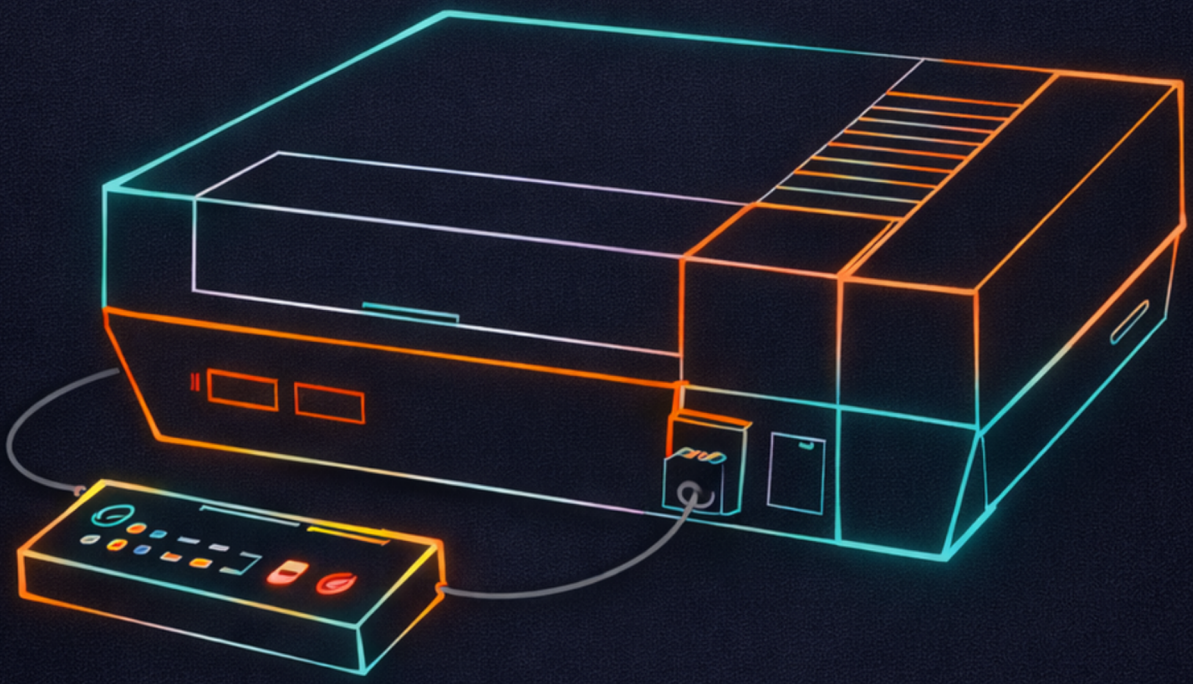
Construí
tu prímer emulador

Cómo funciona un emulador por dentro,
construyendo una NES desde cero



Índice general

Introducción

Un día estaba buscando "Mario Bros NES online" a las 2 de la mañana. Nostalgia pura. Encontré un emulador, jugué 15 minutos, morí en el mundo 2-3 como siempre, y cerré la pestaña. Pero algo me quedó dando vueltas: ¿cómo funciona eso?

Siempre me había fascinado la idea de las máquinas virtuales. Cómo el software puede simular hardware. Cómo podés tener infinitas computadoras dentro de tu computadora, configurándoles los specs con un par de clicks. Pero nunca había cruzado de "**qué interesante**" a "**voy a entender cómo se hace**".

Esa noche crucé. Tres semanas después tenía un emulador funcionando. Y acá estamos.

Para quién es este libro

Para gente que sabe programar (nivel intermedio mínimo) y le gusta aprender haciendo. No necesitás saber nada de emulación, ni de hardware retro, ni de assembler. Solo ganas de construir algo.

Si alguna vez pensaste "me gustaría hacer un proyecto grande pero no sé por dónde empezar", este es un buen candidato. Es acotado (no estamos emulando una PS2), es tangible (al final tenés algo que funciona y podés usar), y tiene esa satisfacción única de ver código tuyo corriendo un juego de tu infancia.

Qué vas a lograr

Cuando termines, vas a tener un emulador de NES funcionando. Uno que entendés de punta a punta: qué hace cada módulo, por qué está ahí, y cómo se conecta con el resto. Va a correr juegos con los mappers 0 y 1 (ya vas a aprender que son los mappers), que incluye la mayoría de los clásicos: Super Mario Bros, Zelda, Metroid, Contra, Mega Man 2.

Mi test de calidad durante todo el desarrollo fue simple: si corre Mario, vamos bien.

Qué NO es este libro

Seamos claros desde el principio:

- **No es un emulador perfecto**: No emulamos instrucciones no documentadas de la CPU, ni replicamos todos los bugs del hardware original. Hay proyectos que hacen eso con precisión de cirujano. Este prioriza que entiendas qué está pasando.

