Building Your First Emulator

Understanding How Emulators Work
by Building a NES from Scratch

Contents

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.

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.

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.

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.

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.

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[1], 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.

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.

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.

[1] https://www.nesdev.org/wiki/Nesdev_Wiki

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.

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.

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.

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:

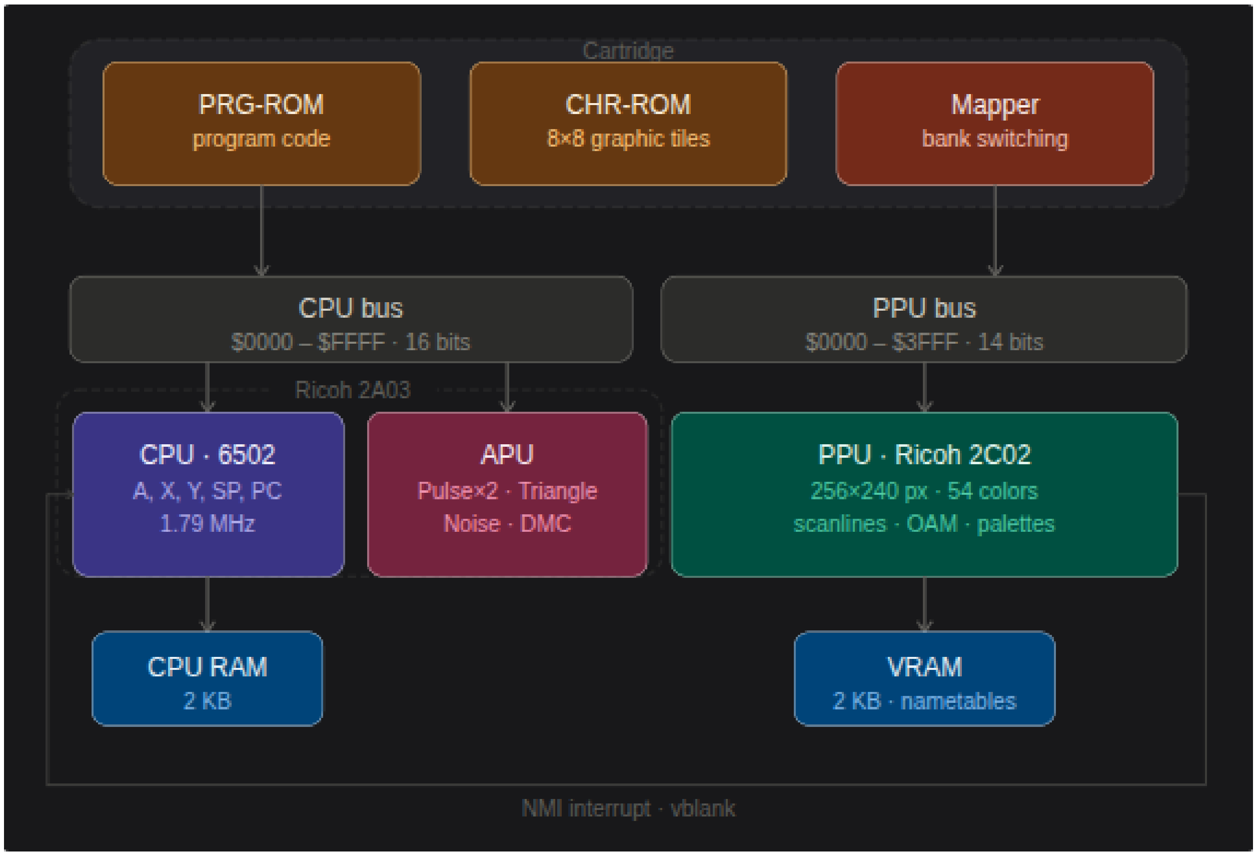

Figure 1. NES Diagram

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:

```ruby
loop do
  # 1. CPU executes one instruction
  cpu_cycles = cpu.step

  # 2. PPU runs 3 cycles for each CPU cycle
  (cpu_cycles * 3).times do
    ppu.step
  end

  # 3. Repeat forever (or until you close the emulator)
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.

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.

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.

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.

Installing Crystal

There are several ways to install Crystal. You can find the full list on the official Crystal[1] 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.

Compile and run

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

```crystal
# main.cr

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] https://crystal-lang.org/install/

```
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)

```
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:

```
# Linux or Mac
./hello

# Windows
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.

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.

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.

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:

```
7 6 5 4 3 2 1 0
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.

