
12     CLEAR_ALL_Y      = 0x0C1F_u16
```

The MASK_* constants extract bits (you AND to read the value). The CLEAR_* constants erase bits (you AND to clear before writing a new value). CLEAR_ALL_Y is the weird one: it keeps only the nametable and coarse X bits, erasing everything Y-related (coarse Y + fine Y). We'll need it when PPUSCROLL writes the Y scroll.

The latch helpers:

```
1      def latch_on? : Bool
2          @w
3      end
4
5      def reset_latch
6          @w = false
7      end
8
9      def toggle_latch
10         @w = !@w
11     end
```

latch_on? returns true if the first write has already happened (and the next one will be the second). toggle_latch alternates between first and second write. reset_latch goes back to zero.

The v increment is for when you read or write PPUDATA:

```
1      def increment_v
2          @v = (@v &+ vram_increment) & VRAM_MASK
3      end
```

vram_increment returns 1 or 32 depending on bit 2 of PPUCTRL (horizontal or vertical increment). The & VRAM_MASK makes sure v doesn't exceed 14 bits.

And the setters that write to @t:

```
1      def set_coarse_x(coarse : UInt16)
2          @t = (@t & CLEAR_COARSE_X) | coarse
3      end
4
5      def set_nametable(nametable : UInt16)
6          @t = (@t & CLEAR_NAMETABLE) | (nametable << 10)
7      end
8
9      def set_scroll_y(fine : UInt16, coarse : UInt16)
10         @t = (@t & CLEAR_ALL_Y) | (fine << 12) | (coarse << 5)
11     end
12 end
13 end
14 end
```

Every setter follows the same pattern: clear the old bits with CLEAR_*, position the new bits with a shift, and combine with OR. set_coarse_x puts coarse X in bits 4-0, set_nametable puts the nametable selection in bits 11-10, and set_scroll_y puts fine Y in bits 14-12 and coarse Y in bits 9-5.

These setters separate the “bit surgery” (positioning data in the right place of a 15-bit register) from the code that calls them. In the SCROLL, CTRL, and ADDR registers we’ll just extract the data and call the right setter, without worrying about masks.

The PPU registers

Alright, now the good stuff: the 8 registers the CPU sees at \$2000-\$2007. Each one lives in its own module, and an umbrella Registers includes them all:

```
1  # src/nes/ppu/registers.cr
2
3  require "../registers/ctrl"
4  require "../registers/mask"
5  require "../registers/status"
6  require "../registers/oam_addr"
7  require "../registers/oam_data"
8  require "../registers/scroll"
9  require "../registers/addr"
10 require "../registers/data"
11
12 module NES
13   class PPU
14     module Registers
15       include Ctrl
16       include Mask
17       include Status
18       include OAMAddr
19       include OAMData
20       include Scroll
21       include Addr
22       include Data
23     end
24   end
25 end
```

Same pattern as CPU : : Instructions. Let's go through each register.

PPUCTRL (\$2000): Configuration

```
1  # src/nes/ppu/registers/ctrl.cr
2
3  module NES
4    class PPU
```

```

5     module Registers
6         module Ctrl
7             FLAG_VRAM_INCREMENT = 0x04_u8
8             FLAG_NMI_ENABLED    = 0x80_u8
9
10            def write_ctrl(value : UInt8)
11                @ctrl = value
12                nametable = value.to_u16 & 0x03
13                set_nametable(nametable)
14            end
15
16            def vram_increment : UInt16
17                (@ctrl & FLAG_VRAM_INCREMENT) != 0 ? 32_u16 : 1_u16
18            end
19
20            def nmi_enabled? : Bool
21                (@ctrl & FLAG_NMI_ENABLED) != 0
22            end
23        end
24    end
25 end
26 end

```

PPUCTRL configures the PPU: which pattern table to use for sprites and background, the VRAM increment, whether to generate NMI on VBlank, and which base nametable to use. The bits are:

```

1 bit 0-1: base nametable (0=$2000, 1=$2400, 2=$2800, 3=$2C00)
2 bit 2:   VRAM increment (0=+1 horizontal, 1=+32 vertical)
3 bit 3:   sprite pattern table (0=$0000, 1=$1000)
4 bit 4:   background pattern table (0=$0000, 1=$1000)
5 bit 5:   sprite size (0=8x8, 1=8x16)
6 bit 6:   master/slave (not used on NES)
7 bit 7:   generate NMI on VBlank

```

The most important thing about `write_ctrl` is that bits 0-1 (nametable selection) go straight to the `@t` register via `set_nametable`. This is part of the loopy mechanism: when Mario says “use nametable 1”, it’s setting bits 11-10 of `@t`.

`vram_increment` returns 1 (advance horizontally) or 32 (advance vertically, going down one row in the nametable). Mario uses increment 1

when writing the nametable column by column while scrolling.

PPUMASK (\$2001): What gets rendered

```
1 # src/nes/ppu/registers/mask.cr
2
3 module NES
4   class PPU
5     module Registers
6       module Mask
7         def write_mask(value : UInt8)
8           @mask = value
9         end
10      end
11    end
12  end
13 end
```

PPUMASK controls what the PPU draws: background on/off, sprites on/off, whether to clip the first 8 pixels, etc. For now we just store the value. When we implement rendering we'll check individual bits to decide what to draw.

PPUSTATUS (\$2002): VBlank and side effects

```
1 # src/nes/ppu/registers/status.cr
2
3 module NES
4   class PPU
5     module Registers
6       module Status
7         FLAG_VBLANK = 0x80_u8
8
9         def read_status : UInt8
10           status_snapshot = @status
11
12           clear_vblank
13           reset_latch
14
15           status_snapshot
16         end
17       end
18     end
19   end
20 end
```

```
18         def set_vblank
19             @status |= FLAG_VBLANK
20         end
21
22         def clear_vblank
23             @status &= (~FLAG_VBLANK).to_u8
24         end
25     end
26 end
27 end
28 end
```

PPUSTATUS has 3 meaningful bits: VBlank (bit 7), Sprite 0 Hit (bit 6), and Sprite Overflow (bit 5). But the interesting thing about this register isn't the data itself, it's the side effects of reading it.

Just reading PPUSTATUS does two extra things: it clears the VBlank bit and resets the write latch. That's why `read_status` first saves a copy of the current value (`status_snapshot`), then clears and resets, and finally returns the copy. The game receives the value *before* the side effects.

Games read PPUSTATUS before writing to SCROLL or ADDR to make sure the latch is reset to "first write". You'll see this pattern in virtually every ROM:

```
1 LDA $2002      ; read status => reset latch
2 LDA #$20
3 STA $2006      ; first write (high byte)
4 LDA #$00
5 STA $2006      ; second write (low byte)
```

`set_vblank` and `clear_vblank` are helpers used by `advance_cycle` (we'll get there) to turn the flag on and off. The OR turns on bit 7, the AND with the complement (`~`) turns it off.

PPUSCROLL (\$2005): The camera position

```
1 # src/nes/ppu/registers/scroll.cr
2
3 module NES
```

```

4  class PPU
5      module Registers
6          module Scroll
7              FINE_BITS = 0x07_u8
8
9              def write_scroll(value : UInt8)
10                 if latch_on?
11                     write_scroll_y(value)
12                 else
13                     write_scroll_x(value)
14                 end
15
16                 toggle_latch
17             end
18
19             def write_scroll_x(value : UInt8)
20                 @fine_x = value & FINE_BITS
21                 coarse = value.to_u16 >> 3
22                 set_coarse_x(coarse)
23             end
24
25             def write_scroll_y(value : UInt8)
26                 fine    = value.to_u16 & FINE_BITS
27                 coarse = value.to_u16 >> 3
28                 set_scroll_y(fine, coarse)
29             end
30         end
31     end
32 end
33 end

```

This is one of the most interesting registers. The CPU writes the scroll position in pixels, and the PPU splits it into two parts: the tile (coarse) and the pixel within the tile (fine).

Let's walk through a concrete example. Mario keeps a variable with the camera's scroll position. Say the camera has moved 203 pixels. The game writes 203 to PPUSCROLL.

The PPU breaks 203 down like this:

- **coarse** = $203 \gg 3 = 25$. The camera starts at tile number 25 of the nametable (the first 25 tiles of the level have already scrolled off to the left).

- **fine** = 203 & 0x07 = 3. Within tile 25, we don't start at the left edge but at pixel 3. This is what gives smooth pixel-by-pixel scrolling. Without fine, the camera would jump 8 pixels at a time (one whole tile) and it would look choppy.

The first write (latch at 0) is the X scroll: fine goes to @fine_x and coarse goes to @t via set_coarse_x. The second write (latch at 1) is the Y scroll: both go to @t via set_scroll_y, because fine Y does fit in the 15 bits of the loopy register (unlike fine X which is stored separately, as we covered earlier).

PPUADDR (\$2006): VRAM address

```
1  # src/nes/ppu/registers/addr.cr
2
3  module NES
4    class PPU
5      module Registers
6        module Addr
7          def write_addr(value : UInt8)
8            if latch_on?
9              @t = (@t & 0xFF00_u16) | value.to_u16
10             @v = @t
11            else
12              @t = (@t & 0x00FF_u16) | ((value.to_u16 & 0x3F) << 8)
13            end
14
15            toggle_latch
16          end
17        end
18      end
19    end
20  end
```

PPUADDR needs two writes to assemble a 14-bit address (the PPU's address space is 16KB). The first write puts the high 6 bits into @t, the second puts the low 8 bits. Only on the second write does @t get copied to @v, which is the active address used by PPUDATA.

Why & 0x3F on the first write? Because the VRAM address is only 14 bits (maximum \$3FFF). The top 2 bits of the byte are ignored.

Notice that PPUADDR writes to @t, the same temporary register as PPUSCROLL. If a game writes to PPUADDR, it clobbers whatever scroll bits were in @t. That's just how the hardware works: SCROLL and ADDR share the loopy registers.

PPUDATA (\$2007): VRAM access

```
1  # src/nes/ppu/registers/data.cr
2
3  module NES
4    class PPU
5      module Registers
6        module Data
7          def read_data : UInt8
8            if @v >= PALETTE_START
9              result = vram_read(@v)
10             @vram_buffer = vram_read(@v &- 0x1000_u16)
11           else
12             result = @vram_buffer
13             @vram_buffer = vram_read(@v)
14           end
15
16           increment_v
17           result
18         end
19
20         def write_data(value : UInt8)
21           vram_write(@v, value)
22           increment_v
23         end
24       end
25     end
26   end
27 end
```

PPUDATA is the CPU's only way to read or write PPU memory. Set the address with PPUADDR, then read or write through this register. Each access auto-increments @v (via increment_v), so you can write consecutive bytes without touching PPUADDR again.

Reading has a hardware quirk: the data doesn't arrive immediately. The PPU stashes the read byte into an internal buffer (@vram_buffer) and hands

you back whatever was *already* in the buffer. So the first read gives you garbage (the buffer was empty), and the second gives you the value you actually wanted. Games do a “dummy read” that they throw away.

The exception is palettes (\$3F00+): they’re read immediately, bypassing the buffer. But the buffer still gets updated with the nametable data that sits “behind” the palette address (@v - \$1000). It’s a hardware quirk that some games actually take advantage of.

OAMADDR (\$2003) and OAMDATA (\$2004): Sprites

```
1 # src/nes/ppu/registers/oam_addr.cr
2
3 module NES
4   class PPU
5     module Registers
6       module OAMAddr
7         def write_oam_addr(value : UInt8)
8           @oam_addr = value
9         end
10      end
11    end
12  end
13 end
```

```
1 # src/nes/ppu/registers/oam_data.cr
2
3 module NES
4   class PPU
5     module Registers
6       module OAMData
7         def read_oam_data : UInt8
8           @oam[@oam_addr]
9         end
10
11        def write_oam_data(value : UInt8)
12          @oam[@oam_addr] = value
13          @oam_addr &+= 1_u8
14        end
15      end
16    end
17  end
```



```
17   end
18 end
```

These are straightforward. OAMADDR sets a pointer inside the OAM (256 bytes of sprite memory). OAMDATA reads or writes at that position. Writing auto-increments the address, so you can write consecutive bytes. In practice almost nobody uses these registers directly. Everyone uses OAM DMA (\$4014), which copies all 256 bytes at once. We'll implement that in a later chapter.

The register router

Back in `ppu.cr`, the methods that connect addresses \$2000-\$2007 to each register's methods:

```
1   def read_register(address : UInt16) : UInt8
2     case address & 0x07
3     when 2 then read_status
4     when 4 then read_oam_data
5     when 7 then read_data
6     else 0_u8
7     end
8   end
9
10  def write_register(address : UInt16, value : UInt8)
11    case address & 0x07
12    when 0 then write_ctrl(value)
13    when 1 then write_mask(value)
14    when 3 then write_oam_addr(value)
15    when 4 then write_oam_data(value)
16    when 5 then write_scroll(value)
17    when 6 then write_addr(value)
18    when 7 then write_data(value)
19    end
20  end
```

`address & 0x07` strips everything but the low 3 bits. This handles mirroring: the registers repeat every 8 bytes across the entire \$2000-\$3FFF range, so \$2000 and \$2008 and \$3FF8 are all PPUCTRL.

Only 3 registers are readable: PPUSTATUS, OAMDATA, and PPUDATA.
The rest return 0 when read (on real hardware they return the last value on the data bus, but 0 works fine for our emulator).

OceanofPDF.com

VRAM: the memory router

The PPU has its own memory map and needs a router, just like the CPU's Bus. `vram.cr` resolves which chip responds to each address:

```
1  # src/nes/ppu/vram.cr
2
3  module NES
4    class PPU
5      module VRAM
6        def vram_read(address : UInt16) : UInt8
7          addr = address & VRAM_MASK
8
9          case addr
10         when PATTERN_TABLE_START..PATTERN_TABLE_END
11           @cartridge.mapper.chr_read(addr)
12         when NAMETABLE_START..NAMETABLE_MIRROR_END
13           @nametable_ram[mirror_nametable_addr(addr)]
14         when PALETTE_START..VRAM_MASK
15           @palette_ram[palette_index(addr)]
16         else
17           0_u8
18         end
19       end
20
21       def vram_write(address : UInt16, value : UInt8)
22         addr = address & VRAM_MASK
23
24         case addr
25         when PATTERN_TABLE_START..PATTERN_TABLE_END
26           @cartridge.mapper.chr_write(addr, value)
27         when NAMETABLE_START..NAMETABLE_MIRROR_END
28           @nametable_ram[mirror_nametable_addr(addr)] = value
29         when PALETTE_START..VRAM_MASK
30           @palette_ram[palette_index(addr)] = value & COLOR_MASK
31         end
32       end

```

Three ranges, three destinations:

- **Pattern tables** go to the cartridge's CHR-ROM (through the mapper).

- **Nametables** go to the 2KB of internal VRAM, but pass through `mirror_nametable_addr` to resolve mirroring.
- **Palettes** go to the 32 bytes of palette RAM. The `& COLOR_MASK` on writes ensures the value is a valid index from 0-63.

Nametable mirroring

The NES has 4 logical nametables but only 2KB of VRAM (enough for 2). The cartridge's mirroring decides which ones are duplicated:

```

1      def mirror_nametable_addr(address : UInt16) : UInt16
2          relative = (address &- NAMETABLE_START) &
NAMETABLE_AREA_MASK
3          nametable = relative // NAMETABLE_SIZE
4          offset = relative % NAMETABLE_SIZE
5
6          physical = case @cartridge.mirroring
7              when Cartridge::Mirroring::Vertical then nametable &
1
8              when Cartridge::Mirroring::Horizontal then nametable >>
1
9              else nametable & 1
10             end
11
12         physical * NAMETABLE_SIZE + offset
13     end

```

First it calculates which logical nametable (0-3) and what offset within it. Then it maps the logical nametable to a physical one (0 or 1) based on the mirroring mode:

- **Vertical** (Mario): nametables 0 and 2 are the same, 1 and 3 are the same. Pattern: 0,1,0,1 (`& 1`). This gives two screens side by side, useful for horizontal scrolling. While Mario is looking at nametable 0, the game prepares nametable 1 with the next section of the level.
- **Horizontal** (Kid Icarus): nametables 0 and 1 are the same, 2 and 3 are the same. Pattern: 0,0,1,1 (`>> 1`). This gives two screens stacked, useful for vertical scrolling.

Palette mirroring

```
1      def palette_index(address : UInt16) : UInt16
2          index = address & PALETTE_MASK
3          index -= PALETTE_SPRITE_BASE if index >=
PALETTE_SPRITE_BASE && (index & PALETTE_COLOR_MASK) == 0
4          index
5      end
6  end
7  end
8  end
```

Sprite palettes start at \$3F10. But color 0 of each sprite palette (\$3F10, \$3F14, \$3F18, \$3F1C) is actually a mirror of the universal background color (\$3F00, \$3F04, \$3F08, \$3F0C). It's a hardware quirk: those 4 sprite addresses don't store their own value, they reflect the background ones.

The condition (index & PALETTE_COLOR_MASK) == 0 checks if it's color 0 of a palette (positions 0, 4, 8, 12, 16, 20, 24, 28). If it's in the sprite range (>= \$10) and it's color 0, it subtracts \$10 to redirect it to the background equivalent.

Timing and VBlank

We have registers and VRAM. The missing piece is the engine that makes everything tick: cycle and scanline counting.

```
1  def step
2      advance_cycle
3  end
4
5  def frame_ready? : Bool
6      @frame_ready
7  end
8
9  def clear_frame_ready
10     @frame_ready = false
11 end
12
13 def advance_cycle
14     @cycle += 1
15
16     if @cycle >= CYCLES_PER_SCANLINE
17         @cycle = 0
18         @scanline += 1
19
20         if @scanline == VBLANK_SCANLINE
21             set_vblank
22             @frame_ready = true
23             trigger_nmi if nmi_enabled?
24         elsif @scanline == PRE_RENDER_SCANLINE
25             clear_vblank
26         elsif @scanline >= SCANLINES_PER_FRAME
27             @scanline = 0
28         end
29     end
30 end
31
32 def trigger_nmi
33     @nmi_callback.not_nil!.call
34 end
```

Each step advances one cycle. Every 341 cycles a scanline ends. When the scanline reaches 241 (VBlank):

1. The VBlank flag is set in PPUSTATUS
2. The frame is marked as ready (@frame_ready = true)
3. If NMI is enabled in PPUCTRL, the callback fires

On scanline 261 (pre-render) VBlank is cleared, preparing the PPU for the next frame. After scanline 261 we go back to 0 and it all starts over.

trigger_nmi uses .not_nil!.call. Remember that @nmi_callback is a Proc(Nil)? (it can be nil). .not_nil! tells Crystal “I’m sure this isn’t nil, blow up if it is”. This is intentional: if the PPU tries to fire an NMI but nobody connected the callback, that’s a bug in our code and we want to catch it early.

OceanofPDF.com

NMI in the CPU

The PPU fires the NMI, but the CPU needs to know what to do with it. We need to add the `nmi` method to the CPU's interrupts module:

```
1  # src/nes/cpu/interrupts.cr (adding the nmi method)
2
3      def nmi
4          stack_push_word(@pc)
5          stack_push_byte((@status | FLAG_U) & (~FLAG_B).to_u8)
6          set_flag(FLAG_I, true)
7
8          lo = read_byte(NMI_VECTOR)
9          hi = read_byte(NMI_VECTOR &+ 1)
10         @pc = (hi.to_u16 << 8) | lo.to_u16
11     end
```

It's nearly identical to the BRK we implemented in chapter 3.6, with two differences:

- It reads the handler address from `NMI_VECTOR` (\$FFFA) instead of `IRQ_VECTOR` (\$FFFE)
- It doesn't set the B flag in the status byte pushed to the stack (BRK sets it so the handler can tell BRK and IRQ apart)

The constants `NMI_VECTOR`, `RESET_VECTOR`, and `IRQ_VECTOR` were already defined in this same file back in the interrupts chapter. When Mario is running and the PPU reaches scanline 241, the NMI redirects the CPU to the handler the game set up at \$FFFA. That's where Mario does the OAM DMA, writes new tiles to the nametable, and sets the scroll.

Integration into Bus and Emulator

Time to hook everything up.

Bus: new constants

We add the PPU range constants to the Bus's address map:

```
1 # src/nes/bus/addresses.cr (adding the PPU section)
2
3 # ===== PPU Registers =====
4 PPU_START      = 0x2000_u16
5 PPU_END        = 0x2007_u16
6 PPU_MIRROR_END = 0x3FFF_u16
```

Bus: routing

The Bus now receives the PPU and routes the \$2000-\$3FFF range:

```
1 # src/nes/bus.cr
2
3 module NES
4   class Bus
5     include Addresses
6
7     def initialize(cartridge : Cartridge, ppu : PPU)
8       @cartridge = cartridge
9       @ppu = ppu
10      @ram = Bytes.new(RAM_SIZE, 0_u8)
11    end
12
13    def cpu_read(address : UInt16) : UInt8
14      case address
15      when RAM_START..RAM_MIRROR_END
16        read_ram(address)
17      when PPU_START..PPU_MIRROR_END
18        @ppu.read_register(address)
19      when PRG_ROM_START..PRG_ROM_END
20        @cartridge.mapper.prg_read(address)
```

```

21     else
22         0_u8
23     end
24 end
25
26 def cpu_write(address : UInt16, value : UInt8)
27     case address
28     when RAM_START..RAM_MIRROR_END
29         write_ram(address, value)
30     when PPU_START..PPU_MIRROR_END
31         @ppu.write_register(address, value)
32     end
33 end

```

Two new lines in the case: when PPU_START..PPU_MIRROR_END delegates to @ppu.read_register and @ppu.write_register. The mirroring (every 8 bytes) is handled by the & 0x07 inside the PPU.

Emulator: PPU + NMI + step

The Emulator gets rewritten to create the PPU, wire up the NMI, and run the PPU 3 times per CPU cycle:

```

1  # src/nes/emulator.cr
2
3  module NES
4      class Emulator
5          include Debugging
6
7          getter cartridge : Cartridge
8          getter cpu       : CPU
9          getter bus       : Bus
10         getter ppu       : PPU
11
12         def initialize(path : String)
13             @cartridge = Cartridge.new(path)
14             @ppu = PPU.new(@cartridge)
15             @bus = Bus.new(@cartridge, @ppu)
16             @cpu = CPU.new(@bus)
17
18             @ppu.nmi_callback = ->{ @cpu.nmi }
19

```

```

20     @cpu.reset
21 end
22
23 def step
24     cycles = @cpu.step
25     (cycles * 3).times { @ppu.step }
26     cycles
27 end
28
29 def run_frame : Bytes
30     @ppu.clear_frame_ready if @ppu.frame_ready?
31
32     until @ppu.frame_ready?
33         step
34     end
35
36     @ppu.frame_buffer
37 end
38 end
39 end

```

The creation order matters: first the cartridge, then the PPU (which needs the cartridge), then the Bus (which needs both), and finally the CPU (which needs the Bus).

The NMI connection is one line: `@ppu.nmi_callback = ->{ @cpu.nmi }`. In Crystal, `->{ ... }` creates a block of code that can be stored in a variable and executed later. This assigns to `nmi_callback` a block that calls `@cpu.nmi`. When the PPU reaches scanline 241, it executes that block, which calls `@cpu.nmi`, which pushes PC and status to the stack and jumps to the game's handler.

`step` executes one CPU instruction and then advances the PPU 3 times per cycle. Remember the PPU runs at 3x the CPU's speed. If a CPU instruction takes 4 cycles, the PPU advances 12 steps.

`run_frame` keeps calling `step` until the PPU marks a frame as ready (i.e., until it reaches VBlank). It returns the frame buffer, which for now is all black.

The require in nes.cr

```
1 # src/nes.cr (adding the new requires)
2
3 require "./nes/emulator"
4 require "./nes/cartridge"
5 require "./nes/palette"
6 require "./nes/ppu"
```

[OceanofPDF.com](https://oceanofpdf.com)

Tests

To verify everything works, we have 10 tests in `bin/test_ppu.cr`. Here are the most interesting ones.

VBlank timing

```
1 cart = NES::Cartridge.new(".roms/mario.nes")
2 ppu = NES::PPU.new(cart)
3
4 # 241 scanlines * 341 cycles - 1 = just before VBlank
5 (241 * 341 - 1).times { ppu.step }
6
7 passed = (ppu.status & 0x80_u8) == 0 && !ppu.frame_ready?
8 # VBlank hasn't triggered yet
9
10 ppu.step
11 # One more step and it triggers
12
13 passed = (ppu.status & 0x80_u8) != 0 && ppu.frame_ready?
14 # VBlank active, frame ready
```

We advance the PPU to the very last cycle before VBlank. The flag should be off. One more step and VBlank triggers.

PPUSCROLL and the loopy registers

```
1 ppu4 = NES::PPU.new(cart)
2
3 # Scroll X = 43 (coarse X = 5, fine X = 3)
4 ppu4.write_register(0x2005_u16, 43_u8)
5
6 passed = ppu4.fine_x == 3_u8
7 passed = (ppu4.t & 0x001F_u16) == 5_u16
8
9 # Scroll Y = 43 (coarse Y = 5, fine Y = 3)
10 ppu4.write_register(0x2005_u16, 43_u8)
11
```

```
12 coarse_y = (ppu4.t >> 5) & 0x1F
13 fine_y = (ppu4.t >> 12) & 0x07
```

Value 43 = tile 5 (43 >> 3) + pixel 3 (43 & 7). The first write sets coarse X and fine_x, the second sets coarse Y and fine Y in @t.

PPUDATA with the buffer

```
1 ppu6 = NES::PPU.new(cart)
2
3 # Write $42 to the nametable at $2000
4 ppu6.write_register(0x2006_u16, 0x20_u8)
5 ppu6.write_register(0x2006_u16, 0x00_u8)
6 ppu6.write_register(0x2007_u16, 0x42_u8)
7
8 # Read: set the address again
9 ppu6.write_register(0x2006_u16, 0x20_u8)
10 ppu6.write_register(0x2006_u16, 0x00_u8)
11 ppu6.read_register(0x2007_u16)           # dummy read (fills the
buffer)
12 result = ppu6.read_register(0x2007_u16) # now the real data
13
14 passed = result == 0x42_u8
```

We write \$42 to the nametable at \$2000, and to read it back we need to set the address again and do a dummy read before the real read. That's the buffer in action.

The full test suite has 10 tests covering: VBlank timing, PPUSTATUS side effects, NMI enabled/disabled, PPUSCROLL loopy, PPUADDR, PPUDATA buffered and palette immediate, OAM, and PPUCTRL nametable bits. You can see the full file in the repo.

To run them:

```
1 crystal run bin/test_ppu.cr
```

And verify that nestest still passes (making sure we didn't break anything on the CPU side):

```
1 crystal run bin/test_nestest.cr
```

OceanofPDF.com

Wrap up

We covered a lot of ground in this chapter:

- **palette.cr**: the 64 fixed NES colors
- **ppu.cr**: the main class with timing, VBlank, and NMI
- **loopy.cr**: the internal registers v/t/fine_x/w with their helpers
- **8 register modules**: ctrl, mask, status, scroll, addr, data, oam_addr, oam_data
- **vram.cr**: memory routing with nametable and palette mirroring
- **nmi in the CPU**: the method the PPU uses to interrupt the CPU
- **Updated Bus**: routes \$2000-\$3FFF to the PPU
- **Rewritten Emulator**: creates the PPU, connects NMI, runs at 3:1 ratio

The PPU is now counting cycles, tracking scanlines, firing VBlank, and communicating with the CPU through NMI. All the plumbing is in place. But the screen is still black.

In the next chapter we open a window with SDL2 and start the real emulation loop. We still won't see graphics (because we haven't implemented rendering), but we'll have a window running at 60 FPS with the emulator inside. It's the first step toward leaving the terminal behind.

You can see all the changes we made in this chapter in the [repo](#)

OceanofPDF.com

Quick Break: A Brief Reflection on the PPU

We just finished building the PPU skeleton, and before we open a window and start seeing pixels, I think this is a good time to pause and talk about something you're probably wondering: *why is everything so weird?*

Registers that serve two purposes at once. Double writes to build an address. Side effects when you read a register. A shared latch between SCROLL and ADDR. Control bits that get copied to a temporary register, which later gets copied to another register that doubles as both a scroll position and a VRAM pointer.

If you're used to modern graphics APIs, or even if you've just read about how a current GPU works, this probably looks like insanity. And yes, it's confusing. But it's not bad design. It's brilliant design given the constraints of the era.

OceanofPDF.com

Modern GPUs: general-purpose processors

A modern GPU (think RTX 4090 or Radeon RX 7900) is, at its core, a massively parallel general-purpose processor. It has thousands of cores that execute programs called **shaders**: code you write that runs directly on the GPU.

Want a water effect? You write a pixel shader that simulates reflections and refractions. Want dynamic shadows? You write a shader that projects geometry from each light's perspective. Want a post-processing effect like bloom or motion blur? Another shader.

The GPU knows nothing about water, shadows, or blur. It's a generic machine that runs whatever programs you throw at it. It's so generic that today GPUs are used to train AI models, mine cryptocurrency, and run scientific simulations. Game shaders are just one application among many.

This is possible because transistors are cheap today. An RTX 4090 has **76 billion** transistors. You can afford thousands of programmable cores, megabytes of cache, sophisticated memory controllers, and the luxury of flexible hardware.

The NES: every transistor counts

In 1983, a chip like the NES PPU had roughly **20,000 transistors**. Not 20 billion. Twenty thousand. And with that you need to draw 256x240 pixels, 60 times per second, with tiles, sprites, palettes, scroll, and priorities.

There's no room for a programmable processor. No room for shaders. No room for anything generic. Every circuit inside the PPU is designed to do exactly one thing, and it does it in hardware, with logic gates wired directly.

That's why registers pull double duty: why waste transistors on two separate registers when one can handle both jobs? The 15 bits of `v` are a VRAM address *and* a scroll position at the same time because, at the hardware level, **they are the same thing**. The tile the camera is looking at is the tile the PPU needs to fetch from memory.

That's why the latch is shared between SCROLL and ADDR: it's the same physical flip-flop on the chip, not two variables in a program. Adding a second latch would cost transistors they didn't have.

That's why PPUSTATUS has side effects when you read it: reading the register resets the VBlank flag and the latch. It's not a "side effect" in the software sense. It's **the mechanism**. The hardware doesn't have a `clear_vblank()` function the CPU can call. The read *is* the clear.

The escalation: from the NES to the SNES

If you think the NES PPU is complex, the **Super Nintendo** (1990) cranks this philosophy up to eleven. The SNES uses *two* PPU chips working in tandem: PPU1 renders tiles and applies transformations, PPU2 handles window effects, mosaic, and fades. It has 8 different background modes, including the famous **Mode 7** that lets you rotate and scale an entire background layer (that's how F-Zero does its track and how Final Fantasy VI does its aerial map views).

And it's all still fixed-function. No shaders, no programs running on the PPU. Pure dedicated hardware, just with way more modes and configurations. Emulating the SNES PPU is considerably harder than the NES, not because it's conceptually different, but because there are way more combinations of states and modes to get right.

OceanofPDF.com

The 3D transition: PlayStation and beyond

The **PlayStation** (1994) was a massive turning point. For the first time on consoles, the industry left the world of 2D tiles and sprites behind and moved to 3D polygons. The PS1 has a dedicated geometry chip (the GTE, Geometry Transformation Engine) that transforms 3D vertices, and a GPU that rasterizes those textured triangles.

But here's the thing: the PS1 is still **fixed-function**. No programmable shaders. You get two lighting options: flat shading or Gouraud shading. That's it. The GPU doesn't execute programs; it just applies whatever operations are hardwired into it. Textures are mapped with affine interpolation (not perspective-correct, which is why polygons "wobble" when you look at them up close). The hardware calls the shots, and the programmer just tweaks parameters.

The real revolution came in 2000-2001 with **DirectX 8** and Nvidia's **GeForce 3**. For the first time, developers could write small programs (vertex shaders and pixel shaders) that ran **on** the GPU. You were no longer limited to what the hardware had baked in. You could invent your own effects, your own transformations, your own lighting logic. That's where modern GPUs as we know them were born.

From that point on, the trend was clear: each GPU generation got more programmable and more generic, until we arrived at today's monsters that can run anything you throw at them.

Back to the NES

With all that context, take another look at the PPU's design and appreciate the elegance.

With 20,000 transistors, 2KB of VRAM, 64 colors, and a handful of registers, the NES produced the worlds of Mario, the dungeons of Zelda, the planets of Metroid. Everything we built in the previous chapter (the loopy registers, the shared latch, the read buffer, the nametable bits in the temporary register) aren't patches or hacks. They're clever solutions to a brutal engineering problem: doing the absolute most with the absolute least.

It's a design from another era, sure. But it's elegant. And now that we have it implemented, let's use it to put something on screen.

OceanofPDF.com

Chapter 5.2: Opening a Window

So far, our emulator lives in the terminal. It loads a ROM, executes instructions, passes tests, but doesn't show anything on screen. In this chapter, that changes: we're going to open a window and hook the emulator into a visual loop running at 60 frames per second.

We won't see actual game graphics yet (background rendering comes in the next chapter). But we'll have all the visual infrastructure in place: window, renderer, texture, and the game loop. This is the last piece of plumbing before real pixels show up.

OceanofPDF.com

SDL2

To draw on screen we need a graphics library. There are several options (GLFW, raylib, SFML), but we're going with **SDL2** (Simple DirectMedia Layer). It's a C library that handles windows, rendering, keyboard input, and audio. It's cross-platform (Linux, macOS, Windows), lightweight, and it's what most emulators use.

The idea is simple: SDL2 gives us a window and a texture. We fill the texture with the pixels generated by the PPU, and SDL2 takes care of putting them on screen. No OpenGL, no shaders, nothing fancy. A 256x240 texture, updated 60 times per second.

Installation

SDL2 comes in two parts: the runtime library (which you probably already have if you've ever played a game that uses it) and the development package (headers and symlinks for compiling). We need both.

Linux (Ubuntu/Debian):

```
1 sudo apt-get install libsdl2-dev
```

Linux (Fedora):

```
1 sudo dnf install SDL2-devel
```

Linux (Arch):

```
1 sudo pacman -S sdl2
```

macOS:

```
1 brew install sdl2
```

Windows (MSYS2):

```
1 pacman -S mingw-w64-x86_64-SDL2
```

Why the dev package? Because Crystal compiles to a native binary and needs to link against the library. When we put `@[Link("SDL2")]` in our code, the compiler tells the linker “go find -lSDL2”. The dev package installs the `libSDL2.so` symlink that the linker needs to find the library.

[OceanofPDF.com](https://oceanofpdf.com)

Bindings: Crystal Talking to C

SDL2 is written in C. Crystal can't call C functions directly; it needs to know what functions exist, what parameters they take, and what they return. That's what **bindings** are: Crystal-side declarations of the C interface.

Crystal has built-in support for this. With `@[Link("SDL2")]` and a `lib` block, you declare C functions and Crystal handles calling them correctly:

```
1  @[Link("SDL2")]
2  lib LibSDL
3    fun init = SDL_Init(flags : UInt32) : Int32
4    fun quit = SDL_Quit
5    fun create_window = SDL_CreateWindow(
6      title : UInt8*, x : Int32, y : Int32,
7      w : Int32, h : Int32, flags : UInt32
8    ) : Void*
9    # ...
10 end
```

Each `fun` declares a C function: `init` is the Crystal name, `SDL_Init` is the actual name in the C library, and the parameter and return types map to Crystal types. `Void*` is a generic pointer (like SDL's opaque handles for windows, renderers, etc). `UInt8*` is a C string.

On top of those raw bindings, we write Crystal wrappers that are nicer to work with:

```
1  module SDL
2    class Window
3      def initialize(title : String, width : Int32, height : Int32)
4        @ptr = LibSDL.create_window(
5          title,
6          LibSDL::WINDOWPOS_CENTERED, LibSDL::WINDOWPOS_CENTERED,
7          width, height,
8          LibSDL::WINDOW_SHOWN
9        )
10       raise "SDL window creation failed" if @ptr.null?
11     end
12
```

```
13     def to_unsafe : Void*
14         @ptr
15     end
16
17     def finalize
18         LibSDL.destroy_window(@ptr)
19     end
20 end
21 end
```

`to_unsafe` is a Crystal convention: when an object needs to be converted to its C representation (a pointer), Crystal calls `to_unsafe` automatically. `finalize` gets called when the garbage collector frees the object, and that's where we destroy the SDL resource.

You don't need to understand every line of the bindings. The full file is in `bin/bindings/sdl2.cr` in the repo. Just copy it and move on. The important thing is knowing what we have: a `lib LibSDL` block with the C functions, and an `SDL` module with Crystal wrappers (`Window`, `Renderer`, `Texture`, `Event`) that make life easier.

The wrappers we wrote cover the bare minimum we need:

- **Window:** opens a centered window on screen
- **Renderer:** draws to the window (hardware-accelerated)
- **Texture:** a 256x240 pixel buffer in ABGR8888 format (the byte order that matches our frame buffer: R, G, B, A in memory)
- **Event:** reads OS events (close window, key presses)

In future chapters we'll add more to the bindings (scancodes for keyboard input, audio for the APU), but this is enough for now.

OceanofPDF.com

The GUI Script

With the bindings in place, the script that ties everything together is surprisingly short:

```
1  # bin/gui.cr
2
3  require "../src/nes"
4  require "../bindings/sdl2"
5
6  class NESApp
7    SCALE          = 3
8    WINDOW_WIDTH   = NES::PPU::Display::WIDTH * SCALE   # 768
9    WINDOW_HEIGHT  = NES::PPU::Display::HEIGHT * SCALE  # 720
10
11    def initialize(rom_path : String)
12      @emulator = NES::Emulator.new(rom_path)
13      @running  = true
14
15      SDL.init
16      @window   = SDL::Window.new("NES", WINDOW_WIDTH,
WINDOW_HEIGHT)
17      @renderer = SDL::Renderer.new(@window)
18      @texture  = SDL::Texture.new(
19        @renderer,
20        NES::PPU::Display::WIDTH,
21        NES::PPU::Display::HEIGHT
22      )
23    end
24
25    def run
26      while @running
27        frame_start = SDL.ticks
28
29        handle_events
30        frame_buffer = @emulator.run_frame
31
32        @texture.update(frame_buffer, NES::PPU::Display::WIDTH * 4)
33        @renderer.clear
34        @renderer.copy(@texture)
35        @renderer.present
36      end
37    end
38  end
```

```

37         # ~60 FPS (16.67ms per frame)
38         elapsed = SDL.ticks &- frame_start
39         SDL.delay(16_u32 &- elapsed) if elapsed < 16
40     end
41
42     SDL.quit
43 end
44
45 def handle_events
46     event = SDL::Event.new
47     while event.poll
48         if event.quit? || (event.key_down? && event.scancode ==
LibSDL::SCancode_ESCAPE)
49             @running = false
50         end
51     end
52 end
53 end

```

Let's break it down.

Initialize

```

1  def initialize(rom_path : String)
2      @emulator = NES::Emulator.new(rom_path)
3      @running = true
4
5      SDL.init
6      @window = SDL::Window.new("NES", WINDOW_WIDTH,
WINDOW_HEIGHT)
7      @renderer = SDL::Renderer.new(@window)
8      @texture = SDL::Texture.new(
9          @renderer,
10         NES::PPU::Display::WIDTH,
11         NES::PPU::Display::HEIGHT
12     )
13 end

```

We create the emulator with the ROM, then initialize SDL. The window is 768x720 (the NES resolution, 256x240, multiplied by 3). The texture is 256x240, the actual size of the frame buffer. SDL handles scaling the texture to fill the window.

RENDERER_ACCELERATED tells SDL to use the GPU for window rendering (not for emulation, we do that on the CPU). This makes copying the texture to the window fast.

The Game Loop

```
1  def run
2    while @running
3      frame_start = SDL.ticks
4
5      handle_events
6      frame_buffer = @emulator.run_frame
7
8      @texture.update(frame_buffer, NES::PPU::Display::WIDTH * 4)
9      @renderer.clear
10     @renderer.copy(@texture)
11     @renderer.present
12
13     elapsed = SDL.ticks &- frame_start
14     SDL.delay(16_u32 &- elapsed) if elapsed < 16
15   end
16
17   SDL.quit
18 end
```

Each iteration of the while loop is one frame. Here's what happens each time:

1. **Save the start tick** so we can measure how long the frame took
2. **Process SDL events** (keyboard, close window)
3. **run_frame**: run the emulator until the PPU completes a frame (the CPU executes instructions, the PPU advances at 3x, and when it hits scanline 241 it marks the frame as ready). Returns the frame buffer with the pixels
4. **Update the texture** with the new pixels. The second parameter (WIDTH * 4) is the pitch: how many bytes per row. With 256 pixels at 4 bytes each (R, G, B, A), that's 1024
5. **Draw**: clear the screen, copy the texture, and present
6. **Frame limiter**: if all of that took less than 16ms (~60fps), we wait for the remainder. Without this, the emulator would run as fast as possible,

which on a modern machine would be way faster than 60fps. The NES ran at exactly 60.0988 FPS (NTSC), so 16ms per frame is a good-enough approximation

`SDL::ticks` returns milliseconds since SDL was initialized. We use `&-` (wrapping subtraction) because it's `UInt32` and could technically overflow after ~49 days. In practice it doesn't matter, but it's good habit.

Events

```
1  def handle_events
2    event = SDL::Event.new
3    while event.poll
4      if event.quit? || (event.key_down? && event.scancode ==
LibSDL::SCancode_ESCAPE)
5        @running = false
6      end
7    end
8  end
```

SDL accumulates events in a queue. `poll` pulls one off and returns true if there was one. We loop to drain all pending events for this frame.

For now we only handle two things: closing the window (clicking the X) and pressing Escape. In the controls chapter we'll add key mappings for the NES controller.

Main

```
1  if ARGV.empty?
2    puts "Usage: crystal run bin/gui.cr -- <rom_path>"
3    exit 1
4  end
5
6  app = NESApp.new(ARGV[0])
7  app.run
```

This reads the ROM path from the command-line arguments. The `--` is necessary because `crystal run` has its own flags, and `--` tells it

“everything after this goes to the program, not to Crystal”.

OceanofPDF.com

Verifying It Works

Before loading Mario, let's make sure the SDL pipeline works with a quick sanity check. This script fills a buffer with color bands and displays them on screen:

```
1  # bin/test_sdl.cr
2
3  require "./bindings/sdl2"
4
5  WIDTH  = 256
6  HEIGHT = 240
7  SCALE  = 3
8
9  buffer = Bytes.new(WIDTH * HEIGHT * 4)
10
11  HEIGHT.times do |y|
12    WIDTH.times do |x|
13      offset = (y * WIDTH + x) * 4
14      case y // 60
15      when 0 # Red
16        buffer[offset] = 255_u8
17      when 1 # Green
18        buffer[offset + 1] = 255_u8
19      when 2 # Blue
20        buffer[offset + 2] = 255_u8
21      else # White
22        buffer[offset] = 255_u8
23        buffer[offset + 1] = 255_u8
24        buffer[offset + 2] = 255_u8
25      end
26      buffer[offset + 3] = 255_u8
27    end
28  end
29
30  SDL.init
31  window = SDL::Window.new("SDL Test", WIDTH * SCALE, HEIGHT *
SCALE)
32  renderer = SDL::Renderer.new(window)
33  texture = SDL::Texture.new(renderer, WIDTH, HEIGHT)
34
35  texture.update(buffer, WIDTH * 4)
```