- **No es una referencia técnica exhaustiva**: Si querés la documentación completa de la NES, hay wikis enteras dedicadas a eso. Acá vas a encontrar lo necesario para construir algo que funcione, explicado de forma que no te duermas.
- **No es un libro para principiantes absolutos**: Asumo que sabés qué es una función, un loop, y que no te asustás si ves código.

Lo que sí es: una guía práctica para construir tu primer emulador, escrita por alguien que tampoco sabía nada cuando empezó.

Por qué Crystal (y no C, y casi Ruby)

Muy poca gente programa en C o C++ en el día a día. Y la idea de este libro es que entiendas cómo funciona un emulador, no que pases 15 minutos descifrando punteros.

Por eso arranqué escribiendo el emulador en Ruby. Legible, expresivo, un placer de leer. Spoiler: También increíblemente lento.

Cuando tuve una versión que corría Mario minimamente, me andaba a 0.5 FPS. Medio frame por segundo. Con optimizaciones llegué a 2 FPS. Cargar el nivel 1-1 tardaba como 4 minutos.

Fue un aprendizaje interesante: siempre había escuchado "Ruby es lento" como una frase abstracta, algo que no importaba para la mayoría de las aplicaciones. Y es verdad, para la mayoría no importa. Pero emular hardware en tiempo real no es la mayoría. Fue la primera vez que me encontré con una limitación real, no teórica, donde un lenguaje simplemente no puede hacer el trabajo.

Así que reescribí todo en Crystal: un lenguaje creado en Argentina que tiene la misma sintaxis que Ruby pero es tipado y compilado. Sé que el tipado puede triggerear a algunos, pero los resultados hablan solos: sin ninguna optimización, el emulador pasó a correr a 120 FPS. El mismo código, 240 veces más rápido.

Todo el código de este libro está en Crystal. Si venís de Ruby, vas a leerlo sin problemas. Si no conocías Crystal, vas a aprender un lenguaje nuevo de paso.

Cómo está organizado el libro

Arrancamos con una vista general de la NES y qué significa emular una consola. Después nos metemos directo con la CPU. Luego el lector de cartuchos, la PPU (la unidad de gráficos), la GUI, el input, refinamiento general, y por último el sonido.

Cada sección tiene código. Hay un repositorio con commits por capítulo para que puedas seguir el progreso, pero te recomiendo que al menos la mitad de cada capítulo lo escribas vos, sin copiar y pegar. Es la diferencia entre entender cómo funciona y tener algo que funciona. Para las partes más repetitivas que no suman al entendimiento, ahí sí, ctrl+c ctrl+v sin culpa.

Todo el código está en inglés. Es una convención de la industria y además hace que el código sea más fácil de comparar contra la NES Wiki[1] que es donde está toda la documentación de como funciona la NES.

Y si al final querés jugar a un juego que necesita un mapper que no cubrimos, te dejo la tarea de implementarlo. Podés subirlo al repo como contribución si querés compartirlo con otros.

Feedback

Si encontrás errores, tenés sugerencias, o querés compartir tu progreso, podés contactarme por mail o Twitter. Los datos están al final del libro.

Dedicatoria

A los desarrolladores japoneses que programaron Super Mario Bros en assembler 6502, metiendo un juego completo en 32KB de memoria. Nosotros nos quejamos cuando una dependencia de npm pesa 50MB.

Empecemos.

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

Capítulo 1: Anatomía de una NES

En este capítulo vamos a entender qué tiene adentro una NES, qué hace cada parte, y cómo se va a traducir eso en código. Al final vas a tener claro el panorama general antes de meternos en los detalles.

Qué significa emular

Emular es mentir. Es convencer a un programa de que está corriendo en un hardware que no existe.

Un juego de NES es una secuencia de instrucciones escritas para un procesador específico (el 6502). Cuando Nintendo fabricaba cartuchos en los 80, esas instrucciones se ejecutaban en chips físicos. Nosotros no tenemos esos chips, pero podemos escribir software que se comporta igual.

La CPU del juego dice "sumá estos dos números y guardá el resultado acá". Nuestro emulador lee esa instrucción, hace la suma, y guarda el resultado en el lugar correcto de nuestra memoria simulada. El juego nunca se entera de que no está corriendo en una NES real.

Es como ser un traductor simultáneo, pero en vez de traducir idiomas, traducís instrucciones de hardware a operaciones de software. En tiempo real. 60 veces por segundo.

Un poco de contexto histórico

La NES (Nintendo Entertainment System) salió en Japón en 1983 como "Famicom". Llegó a América en 1985, disfrazada de "sistema de entretenimiento" porque la industria de videojuegos estaba tan muerta después del crash del 83 que nadie quería vender otra "consola de videojuegos".