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:

```
PC = 0x0000

1. Read the byte at 0x0000 -> it's 0xA9 (LDA immediate instruction)
2. Advance PC by 1 -> 0x0001
3. LDA needs an operand, read the byte at 0x0001 (PC) -> it's 0x42
4. Advance PC by 1 -> 0x0002
5. Execute: load 0x42 into register A
6. Go back to step 1, now with PC at 0x0002

(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:

```
nes-emulator/
├── bin/
│   └── test_cpu.cr       # Entry point for testing
└── src/
    ├── nes.cr            # Main module
    └── nes/
        ├── emulator.cr   # Coordinates components
        ├── bus.cr        # Communication between components
        └── 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:

```
# src/nes.cr

require "./nes/emulator"

module NES
  VERSION = "1.0"
end
```

```
# src/nes/emulator.cr

require "./bus"
require "./cpu"

module NES
  class Emulator
    getter cpu : CPU
    getter bus : Bus

    def initialize # this is how you call the constructor in Ruby and Crystal
        @bus = Bus.new
        @cpu = CPU.new(@bus)
    end

    def whats_up_bro
        puts "Whats up bro"
    end
  end
end
```

```
# src/nes/cpu.cr

module NES
  class CPU
    def initialize(bus : Bus)
      @bus = bus
    end
  end
end
```

```
# src/nes/cpu.cr

module NES
  class Bus
  end
end
```

```
# bin/test_cpu.cr

require "../src/nes"

emulator = NES::Emulator.new
emulator.whats_up_bro
```

The code for this part is here[1]

Now run this to make sure everything is working:

```
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.

[1] https://github.com/matiassalles99/nes-emulator-book/commit/fa49d728012da3965a546df5a7faf22ce37ae694

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.

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.

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:

```
value = 0xA9_u8          # 8 bits, the LDA opcode
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.

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:

```
# src/nes/cpu/op_codes.cr

module NES
  class CPU
    module OpCodes
      # LDA
      CODE_LDA_IMMEDIATE = 0xA9_u8

      # Register Increments
      CODE_INX = 0xE8_u8

      # Cycles per Operation Code
      CYCLES = {
        CODE_LDA_IMMEDIATE => 2,
        CODE_INX => 2
      }
    end
  end
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:

```
# src/nes/cpu.cr

require "./cpu/op_codes"
require "./cpu/instructions"
require "./cpu/flags"

module NES
  class CPU
    include Flags
    include Instructions
    include OpCodes

    getter a      : UInt8   # Accumulator
    getter x      : UInt8   # X register
    getter y      : UInt8   # Y register
    getter sp     : UInt8   # Stack Pointer
    getter pc     : UInt16  # Program Counter
    getter status : UInt8   # Status register (flags)
```

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

Now the `initialize`:

```crystal
# src/nes/cpu.cr

  def initialize(bus : Bus)
    @bus    = bus
    @a      = 0_u8 # Remember that _u8 means it's an 8-bit UInt
    @x      = 0_u8
    @y      = 0_u8
    @sp     = 0xFD_u8
    @status = 0x24_u8
    @pc     = 0_u16 # Same here, _u16 means a 16-bit UInt
  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:

```crystal
# src/nes/cpu.cr

  def step
    opcode = fetch_byte

    case opcode
    when OpCodes::CODE_LDA_IMMEDIATE then op_lda_immediate
    when OpCodes::CODE_INX           then op_inx
    end

    CYCLES[opcode]
  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:

```crystal
# src/nes/cpu.cr

    def fetch_byte : UInt8
      value = read_byte(@pc)
      @pc &+= 1

      value
    end

    def read_byte(address : UInt16) : UInt8
      @bus.cpu_read(address)
    end
  end
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.