```
36 renderer.clear
37 renderer.copy(texture)
38 renderer.present
39
40 puts "Showing test pattern. Press ESC or close window to exit."
41
42 event = SDL::Event.new
43 running = true
44 while running
45   while event.poll
46     if event.quit? || (event.key_down? && event.scancode ==
LibSDL::SCANCODE_ESCAPE)
47       running = false
48     end
49   end
50   SDL.delay(16_u32)
51 end
52
53 SDL.quit
```

```
1 crystal run bin/test_sdl.cr
```

You should see a window with 4 horizontal bands: red, green, blue, and white. If the colors look right (red is red, not blue), the pixel format is configured correctly and the PPU's colors will come through just fine.

Why bother with this test? Because it verifies that the **byte order** is correct. Our frame buffer writes [R, G, B, A] as consecutive bytes. The texture uses the ABGR8888 format, which on a little-endian machine (x86) expects exactly that order in memory. If we'd picked the wrong format, red would show up as blue or the colors would be totally scrambled.



Figure 7. SDL2 Test Pattern

Now, with the ROM:

```
1 crystal run bin/gui.cr -- .roms/mario.nes
```

The window opens and... it's black. That's expected. The emulator is running internally (the CPU is executing, the PPU is counting cycles, NMIs are firing), but we haven't implemented background rendering yet. The frame buffer is full of zeros, which means black.

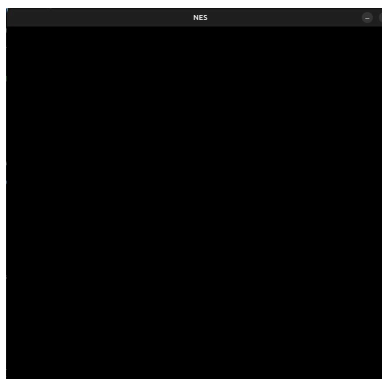


Figure 8. SDL2 Running Mario

In the next chapter, that changes. We'll implement background rendering and for the first time we'll see Mario's tiles on screen.

You can see the full code in the [repo](#).

OceanofPDF.com

Wrap up

- SDL2 is the library we use for windowing, rendering, and (later) input and audio
- The **bindings** in `bin/bindings/sdl2.cr` declare the C functions so Crystal can call them. No need to understand every line
- **bin/gui.cr** is the main script: it creates a window, runs the emulator frame by frame, copies the pixels to an SDL texture, and caps it at ~60 FPS
- The **SDL test** (`bin/test_sdl.cr`) verifies the visual pipeline works with a color test pattern
- The screen is black because we haven't built rendering yet, and that's coming in the next chapter

OceanofPDF.com

Chapter 5.3: Rendering the Background

So far our PPU is a very expensive clock. It counts cycles, tracks scanlines, fires VBlank, notifies the CPU via NMI... but it doesn't draw a single pixel. We opened a window in the previous chapter and the screen is black. Under the hood, Mario has already booted up, written tiles into the nametables, configured the palettes, set the scroll, and is just sitting there waiting for someone to actually display something. Well, the time has come.

In this chapter we teach the PPU how to draw the background. By the end, you'll see the Super Mario Bros title screen: the logo, the copyright, the clouds, the ground. Everything except Mario and the Goombas, which are sprites and will come in a later chapter.

OceanofPDF.com

What Mario Already Did

Before we dive into rendering, let's think about what's been happening since the emulator started.

When we loaded the ROM, the CPU started executing Mario's code. In the first few frames, Mario does a ton of initialization work:

1. Writes to PPUCTRL (\$2000): "use pattern table 0 for the background"
2. Writes to PPUMASK (\$2001): "turn on background rendering"
3. Writes data into the nametables via PPUADDR/PPUDATA: "put tile \$24 (sky) here, tile \$B4 (ground) there..."
4. Writes the palettes at \$3F00+: "palette 0 is blue/light-blue/white, palette 2 is brown/orange/beige..."
5. Writes to PPUSCROLL (\$2005): "the camera is at position 0,0"

All of this already works. Our PPU skeleton received those writes and stored the data correctly in @ctrl, @mask, @nametable_ram, @palette_ram, and @t. The data is there. We just need to read it and turn it into pixels.

Register Helpers

We need to be able to ask the registers specific questions. In chapter 5.1 we left PPUCTRL and PPUMASK pretty barebones. Let's add the helpers that rendering is going to need.

PPUCTRL: Which Pattern Table to Use

```
1  # src/nes/ppu/registers/ctrl.cr
2
3  module NES
4    class PPU
5      module Registers
6        module Ctrl
7          FLAG_VRAM_INCREMENT      = 0x04_u8
8          FLAG_SPR_PATTERN_TABLE   = 0x08_u8
9          FLAG_BG_PATTERN_TABLE    = 0x10_u8
10         FLAG_NMI_ENABLED         = 0x80_u8
11
12         PATTERN_TABLE_0 = 0x0000_u16
13         PATTERN_TABLE_1 = 0x1000_u16
14
15         def write_ctrl(value : UInt8)
16           @ctrl = value
17           nametable = value.to_u16 & 0x03
18           set_nametable(nametable)
19         end
20
21         def vram_increment : UInt16
22           (@ctrl & FLAG_VRAM_INCREMENT) != 0 ? 32_u16 : 1_u16
23         end
24
25         def nmi_enabled? : Bool
26           (@ctrl & FLAG_NMI_ENABLED) != 0
27         end
28
29         def background_pattern_table : UInt16
30           (@ctrl & FLAG_BG_PATTERN_TABLE) != 0 ? PATTERN_TABLE_1
31           : PATTERN_TABLE_0
32         end
33       end
34     end
35   end
```



```

33         def sprite_pattern_table : UInt16
34             (@ctrl & FLAG_SPR_PATTERN_TABLE) != 0 ? PATTERN_TABLE_1
: PATTERN_TABLE_0
35         end
36     end
37 end
38 end
39 end

```

We already had `write_ctrl`, `vram_increment`, and `nmi_enabled?`. The new additions are the two pattern table constants and the two methods that use them.

`background_pattern_table` checks bit 4 of `PPUCTRL`. If it's set, the background tiles live in pattern table 1 (\$1000). If not, they're in pattern table 0 (\$0000). Mario uses pattern table 0 for the background, but other games use 1. `sprite_pattern_table` does the same thing but with bit 3. We'll need it when we implement sprites.

PPUMASK: What Gets Rendered

```

1  # src/nes/ppu/registers/mask.cr
2
3  module NES
4      class PPU
5          module Registers
6              module Mask
7                  FLAG_SHOW_BG      = 0x08_u8
8                  FLAG_SHOW_SPRITES = 0x10_u8
9
10                 def write_mask(value : UInt8)
11                     @mask = value
12                 end
13
14                 def show_background? : Bool
15                     (@mask & FLAG_SHOW_BG) != 0
16                 end
17
18                 def show_sprites? : Bool
19                     (@mask & FLAG_SHOW_SPRITES) != 0
20                 end
21

```

```
22         def rendering_enabled? : Bool
23             show_background? || show_sprites?
24         end
25     end
26 end
27 end
28 end
```

In chapter 5.1, `mask.cr` only had `write_mask`. Now we add three queries the renderer needs:

- `show_background?`: bit 3 of PPUMASK. If it's off, the background doesn't get drawn.
- `show_sprites?`: bit 4. We'll use this in the sprites chapter.
- `rendering_enabled?`: if either one is on, the PPU is rendering. We need this to know whether the scroll increments should run.

OceanofPDF.com

Reading the Scroll Position

In chapter 5.1 we implemented the setters for @t (the methods that PPUSCROLL and PPUADDR use to write the scroll position). Now we need the other direction: reading the current position from @v.

Remember the layout of v:

```
1  yyy NN YYYYY XXXXX
2  ||| || |||||  _____ coarse X scroll (0-31 tiles)
3  ||| ||  LLLLL _____ coarse Y scroll (0-29 tiles)
4  |||  LL _____ nametable select (0-3)
5  LL _____ fine Y scroll (0-7 pixels)
```

To render, we need to extract each field. That's four getters with bit operations:

```
1  # src/nes/ppu/loopy.cr (adding the getters)
2
3  def v_coarse_x : Int32
4    (@v & MASK_COARSE_X).to_i32
5  end
6
7  def v_coarse_y : Int32
8    ((@v & MASK_COARSE_Y) >> 5).to_i32
9  end
10
11 def v_fine_y : Int32
12   ((@v & MASK_FINE_Y) >> 12).to_i32
13 end
14
15 def v_nametable : Int32
16   ((@v & MASK_NAME_TABLE) >> 10).to_i32
17 end
```

Each one ANDs with the corresponding mask to isolate the bits, then right-shifts them down to position 0. v_coarse_x doesn't need a shift because it's already in the lowest bits (4-0). The others do: coarse Y is in bits 9-5 (shift 5), fine Y in bits 14-12 (shift 12), nametable in bits 11-10 (shift 10).

They return Int32 instead of UInt16 for convenience: we'll use them for position arithmetic and it saves us from casting everything all the time.

OceanofPDF.com

The Rendering: From Tile to Pixel

Now for the main event. We're going to implement `display.cr`, the module that converts everything the PPU has in memory into visible pixels.

First, some constants we'll need:

```
1  # src/nes/ppu/display.cr
2
3  module NES
4    class PPU
5      module Display
6        WIDTH = 256
7        HEIGHT = 240
8
9        CYCLES_PER_SCANLINE = 341
10       SCANLINES_PER_FRAME = 262
11       VBLANK_SCANLINE = 241
12       PRE_RENDER_SCANLINE = 261
13
14       TILES_PER_ROW = 32
15       PIXELS_PER_TILE = 8
16       BYTES_PER_TILE = 16
17       BITPLANE_OFFSET = 8
18       ATTRIBUTE_TABLE_OFFSET = 960
19       COLORS_PER_PALETTE = 4
20       BYTES_PER_PIXEL = 4
```

The timing ones we already had. The new ones are all about tiles: 32 tiles per row, 8 pixels per tile, 16 bytes per tile in the pattern table, the high bitplane starts 8 bytes after the low one, the attribute table starts 960 bytes into each nametable (after the $32 \times 30 = 960$ bytes of tile indices), 4 colors per palette, and 4 bytes per pixel in the frame buffer (RGBA).

render_scanline: The Main Loop

```
1  def render_scanline(y : Int32)
2    WIDTH.times do |pixel|
3      bg_color, bg_palette = if show_background?
```

```
4         background_pixel(pixel)
5     else
6         {0, 0}
7     end
8
9     set_pixel(pixel, y, palette_color(bg_palette,
bg_color))
10 end
11 end
```

A loop of 256 iterations, one per pixel. For each pixel:

1. If the background is on, we calculate the color and palette with `background_pixel`
2. If it's off, color 0 and palette 0 (which will give us the universal background color)
3. We convert the (palette, color) pair to the final NES color with `palette_color`
4. We write the pixel to the frame buffer with `set_pixel`

Later on, between steps 2 and 3, we'll add sprite logic: calculate the sprite pixel, decide priorities, and blend with the background. But for now it's background only.

Notice that `background_pixel` returns two things: the color (0-3, the 2-bit value that comes out of the bitplanes) and the palette (0-3, the one the attribute table specifies). We need them separate because color 0 has special meaning: it's transparent, and in that case the universal background color is used regardless of the palette.

background_pixel: The Full Pipeline

This is the most important method in the chapter. Let's walk through it step by step:

```
1     def background_pixel(pixel : Int32)
2         scroll_x      = @fine_x.to_i32 + pixel
3         tile_column   = v_coarse_x + (scroll_x //
PIXELS_PER_TILE)
```

```
4      pixel_in_tile = scroll_x % PIXELS_PER_TILE
5      nametable     = v_nametable
```

First we calculate the real X position of the pixel we want to draw. `@fine_x` is the pixel offset within the first visible tile (0-7), and `pixel` is the pixel within the scanline (0-255). Adding them gives us the X position in scroll coordinates.

From that we get two things:

- `tile_column`: which tile we land on. `v_coarse_x` tells us the first visible tile (the one the scroll pointed to), and we add however many more tiles we need to advance for this pixel's position.
- `pixel_in_tile`: which pixel within that tile (0-7). It's the remainder of dividing by 8.

A concrete example: if the scroll is 0 (`coarse_x = 0`, `fine_x = 0`) and we're drawing pixel 20, then `tile_column = 0 + 20/8 = 2` and `pixel_in_tile = 20 % 8 = 4`. We're on tile 2, pixel 4.

If the scroll were 3 (`fine_x = 3`) and the pixel were 20, then `scroll_x = 23`, `tile_column = 0 + 23/8 = 2`, `pixel_in_tile = 23 % 8 = 7`. Same tile, but pixel 7 (further right). The `fine_x` "shifted" everything 3 pixels over.

```
1      # Nametables sit side by side. If tile_column >= 32,
2      # we've crossed into the next nametable:
3      #   [--- nametable 0 ---][--- nametable 1 ---]
4      #   col 0 ... col 31   col 0 ... col 31
5      if tile_column >= TILES_PER_ROW
6          tile_column -= TILES_PER_ROW
7          nametable ^= 1
8      end
```

Each nametable has 32 columns (tiles 0-31). If `tile_column` reaches 32 or more, we've crossed over to the other side. We subtract 32 to get back into the 0-31 range, and flip the low bit of the nametable with XOR (`^= 1`). This toggles between nametable 0 and 1 (or between 2 and 3, depending on the

base). This is horizontal mirroring in action: the two nametables sit side by side, and when you reach the edge of one, you wrap into the other.

```
1      nametable_address = NAMETABLE_START.to_i32 + nametable *  
NAMETABLE_SIZE.to_i32  
2      tile_address      = nametable_address + v_coarse_y *  
TILES_PER_ROW + tile_column  
3      tile_index        = vram_read(tile_address.to_u16)
```

Now we calculate the VRAM address where the tile index lives. The nametable starts at $\$2000 + \text{nametable} * 1024$ bytes. Within the nametable, tiles are arranged row by row: row 0 occupies bytes 0-31, row 1 bytes 32-63, and so on. So the tile's position is $\text{row} * 32 + \text{column}$.

`vram_read` reads that byte from VRAM. What we get back is the **tile index** in the pattern table. Not the tile's pixels, but a number from 0 to 255 that says "go grab tile number X." The game wrote those indices when it set up the screen.

```
1      color    = tile_pixel(tile_index, v_fine_y, pixel_in_tile)  
2      palette  = color != 0 ? tile_palette(nametable_address,  
tile_column, v_coarse_y) : 0  
3  
4      {color, palette}  
5      end
```

With the tile index, we go fetch the actual pixel from the bitplanes (`tile_pixel`). This gives us a value from 0 to 3. If it's 0, the pixel is transparent and the palette doesn't matter (we use 0). If it's 1, 2, or 3, we consult the attribute table to find out which palette to use (`tile_palette`).

That's the full pipeline. Let's walk through a Mario example:

```
1 pixel 100, scanline 5 (sky, near the logo)  
2 scroll_x = 0 + 100 = 100  
3 tile_column = 0 + 100/8 = 12  
4 pixel_in_tile = 100 % 8 = 4  
5 nametable_address = $2000  
6 tile_address = $2000 + 5*32 + 12 = $20AC  
7 tile_index = vram_read($20AC) = $24 (tile "empty sky")
```

```
8   tile_pixel($24, 5, 4) = 0 (transparent, it's empty sky)
9   color 0 => universal background color => blue
```

tile_pixel: Reading the Bitplanes

```
1   def tile_pixel(tile_index : UInt8, tile_row : Int32,
pixel_in_tile : Int32) : Int32
2       bitplane_address = background_pattern_table.to_i32 +
tile_index.to_i32 * BYTES_PER_TILE + tile_row
3       low_bitplane      = vram_read(bitplane_address.to_u16)
4       high_bitplane     = vram_read((bitplane_address +
BITPLANE_OFFSET).to_u16)
5
6       bit               = (PIXELS_PER_TILE - 1) - pixel_in_tile
7       low_bit           = (low_bitplane >> bit) & 1
8       high_bit          = (high_bitplane >> bit) & 1
9       ((high_bit << 1) | low_bit).to_i32
10      end
```

This is the practical version of the bitplane theory from chapter 5.0. Each tile takes up 16 bytes in the pattern table: 8 bytes for the low bitplane, followed by 8 bytes for the high bitplane. Each byte represents one row of the tile (8 pixels).

The low bitplane address is: `pattern_table_base + tile_index * 16 + row`. `background_pattern_table` returns \$0000 or \$1000 depending on what Mario configured in `PPUCTRL`. The high bitplane is 8 bytes later (`BITPLANE_OFFSET`).

To extract an individual pixel: bit 7 of each byte is the leftmost pixel, and bit 0 is the rightmost. If we want pixel 0 (left), we read bit 7. If we want pixel 4, we read bit 3. The formula is `bit = 7 - pixel_in_tile`.

Finally, we combine the low bit and the high bit into a 2-bit value: `(high << 1) | low`. That gives us a color from 0 to 3.

tile_palette: The Attribute Table

```
1   def tile_palette(nametable_address : Int32, tile_column :
Int32, tile_row : Int32) : Int32
```

```

2      attribute_column = tile_column // 4
3      attribute_row    = tile_row // 4
4      attribute_address = nametable_address +
ATTRIBUTE_TABLE_OFFSET + attribute_row * (TILES_PER_ROW // 4) +
attribute_column
5      attribute_byte    = vram_read(attribute_address.to_u16)
6
7      right = (tile_column // 2) & 1
8      bottom = (tile_row // 2) & 1
9      shift = (bottom * 2 + right) * 2
10
11      ((attribute_byte >> shift) & (COLORS_PER_PALETTE -
1)).to_i32
12      end

```

This implements the attribute table logic from chapter 5.0. Remember: each byte in the attribute table covers a 4x4 block of tiles, subdivided into four 2x2 quadrants. Each quadrant gets 2 bits that select a palette (0-3).

Step by step:

1. **Find the byte:** we divide the tile's column and row by 4 to figure out which attribute table block we're in. The attribute table starts 960 bytes into the nametable and has 8 columns (32 tiles / 4).
2. **Find the quadrant:** within the 4x4 block, we need to know if we're in the left or right half (`right`), and the top or bottom half (`bottom`). Divide by 2 and check the low bit.
3. **Extract the 2 bits:** the quadrants are packed into the byte like this:
 - Bits 1-0: top-left
 - Bits 3-2: top-right
 - Bits 5-4: bottom-left
 - Bits 7-6: bottom-right

The formula $(\text{bottom} * 2 + \text{right}) * 2$ gives us the correct shift: 0, 2, 4, or 6. We shift and AND with 3 to get the 2 bits.

palette_color: The Final Color

```
1      def palette_color(palette : Int32, color : Int32) : UInt8
2          if color == 0
3              vram_read(PALETTE_START)
4          else
5              vram_read((PALETTE_START + palette * COLORS_PER_PALETTE
+ color).to_u16)
6          end
7      end
```

The final piece of the puzzle. If the color is 0 (transparent), we read the universal background color at \$3F00. Otherwise, we index into the palette: palette 0 is at \$3F00-\$3F03, palette 1 at \$3F04-\$3F07, and so on.

It returns an index from 0 to 63 into the NES master palette (the 64 fixed hardware colors we defined in `palette.cr` back in chapter 5.1).

set_pixel: To the Frame Buffer

```
1      def set_pixel(x : Int32, y : Int32, nes_color : UInt8)
2          rgb      = PALETTE[nes_color.to_i32 & 0x3F]
3          offset = (y * WIDTH + x) * BYTES_PER_PIXEL
4
5          @frame_buffer[offset]      = rgb[0].to_u8
6          @frame_buffer[offset + 1] = rgb[1].to_u8
7          @frame_buffer[offset + 2] = rgb[2].to_u8
8          @frame_buffer[offset + 3] = 255_u8
9      end
```

Takes the NES index (0-63), looks up the RGB color in the master palette, and writes it to the frame buffer in RGBA format. The `& 0x3F` clamps the value to 63 (6 bits). Alpha is always 255 (opaque) because the NES doesn't have screen-level transparency (color 0 transparency is resolved earlier, when we decide what to draw).

The offset is linear: `row * width + column`, multiplied by 4 (bytes per pixel). The frame buffer is a flat array that SDL2 will read directly.

Hooking Into step

We have the rendering logic. Now we need the PPU to actually call it at the right time. Let's modify `step` in `ppu.cr`:

```
1 # src/nes/ppu.cr (step and new helpers)
2
3 def step
4   render_if_visible
5   update_scroll_registers
6   advance_cycle
7 end
```

The `step` we had in chapter 5.1 only called `advance_cycle`. Now it does three things: render if we're on a visible scanline, update the scroll registers, and advance the cycle counter.