Algunos números para dimensionar:

- **CPU**: 1.79 MHz (tu teléfono corre a ~3000 MHz)
- **RAM**: 2 KB (este párrafo pesa más que eso)
- **Resolución**: 256x240 píxeles, 54 colores posibles en pantalla

Con esas limitaciones, los desarrolladores hicieron Super Mario Bros, Zelda, Metroid, y Mega Man. El cartucho de Super Mario Bros pesa 32 KB. Una foto de tu almuerzo pesa 3 MB.

Cada vez que te quejes de que JavaScript es lento, acordate de que programadores japoneses hicieron scrolling parallax en 2 KB de RAM escribiendo en assembler.

Los órganos de la bestia

La NES tiene cuatro componentes principales que nos importan:

CPU (Central Processing Unit)

El cerebro. Un Ricoh 2A03, que es básicamente un MOS 6502 con algunas modificaciones. Ejecuta las instrucciones del juego: la lógica, la física, qué pasa cuando pisás un Goomba.

Tiene acceso a 2 KB de RAM para trabajar. Parece poco (y es poco), pero los cartuchos traían su propia ROM con el código del juego, así que esos 2 KB eran solo para variables temporales, posiciones de enemigos, estado del jugador, etc.

PPU (Picture Processing Unit)

Los ojos. Se encarga de dibujar todo lo que ves en pantalla: los fondos, los sprites, Mario, las nubes, los bloques.

Tiene su propia memoria: 2 KB de VRAM (Video RAM) para los fondos, más memoria adicional en el cartucho para los gráficos (los "tiles" que forman las imágenes). La PPU es su propia bestia. No recibe instrucciones de "dibujá a Mario en la posición X,Y". Es más primitivo: recibe patrones de 8x8 píxeles y los organiza en pantalla según unas tablas que la CPU configura. Es elegante y horrible al mismo tiempo.

APU (Audio Processing Unit)

Los oídos. Genera el sonido: los bleeps, los bloops, la música de los niveles. Lo vamos a dejar para el final del libro porque es independiente del resto y el emulador funciona perfectamente sin audio (en silencio, como programamos a las 3 AM).

Cartucho

El cartucho no es solo almacenamiento. Trae:

- **PRG-ROM**: El código del juego (las instrucciones que ejecuta la CPU)
- **CHR-ROM**: Los gráficos (los tiles que usa la PPU)
- **Mapper**: Circuitería adicional para juegos más grandes

Los primeros juegos entraban en el espacio de memoria de la NES sin problemas. Pero cuando los juegos se volvieron más ambiciosos, necesitaron más espacio del que la NES podía direccionar. Los mappers resuelven esto haciendo "bank switching": intercambian secciones de memoria del cartucho

según necesite el juego. Por ahora no te preocupes por los mappers. Vamos a arrancar con juegos simples (mapper 0) y después agregamos complejidad.

Así se conecta todo:

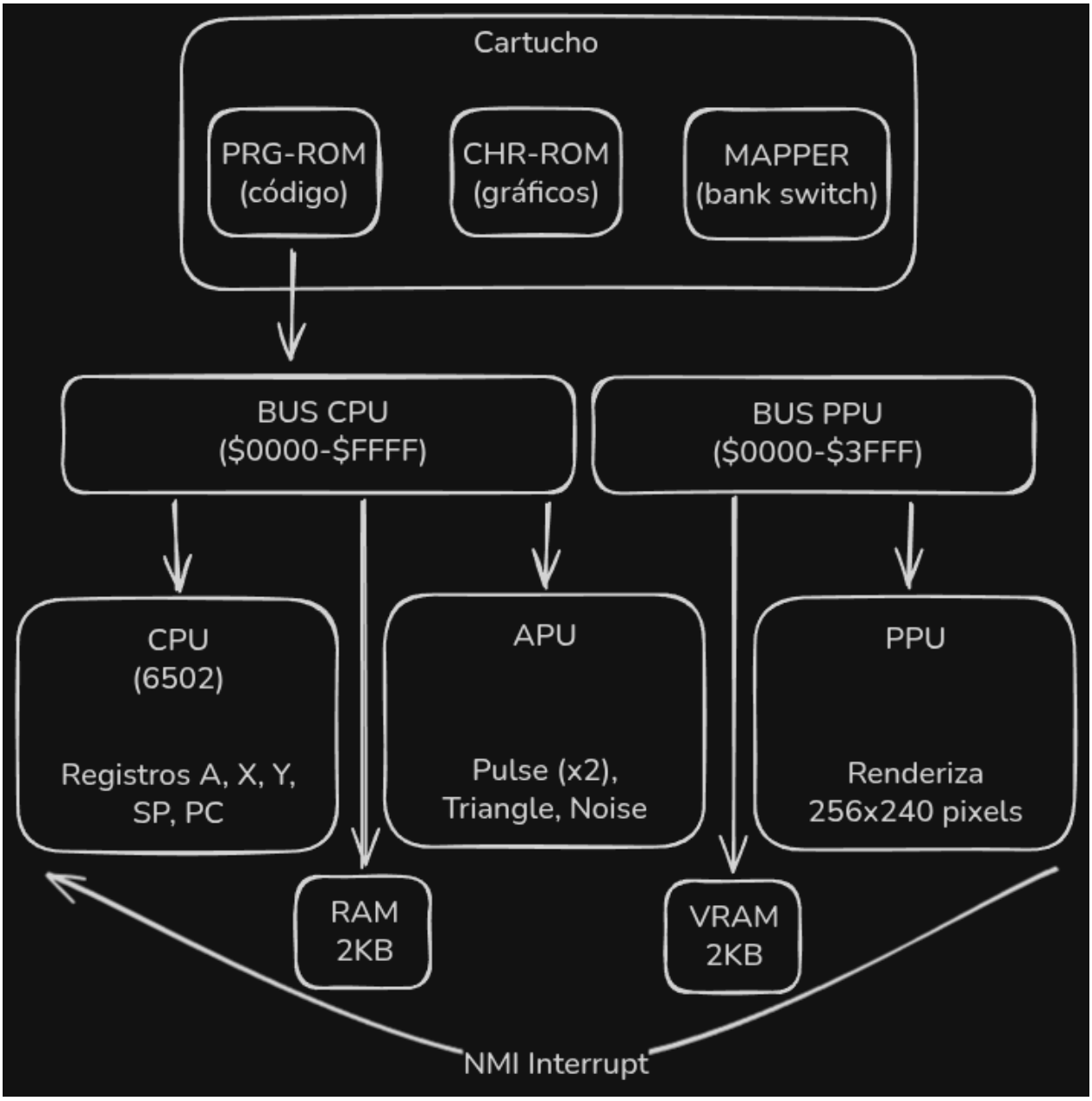


Figura 1. Digrama NES

La CPU ejecuta el juego y le dice a la PPU qué mostrar. La PPU dibuja frames. Los dos trabajan en paralelo, coordinados por un reloj maestro.

El loop del emulador

Acá está el corazón de lo que vamos a construir:

```ruby
loop do
  # 1. CPU ejecuta una instrucción
  ciclos_cpu = cpu.step

  # 2. PPU corre 3 ciclos por cada ciclo de CPU
  (ciclos_cpu * 3).times do
    ppu.step
  end

  # 3. Repetir hasta el infinito (o hasta que cierres el emulador)
end
```

Vamos por partes.

¿Por qué ciclos_cpu es una variable?

No todas las instrucciones de la CPU tardan lo mismo. Algunas son rápidas (2 ciclos), otras más lentas (hasta 7 ciclos). `LDA #$00` (cargar un valor en un registro) tarda 2 ciclos. `JSR $8000` (saltar a una subrutina) tarda 6.

Cuando llamamos a `cpu.step`, la CPU ejecuta una instrucción completa y nos devuelve cuántos ciclos tardó. Necesitamos ese número para saber cuántos ciclos de PPU correr.

¿Por qué 3 ciclos de PPU por cada ciclo de CPU?

La PPU corre a 5.37 MHz y la CPU a 1.79 MHz. Si dividís, te da ~3. Esto significa que por cada ciclo de CPU, la PPU avanza tres ciclos.

Si la CPU ejecuta una instrucción de 4 ciclos, la PPU tiene que avanzar 12 ciclos (4 × 3) para mantenerse sincronizada.

La simplificación que estamos haciendo

Hay dos formas de emular esto:

- **Tick-perfect (la difícil)**: Emulás cada ciclo individual. CPU hace un tick, PPU hace tres ticks. CPU hace otro tick, PPU hace otros tres. Así para cada ciclo de cada instrucción. Esto significa que tenés que trackear en qué paso de la instrucción está la CPU: "ya leí el opcode, ahora estoy leyendo el operando, ahora estoy calculando la dirección...". Una instrucción puede quedar ejecutada por la mitad entre un tick y el siguiente. Es lo que hacen los emuladores que buscan precisión absoluta, y es bastante más complejo de implementar.

- **Instrucción por instrucción (la nuestra)**: La CPU ejecuta la instrucción completa de una, y después la PPU "se pone al día" corriendo todos sus ciclos juntos. Es menos preciso pero mucho más simple de implementar y entender. Para el 99% de los juegos, nuestra aproximación funciona perfecto. Los casos donde falla son juegos que hacen trucos muy específicos dependiendo del timing exacto entre CPU y PPU, ciclo por ciclo. Esos juegos existen, Battletoads es famoso por hacer crashear emuladores en el nivel 2 si el timing no es exacto, pero no son la mayoría. Mario, Zelda, Mega Man, Contra, todos corren perfecto con nuestra aproximación.

Acá elegimos simplicidad sobre perfección. Y no es solo una decisión didáctica: muchos emuladores populares (incluyendo varios que encontrás online) usan exactamente esta estrategia. Funciona bien para la enorme mayoría de los juegos.

Por dónde empezamos

Vamos a construir la CPU primero.