The instructions

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

```crystal
# src/nes/cpu/instructions/lda.cr

module NES
  class CPU
    module Instructions
      module LDA
        def lda(value)
          @a = value

          set_z_flag(@a)
          set_n_flag(@a)
        end

        def op_lda_immediate
          value = fetch_byte

          lda(value)
        end
      end
    end
  end
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[1] has it all. Every instruction is documented with exactly what it does and which flags it touches.

Now `INX`:

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

module NES
  class CPU
    module Instructions
      module RegisterIncrements
        def op_inx
          @x &+= 1

          set_z_flag(@x)
          set_n_flag(@x)
        end
      end
    end
  end
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] https://www.nesdev.org/wiki/Instruction_reference#LDA

```
# src/nes/cpu/instructions.cr

require "./instructions/lda"
require "./instructions/register_increments"

module NES
  class CPU
    module Instructions
      include LDA
      include RegisterIncrements
    end
  end
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.

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:

```
# src/nes/cpu/flags.cr

module NES
  class CPU
    module Flags
      # Flag Masks (N V U B D I Z C)
      FLAG_C = 0b0000_0001_u8  # Carry
      FLAG_Z = 0b0000_0010_u8  # Zero
      FLAG_I = 0b0000_0100_u8  # Interrupt Disable
      FLAG_D = 0b0000_1000_u8  # Decimal
      FLAG_B = 0b0001_0000_u8  # Break
      FLAG_U = 0b0010_0000_u8  # Unused (always 1)
      FLAG_V = 0b0100_0000_u8  # Overflow
      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:

```
# src/nes/cpu/flags.cr

      def set_z_flag(register_value : UInt8)
        is_zero = register_value == 0
        set_flag(FLAG_Z, is_zero)
      end

      def set_n_flag(register_value : UInt8)
        seventh_bit = register_value.bit(7)
        is_on = seventh_bit == 1
        set_flag(FLAG_N, is_on)
      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:

```
# src/nes/cpu/flags.cr

      def set_flag(mask : UInt8, on : Bool)
        if on
          @status |= mask
        else
          @status &= (~mask).to_u8
        end
      end
    end
  end
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`:

```
status = 0000 0000
FLAG_Z = 0000 0010

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

And now we want to turn it back off:

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

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.

The Bus

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

```
# src/nes/bus/addresses.cr

module NES
  class Bus
    module Addresses
      RAM_SIZE = 2048  # 2 KB

      RAM_START      = 0x0000_u16
      RAM_END        = 0x07FF_u16
      RAM_MIRROR_END = 0x1FFF_u16

      RAM_MASK = 0x07FF_u16
    end
  end
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.

`RAM_MASK = 0x07FF` is how we calculate the real address. `0x0800 & 0x07FF = 0x0000`. Bit magic.

Now the Bus itself:

```
# src/nes/bus.cr

require "./bus/addresses"

module NES
  class Bus
    include Addresses

    def initialize
      @ram = Bytes.new(RAM_SIZE, 0_u8)
    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:

```
# src/nes/bus.cr

    def cpu_read(address : UInt16) : UInt8
      case address
      when RAM_START..RAM_MIRROR_END
        read_ram(address)
      else
        0_u8
      end
    end

    def cpu_write(address : UInt16, value : UInt8)
      case address
      when RAM_START..RAM_MIRROR_END
        write_ram(address, value)
      end
    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:

```
# src/nes/bus.cr

    def read_ram(address : UInt16) : UInt8
      mirrored_address = (address & RAM_MASK)
      @ram[mirrored_address]
    end

    def write_ram(address : UInt16, value : UInt8)
      mirrored_address = (address & RAM_MASK)
      @ram[mirrored_address] = value
    end
  end
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.