```
1 def render_if_visible
2   return unless @scanline < HEIGHT && @cycle == 0
3
4   render_scanline(@scanline)
5 end
```

`render_if_visible` checks two things: that we're on a visible scanline (0-239) and that it's cycle 0 of the scanline. This is a simplification: the real NES renders pixel by pixel across the first 256 cycles of the scanline. We render the whole scanline at once on cycle 0. It's simpler and the result is the same for the vast majority of games.

The Scroll Increments

Now for the missing piece: during rendering, the PPU doesn't just read the scroll position from @v to know what to draw. It also **automatically modifies it** as it goes.

Why? Think of it this way: each scanline draws one row of pixels. At the end of the row, the PPU needs to “move down” to the next row. And after all 240 visible scanlines, it needs to go back to the top for the next frame. These movements happen by modifying the fields of @v.

The increments happen at specific points within the scanline:

```
1 # src/nes/ppu.cr (new constants)
2
3 DOT_FINE_Y_INCREMENT      = 256
4 DOT_COPY_HORIZONTAL      = 257
5 DOT_COPY_VERTICAL_START  = 280
6 DOT_COPY_VERTICAL_END    = 304
```

And the method that calls them:

```
1 def update_scroll_registers
2   return unless rendering_enabled?
3   return unless @scanline < HEIGHT || @scanline ==
PRE_RENDER_SCANLINE
4
5   if @cycle == DOT_FINE_Y_INCREMENT
6     fine_y_increment
7   elsif @cycle == DOT_COPY_HORIZONTAL
8     copy_horizontal
9   elsif @scanline == PRE_RENDER_SCANLINE && @cycle >=
DOT_COPY_VERTICAL_START && @cycle <= DOT_COPY_VERTICAL_END
10     copy_vertical
11   end
12 end
```

This only runs if rendering is enabled (at least background or sprites turned on) and if we're on a visible scanline or the pre-render scanline. Three

operations, each at a specific cycle:

- **Cycle 256** (`fine_y_increment`): after finishing the row, we advance one pixel row downward.
- **Cycle 257** (`copy_horizontal`): at the start of the “rest period” between scanlines, we reset the horizontal position from `@t`. This ensures each scanline starts from the same X column the game set via `PPUSCROLL`.
- **Cycles 280-304** of the pre-render scanline (`copy_vertical`): at the end of the frame, we reset the vertical position from `@t`. This ensures the next frame starts from the same Y row the game set.

Now let’s look at the methods in `loopy.cr`. We need some new constants:

```
1  # src/nes/ppu/loopy.cr (new constants)
2
3  # Horizontal bits: coarse X + horizontal nametable bit
4  MASK_HORIZONTAL = 0x041F_u16
5  # Vertical bits: coarse Y + fine Y + vertical nametable bit
6  MASK_VERTICAL   = 0x7BE0_u16
7
8  # XOR these to flip a nametable bit in v
9  FLIP_NAMETABLE_H = 0x0400_u16
10 FLIP_NAMETABLE_V = 0x0800_u16
11
12 # Adding this to v increments fine_y by 1 (fine_y lives in bits
13 # 12-14)
14 FINE_Y_UNIT = 0x1000_u16
15
16 # Boundary values
17 MAX_COARSE_X = 31
18 MAX_COARSE_Y = 29
19 MAX_FINE_Y   = 7
```

`MASK_HORIZONTAL` captures the coarse X bits (bits 4-0) and the horizontal nametable bit (bit 10). `MASK_VERTICAL` captures coarse Y (bits 9-5), fine Y (bits 14-12), and the vertical nametable bit (bit 11). These masks are used to selectively copy from `@t` to `@v`.

`FLIP_NAMETABLE_H` and `FLIP_NAMETABLE_V` are for XORing to toggle between nametables. If you’re in nametable 0 and XOR with

FLIP_NAMETABLE_H, you jump to 1, and vice versa.

coarse_x_increment

```
1 def coarse_x_increment
2   if v_coarse_x == MAX_COARSE_X
3     @v = (@v & CLEAR_COARSE_X) ^ FLIP_NAMETABLE_H
4   else
5     @v = @v &+ 1_u16
6   end
7 end
```

Advances one tile column to the right. If we're at the last column (31), we clear coarse X (back to 0) and flip the horizontal nametable. Think of it like reaching the right edge of a page and turning to the next one. If we're not at the edge, we just add 1.

We don't actually call `coarse_x_increment` directly because we render the entire scanline at once (with `background_pixel` calculating the X position from `v_coarse_x + pixel`). The real PPU advances coarse X as it draws each tile, but in practice, the result is the same.

fine_y_increment

```
1 def fine_y_increment
2   if v_fine_y < MAX_FINE_Y
3     @v = @v &+ FINE_Y_UNIT
4     return
5   end
6
7   @v = @v & ~MASK_FINE_Y
8   y = v_coarse_y
9
10  if y == MAX_COARSE_Y
11    @v = (@v & ~MASK_COARSE_Y) ^ FLIP_NAMETABLE_V
12  elsif y == 31
13    @v = @v & ~MASK_COARSE_Y
14  else
15    @v = (@v & ~MASK_COARSE_Y) | ((y + 1).to_u16 << 5)
```

```
16     end
17 end
```

This one is more complex because it has three levels of overflow:

1. **fine Y < 7** (we're not on the last pixel of the tile): just add 1 to fine Y. Since fine Y lives in bits 14-12, adding `FINE_Y_UNIT` (\$1000) increments that field without touching anything else. Each tile has 8 pixel rows (0-7), so fine Y 0 is the first row and fine Y 7 is the last. As long as we haven't hit 7, we're still within the same tile.
2. **fine Y == 7** (last pixel of the tile, time to move down to the next one): fine Y goes back to 0, and we need to advance coarse Y (move to the next tile row down).
 - **coarse Y == 29** (last visible row): coarse Y goes back to 0 and we flip the vertical nametable. This is the vertical equivalent of what `coarse_x_increment` does when it reaches column 31.
 - **coarse Y == 31** (in the attribute table zone): a hardware quirk. The nametable has 30 tile rows (0-29) and then the remaining 2 rows (30-31) are the attribute table. If we reach 31, we wrap back to 0 without flipping the nametable. Games normally don't end up here, but the real PPU handles it this way.
 - **coarse Y < 29** (still more visible tile rows to go): just add 1 to coarse Y. The nametable has 30 tile rows (0-29), so as long as we haven't reached the last one, we just move down.

copy_horizontal and copy_vertical

```
1 def copy_horizontal
2   @v = (@v & ~MASK_HORIZONTAL) | (@t & MASK_HORIZONTAL)
3 end
4
5 def copy_vertical
6   @v = (@v & ~MASK_VERTICAL) | (@t & MASK_VERTICAL)
7 end
```

These are the simplest ones. They copy bits from `@t` to `@v`. `copy_horizontal` copies coarse X and the horizontal nametable bit.

`copy_vertical` copies coarse Y, fine Y, and the vertical nametable bit.

Why copy from `@t`? Because `@t` holds what the game wrote to `PPUSCROLL`. While the PPU renders, `@v` gets modified by the increments. At the end of each scanline, `copy_horizontal` restores the original X position. And at the end of the frame, `copy_vertical` restores the original Y position. This ensures the next frame starts from the position the game requested.

It's like a typewriter: at the end of each line, the carriage returns to the left margin (`copy_horizontal`). And when the page is full, it goes back to the very top (`copy_vertical`).

OceanofPDF.com

The Complete step

Let's put it all together. Here's `ppu.cr` with all the modifications:

```
1  # src/nes/ppu.cr
2
3  def step
4    render_if_visible
5    update_scroll_registers
6    advance_cycle
7  end
8
9  def render_if_visible
10   return unless @scanline < HEIGHT && @cycle == 0
11
12   render_scanline(@scanline)
13 end
14
15 def update_scroll_registers
16   return unless rendering_enabled?
17   return unless @scanline < HEIGHT || @scanline ==
PRE_RENDER_SCANLINE
18
19   if @cycle == DOT_FINE_Y_INCREMENT
20     fine_y_increment
21   elsif @cycle == DOT_COPY_HORIZONTAL
22     copy_horizontal
23   elsif @scanline == PRE_RENDER_SCANLINE && @cycle >=
DOT_COPY_VERTICAL_START && @cycle <= DOT_COPY_VERTICAL_END
24     copy_vertical
25   end
26 end
```

The order matters: first we render (using the current position in `@v`), then we update the position (for the next scanline), and finally we advance the cycle counter. `advance_cycle` stays exactly as it was in chapter 5.1: counts cycles, detects VBlank, fires NMI.

The Moment of Truth

```
1 crystal run bin/gui.cr -- .roms/mario.nes
```

And if everything went right... the Super Mario Bros title screen appears in the window. The “SUPER MARIO BROS.” logo up top, the Nintendo copyright at the bottom, the clouds, the bushes, the brick ground, and the menu. All rendered from the nametables, pattern tables, and palettes that Mario wrote into VRAM.

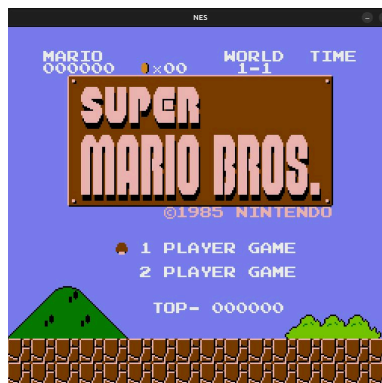


Figure 9. Title Screen Mario (Only the Background)

What you **won't** see: Mario standing on the title screen (he's a sprite), the coins in the level (sprites), the Goombas (sprites). Everything that moves independently of the background is a sprite, and that's coming in chapter 5.5.

Scrolling doesn't work for actual gameplay yet either. If you could somehow move Mario through the level (which you can't, since we don't have controls yet), the background wouldn't scroll correctly. The loopy increments we just implemented keep the frame position stable, which is exactly what the title screen needs. Dynamic scrolling will come together as we move forward.

Wrap up

We covered a lot of ground:

- **ctrl.cr**: `background_pattern_table` and `sprite_pattern_table` to know where to look for tiles
- **mask.cr**: `show_background?`, `show_sprites?`, `rendering_enabled?` to know what to draw
- **loopy.cr**: getters (`v_coarse_x`, `v_coarse_y`, `v_fine_y`, `v_nametable`) to read the scroll position, and increments (`coarse_x_increment`, `fine_y_increment`, `copy_horizontal`, `copy_vertical`) for the PPU to advance automatically
- **display.cr**: the full rendering pipeline: `render_scanline` calls `background_pixel`, which queries the nametable, reads the bitplanes with `tile_pixel`, looks up the palette with `tile_palette`, and converts to RGB with `palette_color` and `set_pixel`
- **ppu.cr**: step now renders on visible scanlines and updates the scroll registers at the right moments

The full chain is: scroll position in @v => nametable => tile index => pattern table => bitplanes => 2-bit color => attribute table => palette => palette RAM => NES index => master palette => RGB => frame buffer => SDL2 => screen.

That's a long journey for a single pixel. But that's how the NES works, and now we have it all wired up.

You can find the code for this chapter in the repo [here](#).

In the next chapter we add controls so you can press Start and actually get into the game. After that, sprites. But for now, enjoy the view. This is the first real image your emulator has ever produced.

Chapter 5.4: Controls

We have Mario's title screen. The logo, the clouds, the floor. But it's just a static image. We can't press Start to begin the game, we can't move, we can't do anything. It's like staring at a TV with the controller unplugged.

In this chapter we plug in the controller. But there's a surprise waiting.

OceanofPDF.com

The NES controller

The NES controller has 8 buttons: A, B, Select, Start, and the 4 D-pad directions. The CPU reads the buttons through two I/O addresses: \$4016 for controller 1 and \$4017 for controller 2.

But it's not as simple as reading a single byte with all 8 buttons. The NES uses a **shift register** with a strobe protocol:

1. The game writes \$01 to \$4016 (strobe ON): this tells the controller “capture the current button state”
2. The game writes \$00 to \$4016 (strobe OFF): freezes the snapshot
3. The game reads \$4016 eight times: each read returns ONE button (in bit 0), and the shift register advances to the next one

The order is fixed: A, B, Select, Start, Up, Down, Left, Right.

Why so complicated? Cost. An 8-bit shift register needs far fewer wires than 8 parallel data lines. In 1983, every pin on the connector cost money.

The Controller class

```
1  # src/nas/controller.cr
2
3  module NES
4    class Controller
5      # Button masks (bit position in the state byte)
6      BUTTON_A      = 0x01_u8
7      BUTTON_B      = 0x02_u8
8      BUTTON_SELECT = 0x04_u8
9      BUTTON_START  = 0x08_u8
10     BUTTON_UP     = 0x10_u8
11     BUTTON_DOWN   = 0x20_u8
12     BUTTON_LEFT   = 0x40_u8
13     BUTTON_RIGHT  = 0x80_u8
14
15     def initialize
16       @buttons = 0_u8          # Current button state
17       @shift_register = 0_u8  # Frozen snapshot, read one bit at
a time
```

```
18         @strobe = false           # If true, the shift register
reloads on every read
19     end
```

Each button is a bit in a byte. A is bit 0, B is bit 1, and so on. @buttons is the live state, updated when the player presses or releases a key on the keyboard. @shift_register is the frozen snapshot that the CPU reads. @strobe controls whether the snapshot keeps updating or stays frozen.

Now the behavior:

```
1     def press(button : UInt8)
2         @buttons |= button
3     end
4
5     def release(button : UInt8)
6         @buttons &= ~button
7     end
8
9     # Bit 0 controls the strobe
10    def write(value : UInt8)
11        @strobe = (value & 1) != 0
12        @shift_register = @buttons if @strobe
13    end
14
15    # Returns one button per read (bit 0)
16    def read : UInt8
17        if @strobe
18            # While strobe is active, always returns the state of A
19            return @buttons & 1
20        end
21
22        value = @shift_register & 1
23        @shift_register >>= 1
24        value
25    end
26 end
27 end
```

press turns a bit on with OR. release turns it off by ANDing with the complement. Classic bit manipulation that should be second nature by this point in the book.

`write` looks at bit 0 of the value. If it's 1, it activates the strobe and captures the current state into the shift register. If it's 0, it deactivates the strobe and freezes the snapshot.

`read` is where the shift register magic happens. If the strobe is active, it always returns the state of button A (bit 0 of buttons). This is a hardware quirk: while the strobe is on, the shift register keeps reloading, so you always get the first button. Once the strobe turns off, each read returns bit 0 of the shift register and shifts it right. So 8 successive reads give you A, B, Select, Start, Up, Down, Left, Right.

To see this in action, here's what Mario does in assembly to read the controller:

```
1  LDA #$01
2  STA $4016      ; strobe ON: capture button state
3  LDA #$00
4  STA $4016      ; strobe OFF: freeze snapshot
5
6  LDA $4016      ; read A
7  LDA $4016      ; read B
8  LDA $4016      ; read Select
9  LDA $4016      ; read Start
10 LDA $4016      ; read Up
11 LDA $4016      ; read Down
12 LDA $4016      ; read Left
13 LDA $4016      ; read Right
```

Simple. Write 1, write 0, read 8 times. Each `LDA $4016` invokes our `read`, which returns one bit and advances to the next.

Connecting to the Bus

The controller needs to be accessible from the CPU's memory map. Let's add the constants in `addresses.cr`:

```
1 # src/nes/bus/addresses.cr
2
3 # ===== IO Registers =====
4 OAM_DMA = 0x4014_u16
5
6 # ===== Controllers =====
7 CONTROLLER_1 = 0x4016_u16
8 CONTROLLER_2 = 0x4017_u16
```

OAM_DMA at \$4014 will come up later in this chapter. You'll see why soon.

Now the Bus. It needs to create both controllers and route reads and writes:

```
1 # src/nes/bus.cr
2
3 module NES
4   class Bus
5     include Addresses
6
7     getter controller_1 : Controller
8     getter controller_2 : Controller
9
10    def initialize(cartridge : Cartridge, ppu : PPU)
11      @cartridge = cartridge
12      @ppu = ppu
13      @ram = Bytes.new(RAM_SIZE, 0_u8)
14      @controller_1 = Controller.new
15      @controller_2 = Controller.new
16    end
```

Two controllers, one for each port. Most games only use controller 1, but the NES supports two.

In `cpu_read`, we add the new ranges:

```
1  def cpu_read(address : UInt16) : UInt8
2    case address
3    when RAM_START..RAM_MIRROR_END
4      read_ram(address)
5    when PPU_START..PPU_MIRROR_END
6      @ppu.read_register(address)
7    when CONTROLLER_1
8      @controller_1.read
9    when CONTROLLER_2
10     @controller_2.read
11    when PRG_ROM_START..PRG_ROM_END
12     @cartridge.mapper.prg_read(address)
13    else
14      0_u8
15    end
16  end
```

And in `cpu_write`:

```
1  def cpu_write(address : UInt16, value : UInt8)
2    case address
3    when RAM_START..RAM_MIRROR_END
4      write_ram(address, value)
5    when PPU_START..PPU_MIRROR_END
6      @ppu.write_register(address, value)
7    when OAM_DMA
8      oam_dma(value)
9    when CONTROLLER_1
10     @controller_1.write(value)
11     @controller_2.write(value)
12    end
13  end
```

Two things worth noting. First, when the game writes to \$4016, the strobe goes to **both** controllers. That's how the hardware works: the strobe pin is shared. Second, \$4014 is already routed to `oam_dma`, a method we haven't implemented yet but will need very soon.

Finally, the require in nes.cr:

```
1 # src/nest.cr
2
3 require "./nest/controller"
```

OceanofPDF.com

New SDL2 bindings

In chapter 5.2 we set up the minimal SDL2 bindings: window, renderer, texture, and events. But we only handled KEYDOWN and SCANCODE_ESCAPE (to close the window). Now we need to detect when the player **releases** a key (to call `release`) and map specific keys to NES buttons.

In `LibSDL` we add the KEYUP event and the scancodes for the keys we want to map:

```
1 # bin/bindings/sdl2.cr (new constants in LibSDL)
2
3 KEYUP = 0x301_u32
4
5 SCANCODE_RETURN = 40
6 SCANCODE_RSHIFT = 229
7 SCANCODE_Z      = 29
8 SCANCODE_X      = 27
9 SCANCODE_UP     = 82
10 SCANCODE_DOWN  = 81
11 SCANCODE_LEFT  = 80
12 SCANCODE_RIGHT = 79
```

Scancodes are fixed numeric codes that identify keys by their **physical position** on the keyboard, not by the character they produce. That's why they're called "scan" codes: they represent which key was scanned, regardless of the layout. `SCANCODE_Z` is 29 on every keyboard in the world, even though on a French keyboard that position produces "W".

And in the `SDL::Event` wrapper, we add the method to detect key up:

```
1 # bin/bindings/sdl2.cr (new method in SDL::Event)
2
3 def key_up? : Bool
4   @event.type == LibSDL::KEYUP
5 end
```

Symmetric to key_down? that we already had. With these changes, we can detect when each key in our map is pressed and released.

OceanofPDF.com

The keyboard

We have the controllers but nothing is feeding them input. We need to map keyboard keys to NES buttons. In `gui.cr`:

```
1 # bin/gui.cr
2
3 KEY_MAP = {
4   LibSDL::SCANCODE_Z      => NES::Controller::BUTTON_A,
5   LibSDL::SCANCODE_X      => NES::Controller::BUTTON_B,
6   LibSDL::SCANCODE_RSHIFT => NES::Controller::BUTTON_SELECT,
7   LibSDL::SCANCODE_RETURN => NES::Controller::BUTTON_START,
8   LibSDL::SCANCODE_UP     => NES::Controller::BUTTON_UP,
9   LibSDL::SCANCODE_DOWN   => NES::Controller::BUTTON_DOWN,
10  LibSDL::SCANCODE_LEFT    => NES::Controller::BUTTON_LEFT,
11  LibSDL::SCANCODE_RIGHT   => NES::Controller::BUTTON_RIGHT,
12 }
```

Z and X for A and B (the classic emulator layout), arrow keys for the D-pad, Enter for Start, Right Shift for Select. Feel free to tweak the mapping if you prefer something different.

Now the `handle_events` method from chapter 5.2 gets a bit more interesting:

```
1 def handle_events
2   event = SDL::Event.new
3   while event.poll
4     if event.quit? || (event.key_down? && event.scancode ==
LibSDL::SCANCODE_ESCAPE)
5       @running = false
6     end
7
8     if button = KEY_MAP[event.scancode]?
9       if event.key_down?
10        @emulator.bus.controller_1.press(button)
11      elsif event.key_up?
12        @emulator.bus.controller_1.release(button)
13      end
14    end
15  end
16 end
```

```
15   end
16 end
```

The `?` in `KEY_MAP[event.scancode] ?` is Crystal saying “if the key exists in the hash, return the value; otherwise, return nil”. If the pressed key is in our map, we call `press` or `release` on controller 1 depending on whether it’s key down or key up.

Simple and clean. Controller 2 is left unmapped for now. If you want two-player support, you can add another hash with different keys (WASD + Q/E, for example).

OceanofPDF.com

Testing the controller

Before testing with Mario, let's verify the controller works in isolation. A quick test in `bin/test_controller.cr`:

```
1 # bin/test_controller.cr
2
3 require "../src/nes"
4
5 puts "Controller Test"
6 puts "====="
7
8 ctrl = NES::Controller.new
```

First test: press Start, strobe, and read all 8 buttons:

```
1 puts "\n1. Press Start, strobe, read 8 buttons:"
2 ctrl.press(NES::Controller::BUTTON_START)
3
4 ctrl.write(1_u8)
5 ctrl.write(0_u8)
6
7 8.times do |i|
8   bit = ctrl.read
9   names = ["A", "B", "Select", "Start", "Up", "Down", "Left",
"Right"]
10  puts "   Read #{i} (#{names[i]}): #{bit}"
11 end
```

We expect to see 0 for everything except Start (read 3), which should return 1.

Second test: press multiple buttons at once:

```
1 puts "\n2. Press A + Start + Right, strobe, read:"
2 ctrl2 = NES::Controller.new
3 ctrl2.press(NES::Controller::BUTTON_A)
4 ctrl2.press(NES::Controller::BUTTON_START)
5 ctrl2.press(NES::Controller::BUTTON_RIGHT)
```



```

6
7 ctrl2.write(1_u8)
8 ctrl2.write(0_u8)
9
10 8.times do |i|
11   bit = ctrl2.read
12   names = ["A", "B", "Select", "Start", "Up", "Down", "Left",
"Right"]
13   puts "    Read #{i} (#{names[i]}): #{bit}"
14 end

```

A (read 0), Start (read 3), and Right (read 7) should return 1. Everything else 0.

And the third test verifies that it works through the Bus, just like the real game would:

```

1 puts "\n3. Test through Bus (like the real game does):"
2 emu = NES::Emulator.new(ARGV[0])
3 emu.bus.controller_1.press(NES::Controller::BUTTON_START)
4
5 # Strobe: write 1 then 0 to $4016
6 emu.bus.cpu_write(0x4016_u16, 1_u8)
7 emu.bus.cpu_write(0x4016_u16, 0_u8)
8
9 # Read 8 times from $4016
10 8.times do |i|
11   bit = emu.bus.cpu_read(0x4016_u16)
12   names = ["A", "B", "Select", "Start", "Up", "Down", "Left",
"Right"]
13   puts "    Read #{i} (#{names[i]}): #{bit}"
14 end
15
16 puts "\nDone!"

```

This last test is the most important: it creates a full emulator, presses Start on controller 1, and runs the same protocol Mario would use in assembly (write \$01 to \$4016, write \$00, read 8 times). If Start shows up on read 3, the controller is properly wired to the Bus.