¿Por qué? Porque es el componente más independiente. Podemos implementar la CPU completa y testearla sin necesitar nada más. Hay test ROMs que verifican que tu CPU funcione correctamente, instrucción por instrucción.

Una vez que la CPU anda, agregamos el cartucho para poder cargar juegos. Después la PPU para ver algo en pantalla. Después el input para poder jugar. Y al final el sonido, porque los bleeps pueden esperar.

Cada pieza que agregamos hace el emulador un poco más real. Es como armar un Frankenstein electrónico: primero el cerebro, después los ojos, después las manos.

Capítulo 2: Setup

Este es el capítulo más aburrido del libro, pero va a ser corto y es necesario para poder arrancar con la CPU. Vamos a instalarnos el entorno de desarrollo para codear en Crystal.

Editor

Mi recomendación es VSCode con la extensión Crystal Language. La encontrás buscando "Crystal Language" en el marketplace de extensiones.

¿Por qué VSCode? Porque es uno de los pocos editores con soporte decente para Crystal. El lenguaje es de nicho y no tiene la comunidad de Python o JavaScript, así que las opciones son limitadas. La extensión te da syntax highlighting, algo de autocompletado, y formateo. No es un IDE completo, pero alcanza.

Si preferís Vim, Emacs, u otro editor, hay plugins disponibles. Buscá "crystal" en el package manager de tu editor y seguramente encontrés algo.

Instalando Crystal

Hay varias opciones para instalar Crystal. La lista completa de opciones la podes encontrar en la pagina oficial de Crystal[1]. Yo personalmente recomiendo usar un package/version manager como `asdf`, `homebrew` o `scoop` (si estas en Windows), pero sentíte libre de instalarlo como te quede mas cómodo.

Compilar y ejecutar

Para asegurarte que tenes todo andando, crea un archivo `main.cr` del estilo:

```
1  # main.cr
2
3  puts "Hola bb" # Si, es igual a Ruby.
```

Ahora tenemos dos opciones para correr nuestro programa. Asegurate de probar las dos antes de saltar al próximo capítulo.

Opción rápida (interpretar)

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

```
crystal run main.cr
```

Esto compila y ejecuta en un solo paso. Es más lento porque compila cada vez, pero está bueno para desarrollo.

Opción para release (compilar)

```
crystal build main.cr -o hola --release
```

Esto genera un ejecutable llamado hola (o hola.exe en Windows). El flag —release activa optimizaciones, el binario es más rápido pero tarda más en compilar.

Finalmente para ejecutar hacemos:

```
# Linux o Mac
./hola

# Windows
hola.exe
```

Si todo salió bien, deberías ver “Hola bb” en tu terminal.

> Y listo! Ya tenés todo para escribir Crystal. Arranquemos con la CPU de una vez por todas.

Capítulo 3: CPU

La CPU es el cerebro de la NES. Es lo que lee instrucciones, las ejecuta, y le dice al resto del sistema qué hacer. Si el emulador fuera una cocina, la CPU sería el chef: lee la receta (el programa), usa los ingredientes que tiene a mano (los registros), y va paso a paso hasta que el plato está listo. Si se saltea un paso o lee mal una instrucción, el resultado es un desastre.

En este capítulo vamos a implementar la CPU completa. Cuando terminemos, vas a poder correr programas en assembler 6502 y ver cómo tu código los ejecuta instrucción por instrucción. Más adelante, cuando tengamos el lector de cartuchos, vamos a correr una ROM de test para verificar que todo funciona como corresponde.

El 6502

La NES usa una variante del MOS 6502, el mismo procesador que usaban la Commodore 64, la Apple II, y la Atari 2600. Era barato, simple, y para la época, bastante capaz.

El 6502 tiene 56 instrucciones oficiales. Digo "oficiales" porque hay un montón de instrucciones no documentadas que surgieron como efectos secundarios del diseño del chip. Algunos juegos las usan, la mayoría no. Nosotros vamos a implementar las 56 oficiales. Si después querés correr algún juego raro que usa las ilegales, te dejo la tarea.

Ahora, 56 suena manejable hasta que te enterás que esas 56 instrucciones tienen en total **151 opcodes** diferentes. Esto es porque una misma instrucción puede operar de distintas formas: a veces lee de memoria, a veces de un registro, a veces de una dirección que calculás sumando cosas. Cada combinación tiene su propio código. No te preocupes, vamos a ir de a poco.