The Emulator

The Emulator coordinates everything:

```
# src/nes/emulator.cr

require "./bus"
require "./cpu"
require "./emulator/debugging"

module NES
  class Emulator
    include Debugging

    getter cpu : CPU
    getter bus : Bus

    def initialize
      @bus = Bus.new
      @cpu = CPU.new(@bus)
    end
```

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

The `step` method executes one step of the emulation:

```
# src/nes/emulator.cr

    def step
      cycles = @cpu.step
      # TODO: PPU steps cycles * 3
    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:

```
# src/nes/emulator.cr

    def load_program(program : Bytes, start_address : UInt16 = Bus::RAM_START)
      program.each_with_index do |byte, i|
        @bus.cpu_write((start_address &+ i), byte)
      end
    end
  end
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.

Debugging

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

```
# src/nes/emulator/debugging.cr

module NES
  class Emulator
    module Debugging
      def debug
        puts "    A=#{hex8(@cpu.a)} X=#{hex8(@cpu.x)} Y=#{hex8(@cpu.y)} SP=#{hex8(@c\
pu.sp)} PC=#{hex16(@cpu.pc)}  [#{status_string}]"
      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:

```
# src/nes/emulator/debugging.cr

      def hex8(value : UInt8) : String
        "$#{value.to_s(16).rjust(2, '0').upcase}"
      end

      def hex16(value : UInt16) : String
        "$#{value.to_s(16).rjust(4, '0').upcase}"
      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:

```
# src/nes/emulator/debugging.cr

      def status_string : String
        s = @cpu.status
        String.build do |str|
          str << (s.bit(7) == 1 ? 'N' : 'n')
          str << (s.bit(6) == 1 ? 'V' : 'v')
          str << (s.bit(5) == 1 ? 'U' : 'u')
          str << (s.bit(4) == 1 ? 'B' : 'b')
          str << (s.bit(3) == 1 ? 'D' : 'd')
          str << (s.bit(2) == 1 ? 'I' : 'i')
          str << (s.bit(1) == 1 ? 'Z' : 'z')
          str << (s.bit(0) == 1 ? 'C' : 'c')
        end
      end
    end
  end
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:

```
# bin/test_cpu.cr

require "./cpu/test_lda"

puts "\n"
puts "#" * 60
puts "\n"

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:

```
# bin/cpu/test_lda.cr

require "../../src/nes"

puts "LDA Immediate"

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:

```
# Test 1: Load value
puts "\n  1. Load value"
program = Bytes[
  0xA9,  # LDA immediate
  0x42   # value
]

emu = NES::Emulator.new
emu.load_program(program)

puts "     LDA #$42"
emu.step
emu.debug

passed = emu.cpu.a == 0x42 && emu.cpu.status.bit(1) == 0 && emu.cpu.status.bit(7) ==\
 0
puts passed ? "     ✅" : "     ❌"
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:

```
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[2].

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:

```
nes-emulator/
├── bin/
│   ├── test_cpu.cr
│   └── cpu/
│       ├── test_lda.cr
│       └── test_inx.cr
└── src/
    ├── nes.cr
    └── nes/
        ├── bus.cr
        ├── cpu.cr
        ├── emulator.cr
        ├── bus/
        │   └── addresses.cr
        ├── cpu/
        │   ├── flags.cr
        │   ├── instructions.cr
```

[2]https://github.com/matiassalles99/nes-emulator-book/tree/c8fd13a6236d0aec3126873562cb93f2dd95dddf/bin/cpu

```
│    ├── op_codes.cr
│    └── instructions/
│        ├── lda.cr
│        └── register_increments.cr
└── emulator/
    └── debugging.cr
```

If something isn't working, you can compare with the code in the repository: Github[3]

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.

[3]https://github.com/matiassalles99/nes-emulator-book/commit/c8fd13a6236d0aec3126873562cb93f2dd95dddf