```

1 crystal run bin/test_controller.cr

```

The first two tests don't need a ROM. To run the third one (through the full Bus), pass a ROM path:

```
1 crystal run bin/test_controller.cr -- .roms/mario.nes
```

You should see something like this:

```
1  Controller Test
2  =====
3
4  1. Press Start, strobe, read 8 buttons:
5      Read 0 (A): 0
6      Read 1 (B): 0
7      Read 2 (Select): 0
8      Read 3 (Start): 1
9      ...
10
11 2. Press A + Start + Right, strobe, read:
12     Read 0 (A): 1
13     Read 1 (B): 0
14     Read 2 (Select): 0
15     Read 3 (Start): 1
16     ...
17     Read 7 (Right): 1
18
19 3. Test through Bus (like the real game does):
20     ...
21     Read 3 (Start): 1
22     ...
23
24 Done!
```

The shift register returns bits in the correct order, the strobe freezes the snapshot, and the Bus routes reads and writes to the right addresses.

The Moment of Truth

Compile, run:

```
1 crystal build bin/gui.cr -o nes_gui && ./nes_gui .roms/mario.nes
```

The title screen appears. You press Enter.

Nothing happens.

You press it again. Nothing. You mash every key. Nothing. The title screen just sits there, completely unresponsive.

[OceanofPDF.com](https://oceanofpdf.com)

Investigating the problem

Are the controls broken? Let's check. `press` sets the bit, `write` captures the snapshot, `read` returns it bit by bit. The code is correct; it's a textbook shift register. The Bus routes \$4016 correctly. The GUI calls `press` and `release`.

The problem isn't the controls. The problem is that **nobody is reading the controls**.

Wait, how? Mario reads the controls every frame, right? Yes, but to read controls it needs to be running its main loop. And to run the main loop, it needs the NMI to fire. And the NMI...

Let's trace this through. In chapter 5.1 we connected the NMI: when the PPU reaches scanline 241 (VBlank), it calls the callback that fires `cpu.nmi`. That works. We verified it. But let's look more closely at what happens with `PPUCTRL`.

Mario writes to \$2000 (`PPUCTRL`) three times during the first few frames:

1. **Init**: writes \$10, NMI **disabled** (bit 7 = 0). Makes sense; you don't want interrupts firing during initialization.
2. **Init finished**: writes \$90, NMI **enabled** (bit 7 = 1). "I'm ready, let me know when VBlank hits."
3. **Inside the NMI handler**: writes \$10, NMI **disabled** again.

That third write is the key. Mario's NMI handler does this:

1. Disables NMI (writes \$10 to \$2000) to prevent one NMI from interrupting another
2. Does its work: updates sprites, palettes, scroll
3. Before re-enabling NMI, **waits for sprite 0 hit**

And here's the loop where it gets stuck:

```
1 wait:
2     BIT $2002          ; read PPUSTATUS
3     BVS found          ; if bit 6 (sprite 0 hit) = 1, exit
```

```
4    JMP wait          ; otherwise, keep waiting
5 found:
6    ; re-enable NMI and return
```

BIT \$2002 reads PPUSTATUS and copies bit 6 (sprite 0 hit) to the CPU's V flag. BVS jumps if V = 1. If there's no hit, it goes back to BIT \$2002. An infinite loop until sprite 0 hit gets set.

And sprite 0 hit **never** gets set.

What is sprite 0 hit?

It's a timing trick that NES games use to do split-screen. In Mario, sprite 0 (the first sprite in OAM) is placed at the base of the status bar, that line that shows "WORLD 1-1", the coins, and the timer.

When the PPU is rendering and reaches the scanline where sprite 0 overlaps with a non-transparent background pixel, it sets bit 6 of PPUSTATUS. Mario uses this as a signal: "the HUD is done rendering, now you can change the scroll for the gameplay area."

It's a raster trick. Without sprite 0 hit, Mario doesn't know when to change the scroll, so it never re-enables the NMI, so no more NMI handlers fire, so the main loop never runs, so nothing ever reads the controls.

Why doesn't it get set?

Two reasons:

1. **OAM is empty:** the game copies sprites to the PPU's OAM by writing to \$4014 (OAM DMA). We routed that address in the Bus but never implemented the actual copy.
2. **We don't detect the overlap:** even if OAM had data, we have no code that checks whether sprite 0 overlaps with the background.

We need to implement both. We don't need to *draw* sprites yet; we just need them to exist in OAM and for the sprite 0 hit detection to work.

OAM DMA

Back in chapter 5.0 we mentioned that \$4014 is the OAM DMA address. When the game writes a byte to \$4014, the CPU copies 256 bytes from a page in memory into the PPU's OAM. It's the standard way to update sprites.

If the game writes \$02 to \$4014, the CPU reads the 256 bytes from \$0200 to \$02FF and passes them to the PPU. Page \$02 is the most commonly used by convention, but the game can use any page.

In `bus.cr`:

```
1  # src/nes/bus.cr
2
3  # Copy 256 bytes from CPU page $XX00 to the PPU's OAM
4  def oam_dma(page : UInt8)
5    base = page.to_u16 << 8
6    data = Bytes.new(256)
7
8    256.times do |i|
9      data[i] = cpu_read(base &+ i.to_u16)
10   end
11
12   @ppu.oam_dma(data)
13 end
```

`page << 8` converts the page number to a base address: page \$02 gives \$0200. Then we read all 256 bytes with `cpu_read` (not directly from RAM, because the data could be in another component mapped to the bus) and pass them to the PPU.

On the PPU side, receiving them is trivial:

```
1  # src/nes/ppu.cr
2
3  # Copy 256 sprite bytes to OAM
4  def oam_dma(data : Bytes)
```

```
5    data.copy_to(@oam)
6  end
```

A straight copy. The sprites now live in OAM. We're not drawing them yet, but they're there. Mario can write to \$4014 and its sprites get stored.

OceanofPDF.com

Sprite 0 hit

We're not going to draw sprites yet; that's next chapter. But we need to detect whether sprite 0 overlaps with the background. It's an invisible check: we test for overlap, set the flag, and the game carries on.

The flags in PPUSTATUS

In `status.cr` we already had `FLAG_VBLANK`. Let's add sprite 0 hit:

```
1 # src/nes/ppu/registers/status.cr
2
3 FLAG_SPRITE_0_HIT = 0x40_u8
4 FLAG_VBLANK       = 0x80_u8
5
6 def set_sprite_0_hit
7   @status |= FLAG_SPRITE_0_HIT
8 end
9
10 def clear_sprite_0_hit
11   @status &= (~FLAG_SPRITE_0_HIT).to_u8
12 end
```

Bit 6 of PPUSTATUS. `set_sprite_0_hit` turns it on, `clear_sprite_0_hit` turns it off. The clear gets called on the pre-render scanline (261) to reset the flag at the start of each frame.

The detection

The logic lives in `display.cr`. After rendering the background scanline, we check for sprite 0 overlap:

```
1 # src/nes/ppu/display.cr (in render_scanline, one new line)
2
3 def render_scanline(y : Int32)
4   WIDTH.times do |pixel|
5     bg_color, bg_palette = if show_background?
6       background_pixel(pixel)
```

```

7     else
8         {0, 0}
9     end
10
11     set_pixel(pixel, y, palette_color(bg_palette, bg_color))
12 end
13
14 check_sprite_0_hit(y) if show_background? && show_sprites?
15 end

```

We only check when both background and sprites are enabled in PPUMASK. If either is off, there can't be an overlap.

Now the method itself:

```

1  # src/nes/ppu/display.cr
2
3  def check_sprite_0_hit(y : Int32)
4      return if (@status & FLAG_SPRITE_0_HIT) != 0 # Already hit this
frame
5
6      sprite_y = @oam[0].to_i32 + SPRITE_Y_OFFSET
7      return if y < sprite_y || y >= sprite_y + PIXELS_PER_TILE
8
9      sprite_tile = @oam[1]
10     sprite_attr = @oam[2]
11     sprite_x     = @oam[3].to_i32
12
13     row = y - sprite_y
14     row = (PIXELS_PER_TILE - 1) - row if (sprite_attr & 0x80) != 0
# Vertical flip
15
16     PIXELS_PER_TILE.times do |col|
17         screen_x = sprite_x + col
18         next if screen_x >= 255 # Sprite 0 hit is not detected at
x=255
19
20         flip_col = (sprite_attr & 0x40) != 0 ? (PIXELS_PER_TILE - 1)
- col : col
21         sprite_color = sprite_tile_pixel(sprite_tile, row, flip_col)
22         next if sprite_color == 0 # Transparent sprite pixel
23
24         bg_color, _ = background_pixel(screen_x)
25         next if bg_color == 0 # Transparent background pixel

```

```
26
27     set_sprite_0_hit
28     return
29 end
30 end
```

Step by step:

1. **Early return if already hit:** sprite 0 hit gets set once per frame. If it's already set, nothing to do.
2. **Read sprite 0 from OAM:** the first 4 bytes of OAM are sprite 0. Byte 0 = Y, byte 1 = tile index, byte 2 = attributes (flip, priority, palette), byte 3 = X. Note the + `SPRITE_Y_OFFSET`: the NES has a quirk where Y = 0 in OAM actually means scanline 1, so we always add 1.
3. **Check if the scanline intersects the sprite:** if the current scanline is outside the range `[sprite_y, sprite_y + 8)`, the sprite isn't on this line.
4. **Calculate the row within the tile:** `y - sprite_y` gives the row (0-7). If the sprite has vertical flip (bit 7 of attributes), we invert: row 0 becomes 7, row 1 becomes 6, etc.
5. **Iterate over the 8 sprite pixels:** for each column, we figure out the screen position. There's a quirk: sprite 0 hit is not detected at `x = 255` (the last pixel on screen). Nobody knows why, but that's what the real PPU does.
6. **Horizontal flip:** if bit 6 of attributes is set, we invert the column.
7. **Check opacity:** we read the sprite pixel from CHR-ROM. If it's 0 (transparent), skip. Then we read the background pixel at the same position. If that's also 0, skip. If **both are opaque**, we have overlap: set the flag and bail out.

Reading the sprite pixel

`sprite_tile_pixel` is practically identical to `tile_pixel` that we already had for the background:

```
1 def sprite_tile_pixel(tile_index : UInt8, tile_row : Int32,
pixel_in_tile : Int32) : Int32
2   bitplane_address = sprite_pattern_table.to_i32 +
tile_index.to_i32 * BYTES_PER_TILE + tile_row
3   low_bitplane      = vram_read(bitplane_address.to_u16)
4   high_bitplane     = vram_read((bitplane_address +
BITPLANE_OFFSET).to_u16)
5
6   bit              = (PIXELS_PER_TILE - 1) - pixel_in_tile
7   low_bit          = (low_bitplane >> bit) & 1
8   high_bit         = (high_bitplane >> bit) & 1
9   ((high_bit << 1) | low_bit).to_i32
10 end
```

The only difference from `tile_pixel` is that it uses `sprite_pattern_table` instead of `background_pattern_table`. Sprites can live in a different pattern table from the background. Mario uses table 0 for the background and table 1 for sprites, controlled by `PPUCTRL` (bit 3 for sprites, bit 4 for background).

Clearing the flag

In `advance_cycle`, when we hit the pre-render scanline (261), we clear both `VBlank` and `sprite 0 hit`:

```
1 # src/nes/ppu.cr (in advance_cycle)
2
3 elsif @scanline == PRE_RENDER_SCANLINE
4   clear_vblank
5   clear_sprite_0_hit
```

We already had `clear_vblank`. `clear_sprite_0_hit` is new. At the start of each frame, the flag resets so it can be detected again in the next frame.

OceanofPDF.com

The Second Attempt

Compile again:

```
1 crystal build bin/gui.cr -o nes_gui && ./nes_gui .roms/mario.nes
```

The title screen appears. You press Enter.

And the background changes.

The “1 PLAYER GAME” screen disappears and the level loads in. Blue sky, clouds, brick floor, question mark blocks, green pipes. It’s World 1-1.

But there’s no Mario. No Goombas. No spinning coins. Everything that moves independently of the background is a sprite, and we’re not drawing those yet, only detecting them.

Press the right arrow. The background scrolls. You’re walking. You can’t see who’s walking, but the world is moving. Press Z (button A) and you jump. Stand under a question block and jump, the block changes tile. Fall into a pit, the game resets.

You’re playing blind.



Figure 10. Mario Bros (Without sprites)

Why did it work?

Let's trace the causal chain, because it's beautiful:

1. **OAM DMA** copies 256 bytes of sprites to the PPU's OAM
2. **Sprite 0 exists in OAM**: Mario placed it at the base of the status bar
3. **check_sprite_0_hit** detects that sprite 0 overlaps with a background pixel
4. **PPUSTATUS bit 6** gets set
5. **The NMI handler** sees the flag via BIT \$2002, exits the loop with BVS
6. **Re-enables NMI** (writes \$90 to \$2000)
7. **Next frame**: the PPU reaches VBlank, fires NMI normally
8. **The main loop runs**: reads controls, processes input, updates the game
9. **The game is alive**

All because of a single bit. One flag in a status register that says “yes, the sprite and the background overlapped on this scanline.” Without it, the NMI handler waits forever, the NMI never gets re-enabled, and the game freezes in an infinite loop. With it, the chain completes and everything works.

OceanofPDF.com

Challenge: Playing Blind

How far can you get in World 1-1 without seeing Mario? You know he starts at the bottom left. Walk right. When you see a pipe coming up in the background, jump. If the background stops scrolling, you probably died and need to wait for the game to restart.

It's harder than it sounds. But the game is playable. And that alone is an achievement.

OceanofPDF.com

Wrap up

We covered a lot of ground in this chapter, and half of it was a debugging mystery:

- **controller.cr**: 8 buttons, shift register, strobe protocol. 45 lines.
- **bus.cr**: routing for \$4014 (OAM DMA), \$4016/\$4017 (controllers), creation of both controllers
- **gui.cr**: KEY_MAP to map keyboard keys to NES buttons, handle_events updated to call press/release
- **status.cr**: FLAG_SPRITE_0_HIT, set_sprite_0_hit, clear_sprite_0_hit
- **display.cr**: check_sprite_0_hit (invisible overlap detection), sprite_tile_pixel
- **ppu.cr**: oam_dma to receive sprites, clear_sprite_0_hit on the pre-render scanline

The most interesting takeaway here isn't technical; it's about how tightly coupled the NES hardware systems are. You can't have working controls without OAM DMA, you can't get past OAM DMA without sprite 0 hit, and you can't understand sprite 0 hit without knowing how Mario uses the NMI. Every piece depends on the others.

The full code for this chapter is available in the [repo](#).

Next chapter, we draw sprites. Mario appears.

OceanofPDF.com

Chapter 5.5: Sprites

In the previous chapter we were playing Mario blind. The background scrolled, blocks broke, but we couldn't see Mario, the Goombas, or the coins. Everything that moves independently from the background is a sprite, and we weren't drawing any of them.

That changes now. By the end of this chapter, Mario shows up.

OceanofPDF.com

Anatomy of a sprite

We already know that OAM holds 256 bytes for 64 sprites, 4 bytes each. In the previous chapter we read those bytes to detect sprite 0 hit, but we didn't go into detail. Let's fix that:

-
- 1 Byte 0: Y position (screen Y position, with a +1 offset)
 - 2 Byte 1: Tile index (which tile from the pattern table to draw)
 - 3 Byte 2: Attributes (palette, priority, flips)
 - 4 Byte 3: X position (screen X position)
-

Byte 2 (attributes) packs several fields together:

-
- 1 76543210
 - 2 | | | | | L L — Palette (0-3, maps to sprite palettes 4-7)
 - 3 | | | L L L — Unused
 - 4 | | — Priority (0 = in front of background, 1 = behind)
 - 5 | — Horizontal flip
 - 6 — Vertical flip
-

Priority is the most interesting part. A sprite with priority 0 draws *on top of* the background. One with priority 1 draws *behind* it, only visible where the background is transparent. Mario uses this so the character walks behind pipes, and so coins inside question blocks appear behind the block.

The constants

Let's add the attribute byte constants to `display.cr`:

```
1 # src/nes/ppu/display.cr (new constants)
2
3 SPRITE_PALETTE_MASK = 0x03_u8 # Bits 0-1: palette (0-3)
4 SPRITE_BEHIND_BG    = 0x20_u8 # Bit 5: priority (0=front,
1=behind)
5 SPRITE_FLIP_H       = 0x40_u8 # Bit 6: horizontal flip
6 SPRITE_FLIP_V       = 0x80_u8 # Bit 7: vertical flip
7 SPRITE_PALETTE_OFFSET = 4     # Sprite palettes are 4-7
(background palettes are 0-3)
8 MAX_SPRITES_PER_SCANLINE = 8
```

`SPRITE_PALETTE_OFFSET` is key: sprites use palettes 4, 5, 6, and 7, which live at `$3F10-$3F1F`. Palettes 0-3 belong to the background (`$3F00-$3F0F`). The attribute byte says “palette 0” but it’s really palette 4 (0 + offset). That’s how the hardware keeps background and sprite palettes separate without the programmer lifting a finger.

`MAX_SPRITES_PER_SCANLINE` is a hardware limitation: the PPU only evaluates 8 sprites per scanline. If there are more than 8 sprites on the same horizontal line, the extras disappear. It’s a famous limitation: if too many enemies bunch up in Mario, some of them flicker. Games rotate the evaluation order each frame to spread the flickering around instead of always dropping the same sprites.

Finding the winning sprite

For each pixel on screen, we need to figure out if there's a visible sprite there, and if so, grab its color, palette, and priority. When multiple sprites overlap, the one with the lowest OAM index wins (the one that comes first in the table).

```
1  # src/nes/ppu/display.cr
2
3  # Finds the first opaque sprite at position (x, y).
4  # Returns {color, palette, behind_background}
5  def sprite_pixel(x : Int32, y : Int32) : Tuple(Int32, Int32,
Bool)
6    count = 0
7
8    SPRITES_TOTAL.times do |i|
9      break if count >= MAX_SPRITES_PER_SCANLINE
10
11      offset = i * SPRITE_BYTES
12      sprite_y = @oam[offset].to_i32 + SPRITE_Y_OFFSET
13      next unless y >= sprite_y && y < sprite_y + PIXELS_PER_TILE
14      count += 1
15
16      sprite_x = @oam[offset + 3].to_i32
17      next unless x >= sprite_x && x < sprite_x + PIXELS_PER_TILE
18
19      sprite_tile = @oam[offset + 1]
20      sprite_attr = @oam[offset + 2]
21
22      row = y - sprite_y
23      col = x - sprite_x
24      row = (PIXELS_PER_TILE - 1) - row if (sprite_attr &
SPRITE_FLIP_V) != 0
25      col = (PIXELS_PER_TILE - 1) - col if (sprite_attr &
SPRITE_FLIP_H) != 0
26
27      color = sprite_tile_pixel(sprite_tile, row, col)
28      next if color == 0
29
30      palette = (sprite_attr & SPRITE_PALETTE_MASK).to_i32
31      behind = (sprite_attr & SPRITE_BEHIND_BG) != 0
32      return {color, palette, behind}
```

```
33     end
34
35     {0, 0, false}
36 end
```

Let's walk through the method step by step.

We iterate over all 64 sprites in OAM order (0 to 63). For each one:

1. **8 per scanline limit:** if we've already counted 8 sprites on this scanline, we stop looking. Notice that `count` increments when the sprite falls in the Y range, *before* checking X. That's how the hardware works: a sprite counts toward the limit even if it's offscreen horizontally.
2. **Y check:** `@oam[offset]` is the sprite's Y position (with the +1 quirk we already covered). If the current scanline doesn't intersect the sprite's vertical range (8 pixels tall), we skip it.
3. **X check:** if the current pixel doesn't fall within the sprite's horizontal range, we skip it.
4. **Flips:** if the attribute byte has vertical or horizontal flip set, we invert the row or column within the tile. We did the same thing in `check_sprite_0_hit`, so it should look familiar.
5. **Read the pixel:** `sprite_tile_pixel` reads the bitplanes from CHR-ROM, just like `tile_pixel` but using `sprite_pattern_table` instead of `background_pattern_table`.
6. **Transparency:** if the color is 0, the pixel is transparent and we keep looking at the next sprite.
7. **First opaque sprite wins:** we return the color (0-3), the palette (0-3 from the attribute, which later gets converted to 4-7), and whether it's behind the background.

If none of the 64 sprites have an opaque pixel at this position, we return `{0, 0, false}`, meaning "no sprite here."

The sprite palette

Sprite palettes live at \$3F10-\$3F1F, right after the background palettes (\$3F00-\$3F0F). The method is almost identical to `palette_color` but with the offset of 4:

```
1  # src/nes/ppu/display.cr
2
3  # Sprite palettes are at $3F10-$3F1F (palettes 4-7)
4  def sprite_palette_color(palette : Int32, color : Int32) : UInt8
5    if color == 0
6      vram_read(PALETTE_START)
7    else
8      vram_read((PALETTE_START + (palette + SPRITE_PALETTE_OFFSET)
9        * COLORS_PER_PALETTE + color).to_u16)
10    end
11  end
```

If the color is 0, it returns the universal background color (\$3F00), same as for the background. Otherwise, it indexes into the appropriate palette: sprite palette 0 is global palette 4, which starts at \$3F10. Palette 1 at \$3F14, and so on.

A quick example: Mario uses sprite palette 0 with red/brown/beige colors. His sprite's attribute byte says "palette 0". We calculate $\$3F00 + (0 + 4) * 4 + \text{color} = \$3F10 + \text{color}$. If the pixel color is 2 (the third color in the palette), we read \$3F12, which holds whatever the game wrote to VRAM during initialization.

The priority logic

Now for the part that ties everything together. The `render_scanline` from chapter 5.3 only drew the background. Now it also needs to figure out the sprite color and decide which one actually gets displayed:

```
1  # src/nes/ppu/display.cr
2
3  def render_scanline(y : Int32)
4    WIDTH.times do |pixel|
5      bg_color, bg_palette = if show_background?
6        background_pixel(pixel)
7      else
8        {0, 0}
9      end
10
11      spr_color, spr_palette, spr_behind = if show_sprites?
12        sprite_pixel(pixel, y)
13      else
14        {0, 0, false}
15      end
16
17      # Priority: sprite wins if opaque AND (background transparent
OR sprite is in front)
18      final_color = if spr_color != 0 && (bg_color == 0 ||
!spr_behind)
19        sprite_palette_color(spr_palette, spr_color)
20      else
21        palette_color(bg_palette, bg_color)
22      end
23
24      set_pixel(pixel, y, final_color)
25    end
26
27    check_sprite_0_hit(y) if show_background? && show_sprites?
28  end
```

For each pixel, we get both the background color and the sprite color. Then we pick a winner. The rule is:

- **Opaque sprite + transparent background:** sprite wins (always, regardless of priority)
- **Opaque sprite + opaque background + sprite in front (priority=0):** sprite wins
- **Opaque sprite + opaque background + sprite behind (priority=1):** background wins
- **Transparent sprite:** background wins (always)

In one line: the sprite wins if it's opaque AND (the background is transparent OR the sprite is in front). Everything else shows the background.

When does “behind” priority actually matter? Think about Mario entering a pipe: Mario's sprite has “front” priority, but coins inside question blocks have “behind” priority, so they appear behind the block before flying out.

OceanofPDF.com

Compiling with `--release`

One important heads-up: with sprites, the emulator does way more work per frame. For each pixel, it potentially iterates over all 64 sprites looking for an opaque one. That's up to $64 * 256 = 16,384$ checks per scanline, times 240 scanlines. In debug mode (`crystal run`), this is painfully slow. Always compile with `--release`:

```
1 crystal build bin/gui.cr -o nes_gui --release
2 ./nes_gui .roms/mario.nes
```

The difference is night and day. `--release` enables LLVM optimizations and the emulator hums along at 60fps. Without it, you might get 5-10fps.

OceanofPDF.com

The moment of truth

You press Enter on the title screen.

And there he is. Mario. Standing in World 1-1, small, facing right. A Goomba walking toward him. Above, the question blocks with their bouncing animation. Coins spinning when you hit them. The mushroom popping out of the block. Mario growing big.

Everything works. Sprites show up in the right colors, flips work (Mario faces the direction he's walking), priority works (Mario goes behind pipes), and the 8 sprites per scanline limit keeps the screen from filling with garbage.



Figure 11. Mario Bros (Completo, con Sprites)

Wrap up

This chapter was short because the heavy lifting was already done. OAM DMA, sprite 0 hit, and `sprite_tile_pixel` all came from chapter 5.4. Here we only added:

- **sprite_pixel**: iterates over all 64 sprites in OAM order, finds the first opaque pixel, respecting the 8 per scanline limit and the attribute byte flips
- **sprite_palette_color**: reads from palettes 4-7 (\$3F10-\$3F1F) instead of 0-3
- **render_scanline**: calculates background AND sprite, applies the priority logic, and draws the winner
- **Attribute byte constants**: `SPRITE_PALETTE_MASK`, `SPRITE_BEHIND_BG`, `SPRITE_FLIP_H`, `SPRITE_FLIP_V`

The full rendering pipeline now looks like this:

```
1 For each scanline (0-239):
2   For each pixel (0-255):
3     1. Calculate background color (nametable + tile + bitplanes +
attribute + palette)
4     2. Find opaque sprite (iterate OAM, check Y/X, flips, read
bitplanes)
5     3. Decide priority (sprite vs background)
6     4. Write final pixel to frame buffer
7   Check sprite 0 hit
```

You can see the full code for this chapter in the [repo](#).

Mario is playable. You can jump, run, break blocks, grab mushrooms, stomp Goombas. Scrolling works, the HUD shows the score. It's a working NES emulator.

Take a second to think about what we built. We started with a loop that read opcodes from a byte array. Then we parsed a real cartridge. Then we opened a window. And now you've got an emulator running Super Mario Bros, with graphics, scrolling, controls, sprites, and rendering priority. All from scratch.

The emulator is 90% there. The only thing missing is sound (the APU), which we'll cover in the next chapters. And in an appendix we'll add Mapper 1 (MMC1), which opens the door to Zelda, Metroid, and Mega Man 2. But that's bonus content. The core is done: the CPU, the bus, the cartridge, the full PPU with background and sprites, scrolling, controls. A working NES emulator that you wrote yourself.

Not bad for a project that started with 0xA9_u8.

OceanofPDF.com

Chapter 6: Let's Play

We've got a working NES emulator. Full CPU, PPU with background and sprites, scroll, controls. Time to test it with something beyond Super Mario Bros.

OceanofPDF.com

Getting ROMs

Back in chapter 4.1 we covered where to get ROMs:

- [Emu-Land](#)
- [Retrostic](#)
- [RomsFun](#)

Download the .nes files and drop them in the .roms/ folder of the project. Then run the emulator with:

```
1 crystal build bin/gui.cr -o nes_gui --release
2 ./nes_gui .roms/game_name.nes
```

Always compile with `--release`. Without it, the PPU is way too slow for 60fps.

OceanofPDF.com

Mapper 0 Games

Our emulator supports Mapper 0 (NRROM), the simplest mapper: no bank switching, 32KB of PRG-ROM max, and 8KB of CHR-ROM. It's what the NES launch titles used, the smallest and most straightforward games. These should all work:

Super Mario Bros. Our benchmark since chapter 5.3. If you've made it this far, you already know it works. But now with sprites, you can actually play it for real.

Donkey Kong. The game that created Mario (well, "Jumpman"). Three screens, a gorilla, barrels. Simple and addictive. A good test since it uses few sprites and the rendering is straightforward.

Ice Climber. A vertical platformer where you climb a mountain by breaking ice blocks. If you've played Smash Bros, you know Popo and Nana. This is where they were born.

Balloon Fight. Nintendo's take on Joust. You fly around with balloons and try to pop your enemies' balloons. Smooth movement, and a nice test for sprite collisions.

OceanofPDF.com

Mapper 1 Games

These games use Mapper 1 (MMC1), which adds bank switching: up to 256KB of PRG-ROM and 128KB of CHR-ROM, plus configurable mirroring and SRAM for save files. Bigger, more ambitious titles:

The Legend of Zelda. One of the first games with a battery for saving your progress. Non-linear exploration, a huge overworld, 9 dungeons. An absolute classic.

Metroid. Sci-fi exploration on planet Zebes. Samus Aran hunts metroids in a giant labyrinth. It defined an entire genre (the “Metroid” in “Metroidvania”).

Mega Man 2. Widely considered the best NES game ever. Eight Robot Masters, stolen weapons, legendary music. If your emulator runs this, it can handle almost anything.

Final Fantasy. The RPG that saved Square from bankruptcy (or so the legend goes). Turn-based combat, magic, four warriors of light.

Blaster Master. An action game where you alternate between driving a tank and exploring on foot. A technical showpiece for its time, with multidirectional scrolling and two distinct gameplay styles.

OceanofPDF.com

Try a Mapper 1 Game

Before moving on, go ahead and try it. Download any of the Mapper 1 games listed above and run the emulator:

```
1 ./nes_gui .roms/zelda.nes
```

The result:

```
1 Mapper 1 not implemented (RuntimeError)
```

It crashes before it even gets going. When we parse the cartridge in `Cartridge`, we read the mapper ID from the header. If it's not 0, `Mapper.create` throws an exception:

```
1 # src/nes/cartridge/mapper.cr
2
3 def self.create(id : UInt8, prg_rom : Bytes, chr_rom : Bytes) :
Mapper
4   case id
5   when 0 then Mapper000.new(prg_rom, chr_rom)
6   else raise "Mapper #{id} not implemented"
7   end
8 end
```

Fair enough: Mapper 1 needs bank switching logic we haven't written yet. When the CPU writes to the \$8000-\$FFFF range, Mapper 1 interprets those writes as commands to switch which ROM bank is mapped. Without that logic, the game can't access more than 32KB of code, and everything falls apart.

Mapper 1 and Beyond

The book's appendix has the full Mapper 1 (MMC1) implementation. It's a great exercise: it brings together everything you've learned about the bus, the cartridge, and the PPU. The interface is already defined (`prg_read`, `chr_read`, `chr_write`), so you just need to create `mapper_001.cr`, implement the 5-bit shift register and bank switching logic, and add the `when 1` case in `Mapper.create`.

Mapper 0 and Mapper 1 together cover a huge chunk of the NES library. But there are over 200 documented mappers. Each one is a different chip with its own bank switching logic. Mapper 2 (UxROM) is almost as simple as 0, and it unlocks Castlevania, Contra, and Duck Tales. Mapper 4 (MMC3) is more complex but covers Super Mario Bros. 3, Kirby's Adventure, and Mega Man 3-6.

Adding new mappers is the natural next step after finishing this book. The structure is already in place: create a `Mapper` subclass, implement `prg_read`/`chr_read`/`chr_write`, and hook it up in `Mapper.create`. Great for community contributions, or just for tinkering on your own.

OceanofPDF.com

Wrap up

We've got an emulator that runs Super Mario Bros. from start to finish. You can jump, run, break blocks, grab mushrooms, stomp Goombas, and reach the castle. The background scrolls, the HUD shows the score, and sprites render with correct priority.

Now's a good time to just play. Try the Mapper 0 games, break things, hunt for bugs. If something looks off, dig into it. That's the best way to understand how the NES works under the hood.

But something's missing. You boot up Mario and there's silence. You jump, no sound. You grab a coin, nothing. The game works, but it feels hollow without audio.

In the next chapter, we give the emulator a voice: the APU.

OceanofPDF.com

Chapter 7: The APU

You fire up Mario. You see him jump, stomp Goombas, grab coins. But there's nothing but silence. Mario without music isn't Mario. That iconic World 1-1 melody, the coin "bling", the jump "boing"... all missing.

The NES has an audio chip: the **APU** (Audio Processing Unit). It lives inside the same chip as the CPU (the Ricoh 2A03), and it's a synthesizer that generates sound from scratch using four channels.

But before we dive into the NES, we need to talk about something most programmers have never had to think about: how sound actually works.

OceanofPDF.com

What Is Sound

Sound is vibrating air. A speaker pushes air back and forth, creating pressure waves. Those waves travel to your ear, vibrate your eardrum, and your brain interprets them as sound.

Three properties define a sound:

Frequency: how many times per second the air vibrates. Measured in Hertz (Hz). 440 Hz = 440 vibrations per second = the note A (the one used to tune instruments). More Hz = higher pitch. Less Hz = lower pitch. The human ear hears between ~20 Hz and ~20,000 Hz.

Amplitude: how hard the speaker pushes. More amplitude = louder. Less amplitude = quieter.

Waveform: the “shape” of the vibration. It’s what makes a piano and a guitar sound different even when playing the same note at the same frequency and volume. A piano has one waveform, a guitar has another, and the NES has its own.

OceanofPDF.com

Digital Waves

A computer can't produce infinitely smooth vibrations like an acoustic instrument. Instead, it generates a sequence of **samples**: numbers representing the speaker's position at a given moment.

Imagine the speaker can be anywhere between -1.0 (all the way back) and +1.0 (all the way forward). A sample is a number in that range. If you send 44,100 samples per second to the speaker (that's what "44.1 kHz" means, CD quality), each sample tells the speaker where to be for a fraction of a second.

1	Sample 1:	0.0	(speaker at center)
2	Sample 2:	0.5	(speaker slightly forward)
3	Sample 3:	1.0	(speaker at max)
4	Sample 4:	0.5	(coming back)
5	Sample 5:	0.0	(at center again)
6	Sample 6:	-0.5	(slightly back)
7	Sample 7:	-1.0	(all the way back)
8	Sample 8:	-0.5	(coming back)
9	Sample 9:	0.0	(completed one cycle)
10	...		

If that sequence repeats, the speaker moves back and forth smoothly. That's a **sine wave**: the purest sound there is. A clean "tooooo" with no character, like a TV test tone.

Waveforms

Things get interesting when you change the shape. What if instead of moving back and forth smoothly, the speaker jumps abruptly between two positions?

1 Square wave:

2 — — —
3 | | | | |
4 | | | | |
5 |_ |_ |_ |_

That's a **square wave**: it sits at one level, snaps to the other, and snaps back. No smooth transitions. The sound is harsh, “electronic”, instantly recognizable as 8-bit. It's the NES's signature sound.

What if instead of jumping between two levels, you go up and down in a straight line?

1 Triangle wave:

2 / \ / \ / \
3 / \ / \ / \
4 / \ / \ / \
5 / \ \ \

That's a **triangle wave**. It sounds softer than the square wave, more “rounded.” The NES uses it for bass lines. That bass in the Mario theme? Triangle wave.

And what if you generate random values?

1 Noise:

2 _ _ _ _ _
3 | | | | | | | |
4 _ | |_ | | | | |_

That's **noise**. It has no defined frequency and doesn't sound like a musical note. It sounds like static, like sand, like a hi-hat or a snare drum. The NES uses it for percussion.

Those three waveforms (square, triangle, and noise) are all the NES has to work with. And somehow that was enough to produce some of the most memorable melodies in video game history.

OceanofPDF.com

Audio Synthesis

The NES doesn't play audio files. There are no MP3s or WAVs stored anywhere. It's a **synthesizer**: it generates waves in real time by tweaking parameters.

Modern synthesizers have thousands of parameters. The NES has few, and that's why the sound is so distinctive:

- **Frequency**: how fast the wave repeats (the musical note)
- **Volume**: how loud it sounds (0-15, four bits)
- **Duty cycle**: for square waves, what percentage of the cycle is "high" (changes the timbre)
- **Duration**: how long the note lasts before going silent
- **Modulation**: automatically changing the frequency over time (for effects like the "wooop" of Mario's flagpole)

The game doesn't say "play the coin sound." It says "pulse 1 channel: high frequency, volume 12, 50% duty, short duration, sweep upward." And the NES generates that sound in real time. If you listen closely, the coin "bling" is literally a note that quickly rises in pitch and then goes silent.

OceanofPDF.com

The NES Channels

The APU has five channels. We'll implement four, and skip one:

Pulse 1 and Pulse 2: The Melody

Two square wave channels. These are the workhorses of NES audio, handling the main melodies, harmonies, and most sound effects.

Why two? Because you can't do harmonies with just one. Think of the Mario theme: there's a lead voice (the melody you'd hum) and a second voice that accompanies it. Each one gets its own pulse channel.

Each pulse has a configurable **duty cycle**. The duty cycle is what percentage of the wave is "high":

1	12.5%:	_X_____	(high only 1/8 of the time, nasal, thin sound)
2	25%:	_XX_____	(high 2/8, the most common, clear sound)
3	50%:	_XXXX_	(high half the time, hollow sound, flute-like)
4	75%:	X____XX	(= inverted 25%, sounds the same as 25%)

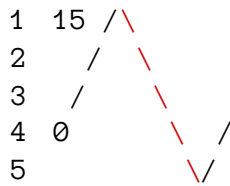
Same frequency, same volume, but the duty cycle completely changes the character of the sound. 25% is the quintessential NES sound. If you hear something and think "that's 8-bit," it's probably a pulse at 25%.

The two pulse channels are nearly identical. The only difference is a minor quirk in how they calculate the sweep (more on that later).

Triangle: The Bass

A triangle wave with fixed volume. You can't control how loud it is: it's either on at full blast or silent. Sounds like a harsh limitation, but it works fine because the triangle is used almost exclusively for bass, and bass on the NES doesn't need fade-out.

The triangle has a 32-step sequence that goes from 0 to 15 and back down from 15 to 0:



It sounds softer and more “rounded” than the pulses. Listen to the Mario theme and you’ll hear the melody (pulses) with a bass line running underneath (triangle). That’s three of the four channels accounted for.

Noise: The Percussion

Generates pseudorandom noise using a circuit called an LFSR (Linear Feedback Shift Register). It doesn’t produce musical notes; it produces texture: snares, hi-hats, explosions, the sound of Mario dying.

It has two modes:

- **Long mode:** white noise, sounds like TV static. A 32,767-step sequence before it repeats. Used for snares and explosions.
- **Short mode:** metallic noise, sounds tonal, almost like a note but gritty. Only 93 steps in the sequence. Used for special effects.

The noise frequency isn’t freely configurable. There are 16 fixed preset periods. The game picks an index from 0 to 15 and the NES uses the corresponding period.

DMC: The One We Skip

The Delta Modulation Channel plays 1-bit compressed audio samples. Each bit says “raise” or “lower” the output level, encoding a rough waveform. Some games use it for voices, realistic drum samples, or special effects.

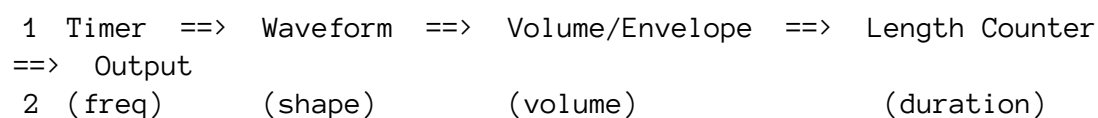
We’re not going to implement it. With the four main channels, Mario sounds complete: all the music and sound effects that matter are there. You’d only

be missing a handful of specific samples.

OceanofPDF.com

Anatomy of a Channel

All channels share a similar structure. Think of them as modules wired in series, where each one can silence or shape the signal:



Timer: The Frequency

Controls how fast the waveform advances, which sets the frequency of the sound (the musical note).

It's a counter that ticks down every cycle. When it hits 0, it advances one step in the waveform sequence and reloads with the **period**. Low period = counts fast = high frequency = high pitch. High period = counts slow = low frequency = low pitch.

The period is 11 bits for the pulses and triangle (values 0-2047), and comes from a fixed table of 16 values for the noise.

Here's a confusing bit: the pulses and noise clock their timer every **2 CPU cycles** (every "APU cycle"). The triangle clocks every **1 CPU cycle**. So the triangle runs at double speed, which lets it reach higher frequencies.

Envelope: The Volume

Controls how the volume evolves over time. It has two modes:

Constant: the game writes a value from 0 to 15 and that's the volume. Simple as that.

Decaying: starts at 15 and drops one notch at a time (~240 times per second). Creates a natural fade-out. When it hits 0, it either stays there (note dies quietly) or loops back to 15 (for sustained sounds).

The decay speed is controlled by a "divider": a second counter that determines how many quarter frames pass between each decay step. If the

divider is 0, the volume drops very fast. If it's 15, it drops slowly.

The pulses and noise have envelopes. The triangle doesn't: it's always at full volume.

Length Counter: The Duration

Controls how long a note lasts. When the game starts a note, it loads the length counter with a value from a lookup table of 32 durations:

1	10,	254,	20,	2,	40,	4,	80,	6,	160,	8,	60,	10,	14,	12,	26,	14,
2	12,	16,	24,	18,	48,	20,	96,	22,	192,	24,	72,	26,	16,	28,	32,	30

Why a table instead of a raw value? The register only has 5 bits for duration (the other 3 go to the timer). Five bits gives you 32 possible values (0-31). Using them directly would give you durations that are way too short. The table maps those 32 indices to musically useful durations: from 2 frames (basically a click) to 254 frames (~4 seconds, a long sustained note).

The counter decrements every half frame (~120 times per second). When it reaches 0, the channel goes silent. If `length_halt` is enabled, the counter doesn't decrement and the note plays indefinitely until the game manually cuts it.

All four channels have it.

Linear Counter: Triangle Only

The triangle has a second duration counter. It takes a direct 7-bit value (0-127) and decrements every quarter frame (~240Hz). That gives it finer timing resolution than the length counter. Both counters have to be > 0 for the triangle to produce sound.

Why does the triangle need two duration counters? The length counter handles long durations ("this note lasts X half frames"). The linear counter enables short, precise notes at double the resolution. For a bass line that needs snappy articulation, the linear counter is essential.

Sweep: Pulses Only

The sweep automatically shifts the frequency over time. It takes the current timer period, shifts it right by N bits, and adds or subtracts the result. Subtracting makes the frequency go up (period shrinks). Adding makes it go down (period grows).

This produces some of the NES's most recognizable effects: the ascending "woop" when Mario grabs the flagpole, the descending sound of the pipes, the "power up" jingle of the mushroom.

The sweep clocks every half frame (~120Hz) and has its own divider to control the speed of the effect.

Here's a hardware quirk: pulse 1 and pulse 2 calculate sweep negation differently. Pulse 1 uses one's complement (subtracts the shifted value, then subtracts 1 more). Pulse 2 uses two's complement (just subtracts the shifted value). The difference is off by 1. Nobody knows why Nintendo did this, probably just a chip layout artifact. But we have to replicate it, because otherwise some notes get silenced when they shouldn't.

The sweep also handles **muting**: if the current timer period is less than 8, or if the target period (what the sweep would produce) exceeds 0x7FF, the channel goes silent. This prevents the channel from producing unstable or ultrasonic frequencies.

The Frame Counter

The envelopes, length counters, linear counter, and sweeps don't tick every CPU cycle. That would be way too fast. They run at slower rates, governed by the **frame counter**.

The frame counter is an internal APU timer that divides the CPU clock into two signals:

- **Quarter frame** (~240Hz): clocks envelopes and the triangle's linear counter
- **Half frame** (~120Hz): clocks length counters and sweeps

There are two modes:

Mode 0 (4-step): runs through 29,829 CPU cycles, synced to the framerate (60Hz). The 4 steps fire at cycles 7457, 14913, 22371, and 29829. Steps 2 and 4 clock both quarter and half frame. Steps 1 and 3 clock quarter frame only.

Mode 1 (5-step): runs through 37,281 CPU cycles (48Hz). Has a fifth step. Less commonly used, but some games prefer it because it avoids an IRQ that mode 0 generates (which we don't implement).

The mode is configured by writing to \$4017. Most games use mode 0. When you write to \$4017 with mode 1, it immediately clocks quarter + half frame. Mode 0 doesn't do that.

APU Registers

The APU has registers mapped at \$4000-\$4013, \$4015 (status), and \$4017 (frame counter). Unlike the PPU with its 8 overloaded registers, the APU's layout is more straightforward: each channel gets 4 dedicated registers.

1	\$4000-\$4003	Pulse 1	(control, sweep, timer low, timer high + length)
2	\$4004-\$4007	Pulse 2	(same layout)
3	\$4008-\$400B	Triangle	(linear counter, -, timer low, timer high + length)
4	\$400C-\$400F	Noise	(control, -, mode + period, length)
5	\$4010-\$4013	DMC	(not implemented)
6	\$4015	Status	(enable/disable channels, status readback)
7	\$4017	Frame counter	(4-step vs 5-step mode)

Each register packs multiple values into a single byte using bitmasks, just like the PPU registers. For example, the pulse control register (\$4000) looks like:

1	7	6	5	4	3	2	1	0	
2	D	D	L	C	V	V	V	V	
3					Volume/envelope period (4 bits)				
4				Constant volume flag					
5			Length counter halt / envelope loop						
6	└─┬─┐		Duty cycle (2 bits)						

Six parameters in one byte. 80s economy.

The status register (\$4015) is special: it lets you enable and disable channels individually. Disabling a channel immediately zeroes its length counter (instant silence). Reading it back reports which channels have a length counter > 0 (i.e., are currently producing sound).

\$4017: The Dual Register

\$4017 pulls double duty: when the CPU **reads** this address, it gets controller 2. When it **writes**, it configures the APU's frame counter. Same address, different hardware depending on whether it's a read or write. In the bus, `cpu_read` routes to `controller_2.read` and `cpu_write` routes to `apu.write_frame_counter`.

OceanofPDF.com

The Mixer

Each channel produces a value between 0 and 15. The mixer combines them into a final audio signal.

The real NES has a non-linear DAC (digital-to-analog converter). The circuit's resistances make the mapping from channel values to output voltage non-proportional. At low volumes, the gap between steps is larger than at high volumes.

For our emulator we use a **linear approximation**:

```
1 pulse_out = 0.00752 * (pulse1 + pulse2)
2 tnd_out   = 0.00851 * triangle + 0.00494 * noise
3 output    = pulse_out + tnd_out
```

Those coefficients are chosen so the output range roughly matches the real hardware. The result is a float between 0.0 and ~0.43 that gets sent to the audio system.

Is the linear approximation perfect? No. Put the emulator's output next to a real NES and you'd notice subtle differences in the volume balance between channels. But for playing Mario, it's more than good enough.

OceanofPDF.com

From the NES to the Speaker

The APU generates values at CPU speed: ~1.79 million times per second. Our audio system (SDL2) expects 44,100 samples per second. We need to **downsample**: take 1 sample every ~40.58 CPU cycles ($1,789,773 / 44,100 \sim 40.58$).

The simplest approach: keep an accumulator. Every CPU cycle, add 1.0. When it crosses 40.58, grab a sample from the mixer and subtract 40.58. It's "nearest neighbor" downsampling: we just take whatever value the APU has at that exact moment, no averaging.

Every frame (~16.67ms), the APU accumulates ~735 samples ($44,100 / 60 \sim 735$). At the end of the frame, the GUI drains the samples from the APU's buffer and queues them in SDL2 for playback.

1 CPU writes to \$4000-\$4017	==>	APU configures channels
2 Emulator.step	==>	APU clocks timers and frame counter
3 Every ~40.58 CPU cycles	==>	APU takes a sample (mix of 4 channels)
4 Every frame (~735 samples)	==>	GUI drains samples and queues them in SDL2
5 SDL2	==>	Plays audio through speakers

What We Don't Implement

Two things the real NES has that we're skipping:

The DMC channel: already covered above. Plays 1-bit samples. Doesn't affect the main music or sound effects in most games.

Audio filters: the real hardware has analog filters (a high-pass and a low-pass) that smooth out the signal. Without them, our sound can be a bit rougher at low frequencies and brighter at high ones. It's a subtle difference.

OceanofPDF.com

What We're Going to Build

In the next chapter we write all the code:

1. The **Pulse** channel: duty cycle, timer, envelope, length counter, sweep
2. The **Triangle** channel: 32-step sequence, timer, linear counter, length counter
3. The **Noise** channel: LFSR, period table, envelope, length counter
4. The **APU** class: frame counter, downsampling, mixer, register dispatch
5. **Bus routing**: \$4000-\$4013, \$4015, \$4017
6. **SDL2 audio**: bindings, AudioDevice with queue

By the end of the next chapter, you run Mario and hear the World 1-1 theme.

OceanofPDF.com

Chapter 7.1: Implementing the APU

In the previous chapter we saw how the APU works: square waves, triangle waves, noise, timers, envelopes, length counters, sweep, and the frame counter that keeps them all in sync. Now we implement it.

OceanofPDF.com

The Constants

Before writing any channel code, let's define all the APU constants in a separate module. There are quite a few: register addresses, frame counter timing, mixer coefficients, and the duration table shared by all channels.

```
1 # src/nes/apu/constants.cr
2
3 module NES
4   class APU
5     module Constants
6       # ===== Audio output =====
7       SAMPLE_RATE      = 44100
8       CPU_CLOCK         = 1789773
9       CYCLES_PER_SAMPLE = CPU_CLOCK.to_f / SAMPLE_RATE # ~40.58
```

CYCLES_PER_SAMPLE is how many CPU cycles pass between each audio sample. $1,789,773 / 44,100 \sim 40.58$. We use this for downsampling.

Next up: the register addresses. Each channel has 3 or 4 registers, and we give them readable names with a `_REGISTER` suffix:

```
1   # ===== Pulse 1 registers =====
2   PULSE1_CONTROL_REGISTER = 0x4000_u16 # DDLC VVVV (duty,
length halt, constant vol, volume)
3   PULSE1_SWEEP_REGISTER   = 0x4001_u16 # EPPP NSSS
(enable, period, negate, shift)
4   PULSE1_TIMER_LOW_REGISTER = 0x4002_u16 # TTTT TTTT (timer
low 8 bits)
5   PULSE1_TIMER_HIGH_REGISTER = 0x4003_u16 # LLLL LTTT (length
index, timer high 3 bits)
6
7   # ===== Pulse 2 registers =====
8   PULSE2_CONTROL_REGISTER = 0x4004_u16
9   PULSE2_SWEEP_REGISTER   = 0x4005_u16
10  PULSE2_TIMER_LOW_REGISTER = 0x4006_u16
11  PULSE2_TIMER_HIGH_REGISTER = 0x4007_u16
12
13  # ===== Triangle registers =====
14  TRIANGLE_LINEAR_REGISTER = 0x4008_u16 # CRRR RRRR
```

```

(control flag, linear reload)
15     TRIANGLE_TIMER_LOW_REGISTER = 0x400A_u16
16     TRIANGLE_TIMER_HIGH_REGISTER = 0x400B_u16
17
18     # ===== Noise registers =====
19     NOISE_CONTROL_REGISTER      = 0x400C_u16 # --LC VVVV (length
halt, constant vol, volume)
20     NOISE_MODE_PERIOD_REGISTER = 0x400E_u16 # M--- PPPP (mode,
period index)
21     NOISE_LENGTH_REGISTER      = 0x400F_u16 # LLLL L--- (length
index)

```

Notice that \$4009 and \$400D aren't listed: they're "unused" in the hardware. And \$4010-\$4013 belong to the DMC, which we're not implementing.

The frame counter timing, status register bits, and mixer coefficients:

```

1     # ===== Frame counter timing (CPU cycles) =====
2     FRAME_STEP_1 = 7457 # quarter
3     FRAME_STEP_2 = 14913 # quarter + half
4     FRAME_STEP_3 = 22371 # quarter
5     FRAME_STEP_4 = 29829 # quarter + half, mode 0 resets to 0
6     FRAME_STEP_5 = 37281 # quarter + half, mode 1 resets to 0
7
8     # ===== Status register bit masks =====
9     STATUS_PULSE1 = 0x01_u8
10    STATUS_PULSE2 = 0x02_u8
11    STATUS_TRIANGLE = 0x04_u8
12    STATUS_NOISE = 0x08_u8
13
14    # ===== Mixer coefficients (linear approximation) =====
15    PULSE_MIX = 0.00752_f32
16    TND_T_MIX = 0.00851_f32
17    TND_N_MIX = 0.00494_f32

```

Finally, the duration table shared by all channels. It's 32 values that map a 5-bit index to a frame count:

```

1     # Shared by all channels: maps 5-bit index to duration
2     LENGTH_TABLE = [
3         10_u8, 254_u8, 20_u8, 2_u8, 40_u8, 4_u8, 80_u8, 6_u8,

```

```
4      160_u8, 8_u8, 60_u8, 10_u8, 14_u8, 12_u8, 26_u8, 14_u8,  
5      12_u8, 16_u8, 24_u8, 18_u8, 48_u8, 20_u8, 96_u8, 22_u8,  
6      192_u8, 24_u8, 72_u8, 26_u8, 16_u8, 28_u8, 32_u8, 30_u8,  
7    ]  
8    end  
9  end  
10 end
```

This module gets pulled into the APU class with `include Constants`, and since the channels are defined inside APU, they automatically get access to all these constants too.

OceanofPDF.com

The Pulse Channel

Let's start with the most involved channel. It's got everything: duty cycle, timer, envelope, length counter, and sweep.

```
1  # src/nes/apu/pulse.cr
2
3  module NES
4    class APU
5      class Pulse
6        DUTY_TABLE = [
7          [0_u8, 1_u8, 0_u8, 0_u8, 0_u8, 0_u8, 0_u8, 0_u8], # 12.5%
8          [0_u8, 1_u8, 1_u8, 0_u8, 0_u8, 0_u8, 0_u8, 0_u8], # 25%
9          [0_u8, 1_u8, 1_u8, 1_u8, 1_u8, 0_u8, 0_u8, 0_u8], # 50%
10         [1_u8, 0_u8, 0_u8, 1_u8, 1_u8, 1_u8, 1_u8, 1_u8], # 75%
11       ]
12
13       getter enabled : Bool
14       getter length_counter : UInt8
15
16       def initialize(@channel : UInt8) # 0 = pulse 1, 1 = pulse 2
17         @enabled = false
18         @duty = 0_u8
19         @length_halt = false
20         @constant_volume = false
21         @volume = 0_u8
22         # ... sweep, timer, envelope variables ...
23       end
```

DUTY_TABLE defines the 4 waveforms we saw in the theory chapter. Each one is an 8-step sequence where 1 = high and 0 = low.

@channel tells us whether this is pulse 1 or pulse 2. We need it because of the sweep quirk we covered in the theory chapter.

Channel Enable

When the game disables a channel via \$4015, the length counter is

immediately set to 0:

```
1 def enabled=(value : Bool)
2   @enabled = value
3   @length_counter = 0_u8 unless @enabled
4 end
```

Pulse Registers

Each pulse has 4 registers. Pulse 1's are at \$4000-\$4003, pulse 2's at \$4004-\$4007:

```
1 # $4000/$4004: DDLC VVVV
2 def write_control(value : UInt8)
3   @duty = (value >> 6) & 0x03
4   @length_halt = (value & 0x20) != 0
5   @constant_volume = (value & 0x10) != 0
6   @volume = value & 0x0F
7 end
8
9 # $4002/$4006: TTTT TTTT (timer low)
10 def write_timer_low(value : UInt8)
11   @timer_period = (@timer_period & 0x0700) | value.to_u16
12 end
13
14 # $4003/$4007: LLLL LTTT (length load + timer high)
15 def write_timer_high(value : UInt8)
16   @timer_period = (@timer_period & 0x00FF) | ((value.to_u16 &
17 0x07) << 8)
18   @length_counter = LENGTH_TABLE[(value >> 3) & 0x1F] if @enabled
19   @duty_pos = 0_u8
20   @envelope_start = true
21 end
```

The timer period is 11 bits, split across two registers. `write_timer_high` also loads the length counter from `LENGTH_TABLE`: the top 5 bits of the register are an index, not a direct value. It also resets the duty cycle position and marks the envelope for restart.

Timer and Output

The timer decrements every APU cycle (every 2 CPU cycles). When it hits 0, it advances the duty cycle position:

```
1 def clock_timer
2   if @timer == 0
3     @timer = @timer_period
4     @duty_pos = (@duty_pos &+ 1) & 0x07
5   else
6     @timer &-= 1
7   end
8 end
```

And the channel's output:

```
1 def output : UInt8
2   return 0_u8 unless @enabled
3   return 0_u8 if @length_counter == 0
4   return 0_u8 if DUTY_TABLE[@duty][@duty_pos] == 0
5   return 0_u8 if @timer_period < 8 || sweep_target_period > 0x7FF
6
7   @constant_volume ? @volume : @envelope_decay
8 end
```

There are five conditions that produce silence: channel disabled, note expired (length counter = 0), duty cycle at 0, timer period out of range, or sweep muting. If all those checks pass, the output is the volume (either constant or from the envelope).

Envelope

The envelope starts at 15 and counts down to 0, one step per quarter frame (~240Hz). @volume pulls double duty here: when @constant_volume is

on, it's the literal volume level. When it's off, it's the envelope divider period:

```
1 def clock_envelope
2   if @envelope_start
3     @envelope_start = false
4     @envelope_decay = 15_u8
5     @envelope_divider = @volume
6   else
7     if @envelope_divider == 0
8       @envelope_divider = @volume
9       if @envelope_decay > 0
10        @envelope_decay &-= 1
11      elsif @length_halt
12        @envelope_decay = 15_u8
13      end
14    else
15      @envelope_divider &-= 1
16    end
17  end
18 end
```

@volume of 0 means fast decay, 15 means slow decay. If @length_halt is on, the envelope loops back to 15 when it reaches 0.

Length Counter

Dead simple: if it's not halted and greater than 0, decrement. Gets clocked on the half frame (~120Hz):

```
1 def clock_length_counter
2   if !@length_halt && @length_counter > 0
3     @length_counter &-= 1
4   end
5 end
```

Sweep

```
1 # $4001/$4005: EPPP NSSS
2 def write_sweep(value : UInt8)
```

```
3  @sweep_enabled = (value & 0x80) != 0
4  @sweep_period = (value >> 4) & 0x07
5  @sweep_negate = (value & 0x08) != 0
6  @sweep_shift = value & 0x07
7  @sweep_reload = true
8  end
```

Every half frame, if the sweep is enabled, it calculates a target period and updates the timer period:

```
1  def clock_sweep
2    target = sweep_target_period
3
4    if @sweep_divider == 0 && @sweep_enabled && @sweep_shift > 0
5      if @timer_period >= 8 && target <= 0x7FF
6        @timer_period = target
7      end
8    end
9
10   if @sweep_divider == 0 || @sweep_reload
11     @sweep_divider = @sweep_period
12     @sweep_reload = false
13   else
14     @sweep_divider &-= 1
15   end
16 end
```

And here's the target calculation, complete with the negation quirk that differs between pulse 1 and pulse 2:

```
1  private def sweep_target_period : UInt16
2    change = @timer_period >> @sweep_shift
3    if @sweep_negate
4      if @channel == 0
5        @timer_period &- change &- 1_u16 # Pulse 1: one's
complement
6      else
7        @timer_period &- change          # Pulse 2: two's
complement
8      end
9    else
10     @timer_period &+ change
```

```
11    end
12 end
```

Pulse 1 subtracts the change and subtracts 1 more (one's complement). Pulse 2 just subtracts the change (two's complement). Nobody knows why Nintendo did this, but we have to replicate it.

The full code for `pulse.cr` is in the [repo](#).

OceanofPDF.com

The Triangle Channel

The triangle is simpler. No envelope, no sweep, no duty cycle. Just the 32-step sequence, the timer, the linear counter, and the length counter:

```
1 # src/nes/apu/triangle.cr
2
3 SEQUENCE = [
4   15_u8, 14_u8, 13_u8, 12_u8, 11_u8, 10_u8, 9_u8, 8_u8,
5   7_u8,  6_u8,  5_u8,  4_u8,  3_u8,  2_u8, 1_u8, 0_u8,
6   0_u8,  1_u8,  2_u8,  3_u8,  4_u8,  5_u8, 6_u8, 7_u8,
7   8_u8,  9_u8, 10_u8, 11_u8, 12_u8, 13_u8, 14_u8, 15_u8,
8 ]
```

Goes up from 0 to 15, back down from 15 to 0. That's the triangle wave.

The linear counter is configured via \$4008:

```
1 # $4008: CRRR RRRR
2 def write_linear_counter(value : UInt8)
3   @control_flag = (value & 0x80) != 0
4   @linear_reload = value & 0x7F
5 end
```

The timer clocks every CPU cycle (not every 2 like the pulse), but only advances the sequence if both the length counter and linear counter are greater than 0:

```
1 def clock_timer
2   if @timer == 0
3     @timer = @timer_period
4     if @length_counter > 0 && @linear_counter > 0
5       @sequence_pos = (@sequence_pos &+ 1) & 31
6     end
7   else
8     @timer &-= 1
9   end
10 end
```

The linear counter is clocked every quarter frame. If the reload flag is set, it loads the reload value. Otherwise, it decrements:

```
1 def clock_linear_counter
2   if @linear_reload_flag
3     @linear_counter = @linear_reload
4   elsif @linear_counter > 0
5     @linear_counter &-= 1
6   end
7   @linear_reload_flag = false unless @control_flag
8 end
```

If the timer period is less than 2, we silence the output to avoid ultrasonic clicks:

```
1 def output : UInt8
2   return 0_u8 unless @enabled
3   return 0_u8 if @length_counter == 0
4   return 0_u8 if @linear_counter == 0
5   return 0_u8 if @timer_period < 2 # Silence ultrasonic
6                                     frequencies
7   SEQUENCE[@sequence_pos]
8 end
```

The Noise Channel

A 15-bit LFSR, initialized to 1. The shift operation is the heart of the channel:

```
1 # src/nes/apu/noise.cr
2
3 private def shift
4   bit = @mode ? 6 : 1
5   feedback = (@shift_register & 1) ^ ((@shift_register >> bit) &
6   @shift_register >>= 1
7   @shift_register |= (feedback << 14)
8 end
```

Long mode (bit 1): 32,767 steps, white noise. Short mode (bit 6): 93 steps, metallic.

The frequency comes from a fixed table of 16 periods:

```
1 PERIOD_TABLE = [
2   4_u16, 8_u16, 16_u16, 32_u16, 64_u16, 96_u16, 128_u16, 160_u16,
3   202_u16, 254_u16, 380_u16, 508_u16, 762_u16, 1016_u16, 2034_u16,
4   4068_u16,
5 ]
```

The \$400E register is straightforward: the game picks an index from 0 to 15 and the noise uses the corresponding period:

```
1 # $400E: M--- PPPP
2 def write_mode_period(value : UInt8)
3   @mode = (value & 0x80) != 0
4   @timer_period = PERIOD_TABLE[value & 0x0F]
5 end
```

The output: if bit 0 of the shift register is 1, silence. If it's 0, emit the volume:

```
1 def output : UInt8
2   return 0_u8 unless @enabled
```

```
3   return 0_u8 if @length_counter == 0
4   return 0_u8 if (@shift_register & 1) == 1
5
6   @constant_volume ? @volume : @envelope_decay
7 end
```

The noise channel has the same envelope and length counter as the pulse.
Feel free to just copy it over.

OceanofPDF.com

The APU Class

Time to bring it all together. The APU class coordinates the 4 channels, manages the frame counter, handles downsampling, and mixes the output:

```
1  # src/nes/apu.cr
2
3  require "./apu/constants"
4  require "./apu/pulse"
5  require "./apu/triangle"
6  require "./apu/noise"
7
8  module NES
9    class APU
10      include Constants
11
12      # Channels
13      getter pulse1 : Pulse
14      getter pulse2 : Pulse
15      getter triangle : Triangle
16      getter noise : Noise
17
18      getter frame_mode : UInt8 # 0 = 4-step (~60Hz), 1 = 5-step
19      getter frame_cycle : Int32 # Current cycle within the frame
20      getter sample_clock : Float64 # Accumulator for downsampling
21      getter sample_buffer : Array(Float32) # Float32 samples, GUI
22      getter cycle : UInt32 # CPU cycle counter, even/odd
23
24      def initialize
25        @pulse1 = Pulse.new(0_u8)
26        @pulse2 = Pulse.new(1_u8)
27        @triangle = Triangle.new
28        @noise = Noise.new
29
30        @frame_mode = 0_u8
31        @frame_cycle = 0_i32
32        @sample_clock = 0.0
```

```
33     @sample_buffer = Array(Float32).new(800)
34     @cycle          = 0_u32
35 end
```

`include Constants` brings in everything we defined earlier. The comments next to each getter document what each variable does. The sample buffer is pre-allocated with a capacity of 800, which is enough for one frame (~735 samples) plus some headroom.

The Main Step

The `step` method takes the number of elapsed CPU cycles and clocks everything accordingly:

```
1 def step(cpu_cycles : Int32)
2   cpu_cycles.times do
3     @triangle.clock_timer # Triangle clocks every CPU cycle
4
5     # Pulse and noise clock every 2 CPU cycles (APU cycle)
6     if @cycle.even?
7       @pulse1.clock_timer
8       @pulse2.clock_timer
9       @noise.clock_timer
10    end
11
12    clock_frame_counter
13
14    # Downsample to 44100 Hz
15    @sample_clock += 1.0
16    if @sample_clock >= CYCLES_PER_SAMPLE
17      @sample_clock -= CYCLES_PER_SAMPLE
18      take_sample
19    end
20
21    @cycle &+= 1
22  end
23 end
```

The triangle clocks every CPU cycle. The others only clock on even cycles, meaning every “APU cycle” (half the CPU speed). We check `@cycle.even?` to distinguish.

The downsampling uses an accumulator approach: every CPU cycle we add 1.0 to `sample_clock`. When it crosses `CYCLES_PER_SAMPLE` (~40.58), we grab a sample and subtract. Basically “nearest neighbor” downsampling.

Register Dispatch

This is where those named constants really pay off. Each address routes to the right channel method:

```
1 def write_register(address : UInt16, value : UInt8)
2   case address
3   when PULSE1_CONTROL_REGISTER      then
@pulse1.write_control(value)
4   when PULSE1_SWEEP_REGISTER        then
@pulse1.write_sweep(value)
5   when PULSE1_TIMER_LOW_REGISTER    then
@pulse1.write_timer_low(value)
6   when PULSE1_TIMER_HIGH_REGISTER   then
@pulse1.write_timer_high(value)
7   when PULSE2_CONTROL_REGISTER      then
@pulse2.write_control(value)
8   when PULSE2_SWEEP_REGISTER        then
@pulse2.write_sweep(value)
9   when PULSE2_TIMER_LOW_REGISTER    then
@pulse2.write_timer_low(value)
10  when PULSE2_TIMER_HIGH_REGISTER    then
@pulse2.write_timer_high(value)
11  when TRIANGLE_LINEAR_REGISTER      then
@triangle.write_linear_counter(value)
12  when TRIANGLE_TIMER_LOW_REGISTER  then
@triangle.write_timer_low(value)
13  when TRIANGLE_TIMER_HIGH_REGISTER then
@triangle.write_timer_high(value)
14  when NOISE_CONTROL_REGISTER        then
@noise.write_control(value)
15  when NOISE_MODE_PERIOD_REGISTER    then
@noise.write_mode_period(value)
16  when NOISE_LENGTH_REGISTER         then
@noise.write_length_counter(value)
17  # $4009, $400D: unused
18  # $4010-$4013: DMC (not implemented)
19  end
20 end
```

Compare this to a version with raw hex: when `0x4000_u16` then . . . , when `0x4001_u16` then It would work, but good luck remembering which register is which.

Status Register

\$4015 toggles channels on and off. When a channel gets disabled, its length counter is immediately zeroed out (the `enabled=` setter on each channel handles that):

```
1 # $4015: enable/disable channels
2 def write_status(value : UInt8)
3   @pulse1.enabled = (value & STATUS_PULSE1) != 0
4   @pulse2.enabled = (value & STATUS_PULSE2) != 0
5   @triangle.enabled = (value & STATUS_TRIANGLE) != 0
6   @noise.enabled = (value & STATUS_NOISE) != 0
7 end
8
9 # $4015: read channel status
10 def read_status : UInt8
11   status = 0_u8
12   status |= STATUS_PULSE1 if @pulse1.length_counter > 0
13   status |= STATUS_PULSE2 if @pulse2.length_counter > 0
14   status |= STATUS_TRIANGLE if @triangle.length_counter > 0
15   status |= STATUS_NOISE if @noise.length_counter > 0
16   status
17 end
```

When read, it reports which channels have a length counter > 0 (i.e., which ones are currently making sound).

Frame Counter

The frame counter divides the CPU clock into quarter frames ($_{240\text{Hz}}$) and half frames ($_{120\text{Hz}}$):

```
1 # $4017: frame counter mode
2 def write_frame_counter(value : UInt8)
3   @frame_mode = (value >> 7) & 1
```



```
4  @frame_cycle = 0
5
6  # Mode 1: immediately clocks all units
7  if @frame_mode == 1
8      clock_quarter_frame
9      clock_half_frame
10 end
11 end
```

Writing to \$4017 with mode 1 immediately clocks all units. Mode 0 doesn't do that.

The frame counter clock checks where we are in the sequence using the FRAME_STEP_* constants:

```
1  def clock_frame_counter
2      @frame_cycle += 1
3
4      quarter = false
5      half = false
6
7      if @frame_mode == 0 # 4-step
8          case @frame_cycle
9              when FRAME_STEP_1
10                 quarter = true
11             when FRAME_STEP_2
12                 quarter = true
13                 half = true
14             when FRAME_STEP_3
15                 quarter = true
16             when FRAME_STEP_4
17                 quarter = true
18                 half = true
19                 @frame_cycle = 0
20             end
21         else # 5-step
22             case @frame_cycle
23                 when FRAME_STEP_1
24                     quarter = true
25                 when FRAME_STEP_2
26                     quarter = true
27                     half = true
28                 when FRAME_STEP_3
29                     quarter = true
```

```
30     when FRAME_STEP_5
31         quarter = true
32         half = true
33         @frame_cycle = 0
34     end
35 end
36
37 clock_quarter_frame if quarter
38 clock_half_frame if half
39 end
```

Mode 0 has 4 steps and resets at FRAME_STEP_4 (29829 cycles). Mode 1 has 5 steps and resets at FRAME_STEP_5 (37281 cycles). Steps 2 and 4/5 clock both quarter and half frame. Steps 1 and 3 clock quarter only.

Then the methods that actually clock each channel's units:

```
1 def clock_quarter_frame
2     @pulse1.clock_envelope
3     @pulse2.clock_envelope
4     @triangle.clock_linear_counter
5     @noise.clock_envelope
6 end
7
8 def clock_half_frame
9     @pulse1.clock_length_counter
10    @pulse1.clock_sweep
11    @pulse2.clock_length_counter
12    @pulse2.clock_sweep
13    @triangle.clock_length_counter
14    @noise.clock_length_counter
15 end
```

Quarter frame: envelopes and the triangle's linear counter. Half frame: length counters and sweeps.

The Mixer and Downsampling

```
1 def take_sample
2     p1 = @pulse1.output.to_f32
3     p2 = @pulse2.output.to_f32
```

```
4   t = @triangle.output.to_f32
5   n = @noise.output.to_f32
6
7   pulse_out = PULSE_MIX * (p1 + p2)
8   tnd_out   = TND_T_MIX * t + TND_N_MIX * n
9
10  @sample_buffer << (pulse_out + tnd_out)
11 end
```

The PULSE_MIX, TND_T_MIX, and TND_N_MIX coefficients are the linear approximation of the NES mixer from the theory chapter. Each channel outputs 0-15, and the mixer combines them into a float between 0.0 and ~0.43.

Then there's `drain_samples`, which the GUI calls every frame:

```
1 def drain_samples : Array(Float32)
2   samples = @sample_buffer
3   @sample_buffer = Array(Float32).new(800)
4   samples
5 end
```

Returns the current buffer and creates a new one. No copying, no clearing: we just swap the reference.

Connecting to the Bus

In `addresses.cr`, we add the APU address constants:

```
1 # src/nes/bus/addresses.cr
2
3 # ===== APU registers =====
4 APU_START = 0x4000_u16
5 APU_END   = 0x4013_u16
6 APU_STATUS = 0x4015_u16
7 APU_FRAME_COUNTER = 0x4017_u16
```

In `bus.cr`, the constructor now receives the APU, and reads/writes get routed:

```
1 def cpu_read(address : UInt16) : UInt8
2   case address
3     # ...
4     when APU_STATUS
5       @apu.read_status
6     when CONTROLLER_1
7       @controller_1.read
8     when CONTROLLER_2
9       @controller_2.read
10    # ...
11  end
12 end
13
14 def cpu_write(address : UInt16, value : UInt8)
15   case address
16     # ...
17     when APU_START..APU_END
18       @apu.write_register(address, value)
19     when OAM_DMA
20       oam_dma(value)
21     when APU_STATUS
22       @apu.write_status(value)
23     when CONTROLLER_1
24       @controller_1.write(value)
25       @controller_2.write(value)
26     when APU_FRAME_COUNTER
```

```
27     @apu.write_frame_counter(value)
28   end
29 end
```

Remember that \$4017 does double duty: reads go to controller 2, writes go to the APU's frame counter. CONTROLLER_2 and APU_FRAME_COUNTER are the same address (0x4017), but the bus routes them differently depending on whether it's a read or a write.

OceanofPDF.com

The Emulator

In `emulator.cr`, we create the APU and wire it into the loop:

```
1 # src/nes/emulator.cr
2
3 getter apu : APU
4
5 def initialize(path : String)
6   @cartridge = Cartridge.new(path)
7   @ppu = PPU.new(@cartridge)
8   @apu = APU.new
9   @bus = Bus.new(@cartridge, @ppu, @apu)
10  @cpu = CPU.new(@bus)
11
12  @ppu.nmi_callback = ->{ @cpu.nmi }
13  @apu.reset
14 end
15
16 def step
17   cycles = @cpu.step
18   (cycles * 3).times { @ppu.step }
19   @apu.step(cycles)
20   cycles
21 end
```

The PPU runs at 3x the CPU speed. The APU runs at 1x. Each CPU instruction generates N cycles, the PPU advances $N*3$ steps, and the APU advances N steps.

Audio with SDL2

We need to turn the APU's samples into actual sound. SDL2 has a queue-based audio system: we push samples into a queue, and SDL2 plays them back.

First, the audio bindings in `sdl2.cr`:

```
1  # bin/bindings/sdl2.cr
2
3  INIT_AUDIO = 0x00000010_u32
4  AUDIO_F32SYS = 0x8120_u16
5
6  struct AudioSpec
7    freq : Int32
8    format : UInt16
9    channels : UInt8
10   silence : UInt8
11   samples : UInt16
12   padding : UInt16
13   size : UInt32
14   callback : Void*
15   userdata : Void*
16 end
17
18 fun open_audio_device = SDL_OpenAudioDevice(...) : UInt32
19 fun pause_audio_device = SDL_PauseAudioDevice(dev : UInt32,
pause_on : Int32)
20 fun queue_audio = SDL_QueueAudio(dev : UInt32, data : Void*, len
: UInt32) : Int32
21 fun get_queued_audio_size = SDL_GetQueuedAudioSize(dev : UInt32)
: UInt32
```

We use `AUDIO_F32SYS` (32-bit float in the system's byte order), one channel (mono), 44100Hz. Instead of callbacks, we use `SDL_QueueAudio`, which is way simpler: just hand it an array of samples and SDL plays them in order.

The AudioDevice wrapper:

```
1 class AudioDevice
2   def initialize(sample_rate : Int32 = 44100, buffer_size :
  UInt16 = 1024_u16)
3     desired = LibSDL::AudioSpec.new
4     desired.freq = sample_rate
5     desired.format = LibSDL::AUDIO_F32SYS
6     desired.channels = 1_u8
7     desired.samples = buffer_size
8     desired.callback = Pointer(Void).null
9
10    obtained = LibSDL::AudioSpec.new
11    @dev = LibSDL.open_audio_device(
12      Pointer(UInt8).null, 0,
13      pointerof(desired), pointerof(obtained), 0
14    )
15    raise "SDL audio device failed" if @dev == 0
16    LibSDL.pause_audio_device(@dev, 0)
17  end
18
19  def queue(samples : Array(Float32))
20    return if samples.empty?
21    return if LibSDL.get_queued_audio_size(@dev) > 8192 * 4
22
23    LibSDL.queue_audio(
24      @dev, samples.to_unsafe.as(Void*),
25      (samples.size * 4).to_u32
26    )
27  end
28 end
```

The `get_queued_audio_size` check keeps the queue from growing unbounded. If there's more than ~185ms of queued audio (8192 samples * 4 bytes each), we stop adding until some of it gets consumed. This prevents audio from drifting out of sync with video when the emulator runs faster than real time.

We initialize audio in `NESApp` and update the SDL init call to include audio:

```
1 # bin/gui.cr
2
```



```
3 SDL.init # Now initializes video + audio
4 @audio = SDL::AudioDevice.new
```

And in the game loop, after each frame:

```
1 frame_buffer = @emulator.run_frame
2 @audio.queue(@emulator.apu.drain_samples)
```

`drain_samples` returns the ~735 samples accumulated during the frame (44100 / 60 ~ 735) and clears the internal buffer.

OceanofPDF.com

The Moment of Truth

```
1 crystal build bin/gui.cr -o nes_gui --release
2 ./nes_gui .roms/mario.nes
```

Hit Enter on the title screen.

And there it is: the World 1-1 theme. The two melodic voices (pulse 1 and 2), the bass line (triangle), the percussion (noise). Jump and you hear a “boing”. Grab a coin and there’s a “bling”. Stomp a Goomba and you get a “thud”. Die and the funeral march plays.

The emulator has sound.

It won’t sound perfect. Without the DMC channel, some sample-based effects are missing. Without an audio filter, the sound can be a bit rough around the edges. But the music is there, the melodies are recognizable, and the game feels alive.

OceanofPDF.com

Wrap up

We added the APU to the emulator with four audio channels:

- **apu/constants.cr**: all APU constants in a Constants module. Registers, timing, mixer, duration table.
- **apu/pulse.cr**: square wave with duty cycle, timer, envelope, length counter, and sweep. Two instances (pulse 1 and pulse 2) with a minor difference in sweep negation.
- **apu/triangle.cr**: triangle wave with linear counter and length counter. Fixed volume, clocks at double speed.
- **apu/noise.cr**: 15-bit LFSR with long mode (white noise) and short mode (metallic). Envelope and length counter.
- **apu.cr**: includes Constants, coordinates the 4 channels. Frame counter (quarter and half frame), downsampling to 44100Hz, linear mixer, register dispatch.
- **bus.cr**: routing for \$4000-\$4013, \$4015, \$4017.
- **sd12.cr**: SDL2 audio bindings, AudioDevice wrapper with queue.

The full audio flow:

1 CPU writes to \$4000-\$4017	=>	APU updates channel registers
2 Emulator.step	=>	APU clocks timers and frame counter
3 Every ~40.58 CPU cycles	=>	APU takes a sample (mix of 4 channels)
4 Every frame (~735 samples)	=>	GUI drains samples and queues them in SDL2
5 SDL2	=>	Plays audio through speakers

The full code for this chapter is in the [repo](#).

The emulator is complete. CPU, PPU, APU, controls, cartridges. We started with a loop that read opcodes from a byte array, and ended up with an emulator that runs Super Mario Bros with graphics, scroll, sprites, controls, and sound. All from scratch.

In the appendix you'll find the Mapper 1 (MMC1) implementation, which opens the door to Zelda, Metroid, and Mega Man 2. But that's just bonus

material. The important stuff is already done.

OceanofPDF.com

Chapter 8: Closing

We started with a loop that read bytes from an array. We ended with an emulator that runs Super Mario Bros with graphics, scroll, sprites, controls, and sound. All from scratch.

If you made it this far, you're not the same person who cracked open chapter 1. You now know what happens inside a NES. Not in the abstract, not "sort of." You know how the CPU decodes an opcode, how the bus routes an address, how the PPU builds a frame pixel by pixel, how the APU mixes square waves into music.

And more importantly: you built it yourself.

OceanofPDF.com

What we actually learned

If someone asked you “what does it mean to emulate?”, before this book you’d probably have said something like “making a program run on a different platform.” Now you can give a better answer:

Emulating means understanding hardware and modeling it in software.

Two steps. The first is the most important and the most time-consuming: understanding what the hardware actually does. Not at the level of “the CPU executes instructions.” At the level of “the CPU has an 8-bit register called A, a stack pointer that starts at \$FD, and when it executes ADC it adds the value, the carry, and the accumulator, then sets four flags based on the result.” That level of detail.

The second step is translating that understanding into code. And here’s what might surprise you: the programming part is nothing special. It’s just object-oriented programming (or whatever paradigm you prefer) applied to a specific domain. A CPU is a class with registers, a step method that decodes opcodes, and methods for each instruction. A bus is a case/when that routes addresses. A PPU is internal state that produces a frame buffer.

There’s no magic. No exotic algorithms or complex math. You read the documentation, understand the behavior, and write code that replicates it. It’s craftsmanship, not rocket science.

OceanofPDF.com

The third component everyone forgets

There's something people tend to overlook when they talk about emulation: **presentation**.

Emulating the hardware is only half the job. The other half is displaying the result on a modern system. It doesn't matter if your PPU generates perfect frames if you can't draw them in a window. It doesn't matter if your APU mixes audio flawlessly if you can't send it to the speakers.

That's why we spent time on SDL2: bindings, window, renderer, textures, audio queue. That glue layer between your emulator and the operating system is just as essential as the emulation itself. If you wanted to port the emulator to the web tomorrow, you'd swap SDL2 for WebGL and Web Audio, but the NES logic wouldn't change at all. That separation is key.

OceanofPDF.com

So what do I emulate next?

If you've caught the bug, there's a huge world out there. Here are a few options, roughly sorted by difficulty:

CHIP-8: the "Hello World" of emulation

If you have a friend who wants to get into emulation but finds the NES intimidating, point them to CHIP-8. It's a virtual machine from the 70s with 35 instructions, 4KB of RAM, and a 64x32 black-and-white screen. Totally doable in a weekend. It's not a real console, but it teaches all the fundamental concepts: fetch-decode-execute, registers, memory, input, display.

Game Boy: the natural next step

The Game Boy's CPU (a Sharp LR35902) has way more instructions than the 6502 (around 500 opcodes if you count the CB-prefixed ones), but the PPU is considerably simpler than the NES's. No attribute tables, no convoluted sprite priority system, and scrolling is more straightforward. It's a nice tradeoff: more tedious CPU, friendlier PPU.

Plus, it has Tetris, Pokemon, and Link's Awakening. That's all the motivation you need.

Atari 2600: looks easy, it's not

The Atari 2600 has a 6502 processor (almost the same as the NES) and only 128 bytes of RAM. "128 bytes? Must be trivial," you'd think. Nope.

The Atari has no frame buffer. No VRAM. Nothing resembling a modern PPU. The programmer has to feed the graphics chip (the TIA) scanline by scanline, in real time, counting exact cycles. Fall behind by one cycle and the image breaks. Atari games are basically programs racing against the electron beam every single frame.

Emulating that requires cycle-exact precision. You can't get away with the "execute the full instruction then update the video" approach. Every single cycle counts. It's a fascinating but deeply frustrating exercise. If you enjoy suffering, this is your console.

SNES, Genesis, GBA...

From here on out, things get seriously complex. 16-bit CPUs, multiple video chips, elaborate graphics modes, DMA, coprocessors. The SNES has an audio chip that's basically a separate computer (the SPC700). The Genesis has two CPUs (a 68000 and a Z80) running in parallel.

These are month-long projects, not weekend ones. But the foundation is the same: understand the hardware and model it in code. The only thing that changes is how much hardware there is to understand.

OceanofPDF.com

Improvements for this emulator

If you want to keep improving what we built, there's plenty to explore:

- **More mappers:** Mapper 2 (UxROM) is almost trivial and unlocks Castlevania and Contra. Mapper 4 (MMC3) is more complex but covers Mario 3 and Kirby.
- **DMC channel:** the fifth APU channel that we skipped. It plays 1-bit samples. Some games use it for voices or special effects.
- **Audio filters:** the real hardware applies high-pass and low-pass filters to the audio signal. Adding them smooths out the sound noticeably.
- **Pixel-by-pixel rendering:** our PPU renders an entire scanline at once. Rendering pixel by pixel would let you emulate mid-scanline effects that some games rely on.
- **Save states:** snapshot and restore the emulator's complete state. Great for debugging and, let's be honest, cheating.

OceanofPDF.com

Final words

If this book was good for anything, I hope it's this: showing you that things that seem complicated are usually just a bunch of simple things wired together. The NES looks like magic until you realize it's a bus connecting a CPU, a PPU, and a cartridge. The CPU seems complex until you see it's a giant switch statement reading opcodes. The PPU seems impossible until you see it's a loop iterating over tiles.

There's no magic. Just patience and a desire to understand how things work.

Thanks for sticking with it. If you have questions, found a bug, or just want to show me your emulator running Zelda, drop me a line at hello@matiassalles99.codes. You can also check out my blog at <https://matiassalles99.codes>, where I write about this and other projects.

And if you implement a new mapper, send it my way. I'd love to check it out.

Now go play.

OceanofPDF.com

Appendix: Mappers

OceanofPDF.com

Mapper 1 (MMC1)

Used by Zelda, Metroid, Mega Man 2, Final Fantasy, Blaster Master, and many more. It's the most popular mapper after NROM and the first one that does real bank switching on the NES.

The problem it solves is simple: the CPU can only see 32KB of PRG-ROM at a time (\$8000-\$FFFF), but these games have up to 256KB of code. The MMC1 lets you change which portion of the ROM is visible in that range.

The way you talk to this mapper is unusual. Instead of writing a value directly, the CPU sends **one bit at a time** through a 5-bit shift register. After 5 consecutive writes to \$8000-\$FFFF, the mapper interprets the accumulated value as a command. Weird design, but there's logic behind it: Nintendo wanted to save pins on the chip, so they opted to receive data serially instead of in parallel.

With those 5 bits, the game can configure four things:

- **Which PRG-ROM bank** is mapped (up to 16 banks of 16KB)
- **Which CHR bank** the PPU uses for tiles (in 4KB or 8KB blocks)
- **The nametable mirroring mode** (horizontal, vertical, or single-screen)
- **The PRG bank switching mode** (fix the first bank, fix the last bank, or switch full 32KB chunks)

On top of that, the MMC1 includes **8KB of WRAM** at \$6000-\$7FFF. In battery-backed cartridges like Zelda, that RAM holds save data. But it's not just for saves: Zelda actually loads executable routines into WRAM and jumps there with JMP (yeah, kind of dirty). Without WRAM implemented, the game crashes the moment you press Start.

Dynamic mirroring is another big difference from Mapper 0. NROM uses whatever mirroring the header specifies and never changes it. The MMC1 can switch between horizontal, vertical, and single-screen on the fly. To support this, the PPU needs to check with the mapper first before falling back to the header's mirroring value.

Code in the [repo](#).

OceanofPDF.com