Familias de instrucciones

Las instrucciones se agrupan en familias según qué hacen:

Familia	Qué hace	Ejemplos
Load/Store	Mueven datos entre registros y memoria	LDA, LDX, LDY, STA, STX, STY
Transferencia	Mueven datos entre registros	TAX, TAY, TXA, TYA, TSX, TXS
Aritméticas	Sumas, restas	ADC, SBC
Lógicas	AND, OR, XOR, shifts	AND, ORA, EOR, ASL, LSR, ROL, ROR
Comparación	Comparan valores y setean flags	CMP, CPX, CPY
Branches	Saltos condicionales	BEQ, BNE, BCS, BCC, BMI, BPL, BVS, BVC
Jumps	Saltos incondicionales	JMP, JSR, RTS, RTI
Stack	Push y pop al stack	PHA, PHP, PLA, PLP
Flags	Modifican flags directamente	CLC, SEC, CLI, SEI, CLV, CLD, SED
Otras	Varias	INC, DEC, INX, INY, DEX, DEY, BIT, NOP, BRK

No hace falta que memorices nada de esto ahora. La tabla está acá para que tengas el panorama y para que vuelvas a consultarla cuando estemos implementando.

Los registros

Si nunca trabajaste con arquitectura de computadoras, pensá en los registros como la memoria interna de la CPU. La RAM es la mochila: entra mucho, pero tenés que abrirla, buscar, sacar. Los registros son tus bolsillos: pocos y chicos, pero el acceso es instantáneo. Cuando vas a hacer una operación, traés los datos de la RAM a los registros, hacés lo que tenés que hacer, y guardás el resultado.

El 6502 tiene muy pocos registros. Esto era común en la época: la memoria era cara, los transistores eran caros, todo era caro. Te daban lo mínimo indispensable y vos te arreglabas.

- **Accumulator (A)** - 8 bits El registro principal. Casi todas las operaciones aritméticas y lógicas pasan por acá. Si querés sumar dos números, uno tiene que estar en A.
- **Index X e Index Y** - 8 bits cada uno Se usan para indexar memoria. Por ejemplo, "cargá el valor que está en la dirección 0x200 + X". También sirven como contadores en loops.
- **Program Counter (PC)** - 16 bits La dirección de la próxima instrucción a ejecutar. La CPU lee de donde apunta el PC, ejecuta, y avanza el PC. Los saltos y branches lo modifican para ir a otras partes del código.
- **Stack Pointer (SP)** - 8 bits que apuntan al tope del stack en memoria.
- **Status (P)** - 8 bits Este registro es especial: no guarda datos, guarda *estado*. Cada bit es una bandera que indica algo sobre lo que acaba de pasar:

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

- **N** (Negative): El resultado fue negativo (bit 7 = 1)
- **V** (Overflow): Hubo overflow en operaciones con signo
- **U** (Unused): Siempre en 1, no se usa
- **Z** (Zero): El resultado fue cero
- **C** (Carry): Hubo carry en una suma o borrow en una resta
- **I** (Interrupt Disable): Las interrupciones IRQ están deshabilitadas
- **D** (Decimal): Modo decimal. La NES lo ignora, pero el flag existe
- **B** (Break): La interrupción fue por software (BRK)

¿Para qué sirve esto? Para tomar decisiones. Las instrucciones de branch miran estos flags para decidir si saltan o no. Por ejemplo: `BEQ` (Branch if Equal) salta si el flag Z está en 1, o sea, si la última operación dio cero. Así es como implementás un `if` en assembler: hacés una comparación, que setea los flags, y después un branch que mira el flag correspondiente.

El Status es lo que le da "inteligencia" al programa. Sin él, la CPU solo podría ejecutar instrucciones en secuencia, sin nunca decidir nada.

Cómo funciona una CPU (versión corta)

Una CPU hace siempre lo mismo, en loop infinito:

1. **Fetch**: Lee el byte en la dirección que indica el PC (program counter)
2. **Decode**: Determina qué instrucción es y cómo obtener sus operandos
3. **Execute**: Ejecuta la instrucción
4. **Repeat**: Incrementa el PC según corresponda y vuelve al paso 1

Algunas instrucciones son de 1 byte (solo el opcode), otras de 2 (opcode + un operando), otras de 3 (opcode + dirección de 16 bits). El PC avanza lo que corresponda después de cada instrucción.

Por ejemplo:

```
PC = 0x0000

1. Leo el byte en 0x0000 -> es 0xA9 (instrucción LDA immediate)
2. Avanzo 1 el PC -> 0x0001
3. LDA necesita un operando, leo el byte en 0x0001 (PC) -> es 0x42
4. Avanzo 1 el PC -> 0x0002
5. Ejecuto: cargo 0x42 en el registro A
6. Vuelvo al paso 1, ahora con PC en 0x0002

(0x es un prefijo que indica que el número esta en formato hexadecimal)
```

Cada instrucción tarda una cantidad de ciclos en ejecutarse. Un `NOP` tarda 2 ciclos, un `LDA` puede tardar entre 2 y 6 dependiendo del modo de direccionamiento. Esto importa porque la PPU (el chip de gráficos) corre en paralelo y espera que la CPU tarde lo que tiene que tardar. Si tu emulador ejecuta todo instantáneamente, la sincronización se rompe. Por ahora no te preocupes por esto, lo vamos a manejar más adelante.

Estructura del proyecto a alto nivel

Esta va a ser nuestra estructura de archivos:

```
nes-emulator/
├── bin/
│   └── test_cpu.cr      # Entry point para probar
└── src/
    ├── nes.cr           # Módulo principal
    └── nes/
        ├── emulator.cr  # Coordina los componentes
        ├── bus.cr       # Comunicación entre componentes
        └── cpu.cr       # La CPU
```

La filosofía es simple: una clase por componente. A medida que avancemos, cada módulo de la NES (CPU, PPU, APU, etc.) va a tener su propio archivo. También vamos a crear archivos mas chicos y enfocados para lidiar con la complejidad y que todo sea mas fácil de leer y de debuggear.

El **Emulator** es el coordinador general. El **Bus** es el sistema de comunicación: en el hardware real, la CPU no habla directamente con la memoria o la PPU, todo pasa por un bus de datos. Replicamos eso porque hace el código más limpio y más fiel a la arquitectura real.

El contenido de estos archivos van a ser los siguientes por ahora:

```crystal
# src/nes.cr

require "./nes/emulator"

module NES
  VERSION = "1.0"
end
```

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

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

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

    def initialize # asi se le llama al constructor en Ruby y Crystal
        @bus = Bus.new
        @cpu = CPU.new(@bus)
    end

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

```crystal
# 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
```

El código para esta parte esta aquí[1]

Ahora corré esto para verificar que todo está bien:

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

Deberías ver “Whats up bro”

Si ves eso, estás listo. Si no, revisá que tenés Crystal instalado correctamente (volvé al capítulo 2 si hace falta).

Bueno ahora sí, vamos a hacer que la CPU haga algo útil.

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

Capítulo 3.1: Nuestras dos primeras instrucciones

Vamos a arrancar con dos instrucciones: `LDA` (Load Accumulator) e `INX` (Increment X). Son simples, pero suficientes para armar toda la estructura de la CPU y verificar que funciona.

Un poco de hexadecimal

Si no estás acostumbrado a hexadecimal, ahora es el momento. Vamos a usarlo en todas partes: direcciones de memoria, valores de registros, opcodes. Es el idioma del bajo nivel.

Hexadecimal es base 16. Después del 9 vienen A, B, C, D, E, F. Así:

Decimal	Hexadecimal	Binario
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

¿Por qué hexadecimal y no binario? Porque es más compacto. Un byte (8 bits) siempre se representa con exactamente dos dígitos hexadecimales. `0xFF` es más fácil de leer que `11111111`.

La calculadora de Windows (modo Programador) o cualquier calculadora online te convierte entre bases. Usala hasta que te acostumbres.

Tipos en Crystal

Crystal es tipado, y para trabajar con bytes necesitamos ser explícitos sobre el tamaño. Los sufijos que vas a ver todo el tiempo:

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

Por ejemplo:

```
valor = 0xA9_u8          # 8 bits, el opcode de LDA
direccion = 0x8000_u16   # 16 bits, una dirección de memoria
```

Si venís de lenguajes dinámicos como Ruby o Python, esto puede parecer verboso. Pero cuando estás emulando hardware, necesitás control exacto sobre cuántos bits ocupa cada cosa. Un registro de 8 bits tiene que ser de 8 bits, no de 64.

Los opcodes

Cada instrucción de la CPU tiene un código numérico llamado opcode. Cuando la CPU lee `0xA9` de memoria, sabe que tiene que ejecutar `LDA immediate`. Cuando lee `0xE8`, ejecuta `INX`.

Vamos a centralizar todos los opcodes en un módulo:

```
# 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` en binario es `10101001`. Eso es lo que la CPU real leía: una secuencia de unos y ceros que activaban ciertos circuitos. Nosotros simulamos ese comportamiento en software.

El hash `CYCLES` guarda cuántos ciclos tarda cada instrucción. Lo vamos a necesitar para sincronizar con la PPU más adelante.

La CPU

Ahora sí, la CPU en serio. Creá **src/nes/cpu.cr**. Vamos a ir construyéndolo de a poco.

Primero la estructura básica con los registros:

```
# 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)
```

Los `getter` son los registros que vimos antes: A, X, Y, SP, PC, y Status.

Ahora el `initialize`:

```
# src/nes/cpu.cr

    def initialize(bus : Bus)
      @bus    = bus
      @a      = 0_u8 # Acordarse que _u8 indica que es un UInt de 8 bits
      @x      = 0_u8
      @y      = 0_u8
      @sp     = 0xFD_u8
      @status = 0x24_u8
      @pc     = 0_u16 # Lo mismo aca, _u16 indica un UInt de 16 bits
    end
```

¿Por qué `@sp = 0xFD_u8`? Es el valor inicial que tiene el stack pointer cuando se prende el 6502.

¿Y `@status = 0x24_u8`? En binario es `00100100`. Eso setea los flags U (Unused, siempre en 1) e I (Interrupt Disable) como activos. Al igual que con el caso del `@sp`, es el estado inicial del procesador.

Ahora el corazón de todo, el método `step`:

```
# 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
```

Lee la proxima instrucción (opcode), la decodifica con el `case`, la ejecuta, y devuelve cuántos ciclos tardó. Simple.

A medida que implementemos más instrucciones, este `case` va a crecer. Por ahora solo tiene dos.

Finalmente, los métodos para leer de memoria:

```
# 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` lee el byte en la posición actual del Program Counter y avanza el PC.

Fijate el `&+=`. Es suma con overflow. Si el PC está en `0xFFFF` y sumamos 1, vuelve a `0x0000` en vez de tirar error. Así funciona el hardware real.

Las instrucciones

Vamos a organizar las instrucciones en módulos separados. Vamos primero con la instrucción `LDA`:

```
# 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` lee el siguiente byte del programa y lo guarda en el Accumulator. Después actualiza los flags Z (Zero) y N (Negative) según el valor cargado.

¿Cómo sé qué flags afecta cada instrucción? De la NES Wiki[1]. Cada instrucción tiene documentado exactamente qué hace y qué flags modifica.

Ahora `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` incrementa X en 1. Si X está en 255 y lo incrementás, vuelve a 0 (por el `&+`). También actualiza Z y N.

Y el módulo que junta todo, **Instructions**:

[1] https://www.nesdev.org/wiki/Instruction_reference#LDA

```cr
# 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
```

Este patrón de un archivo por familia de instrucciones nos va a salvar la vida cuando tengamos las 56 instrucciones implementadas. Archivos chicos, fáciles de encontrar.

Los flags

Los flags son bits individuales dentro del registro Status. Necesitamos poder prenderlos y apagarlos.

Primero definimos las máscaras, una por cada flag:

```cr
# 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 (siempre 1)
      FLAG_V = 0b0100_0000_u8  # Overflow
      FLAG_N = 0b1000_0000_u8  # Negative
```

Cada constante tiene un solo bit prendido en la posición correspondiente. `FLAG_Z = 0b0000_0010` tiene el bit 1 prendido.

¿Por qué se llaman máscaras? Porque las usamos para "tapar" todos los bits que no nos interesan y operar solo sobre uno específico. Es como una máscara de pintor: cubre todo menos la parte que querés modificar.

Ahora los métodos para setear los flags más comunes:

```
# 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` prende el flag si el valor es cero. `set_n_flag` prende el flag si el bit 7 (el más significativo) está en 1, lo que indica un número negativo en complemento a dos.

Y el método helper que hace el trabajo sucio:

```
# 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
```

El método `set_flag` usa operaciones bitwise:

- Para prender un bit: `OR` con la máscara (`|=`)
- Para apagar un bit: `AND` con la máscara invertida (`&= ~mask`)

Veamos un ejemplo concreto. Queremos prender el flag Z (Zero) en un status que está en `0x00`:

```
status = 0000 0000
FLAG_Z = 0000 0010

status | FLAG_Z = 0000 0010  # <= El bit 1 quedó prendido
```

Ahora queremos apagar ese flag:

```
status   = 0000 0010
FLAG_Z   = 0000 0010
~FLAG_Z  = 1111 1101  # <= Invertimos todos los bits

status & ~FLAG_Z = 0000 0000  # <= El bit 1 quedó apagado
```

El `OR` prende bits, el `AND` con máscara invertida los apaga. Una vez que lo ves en acción, tiene sentido.

El Bus

La CPU no accede a la memoria directamente. Todo pasa por el Bus. Primero definimos las constantes de direcciones:

```
# 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
```

¿Qué es el mirroring? La NES tiene 2KB de RAM (direcciones `0x0000` a `0x07FF`), pero el espacio de direcciones de la CPU va hasta `0x1FFF` para RAM. Las direcciones `0x0800` a `0x1FFF` son "espejos" de los primeros 2KB.

Si escribís en `0x0800`, en realidad estás escribiendo en `0x0000`. Si escribís en `0x1000`, también es `0x0000`. Es un truco del hardware para simplificar el decodificador de direcciones.

El `RAM_MASK = 0x07FF` nos sirve para calcular la dirección real. `0x0800 & 0x07FF = 0x0000`. Magia de bits.

Ahora el Bus en sí:

```
# src/nes/bus.cr

require "./bus/addresses"

module NES
  class Bus
    include Addresses

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

La RAM es simplemente un array de bytes inicializado en cero. `Bytes.new(2048, 0_u8)` nos da 2KB de memoria lista para usar.

Los métodos de lectura y escritura:

```
# 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
```

¿Por qué `cpu_read` y no solo `read`? Porque más adelante la PPU también va a leer y escribir del Bus, pero con reglas distintas. Vamos a tener `cpu_read`, `cpu_write`, `ppu_read`, y `ppu_write`. El prefijo nos ayuda a distinguir quién está accediendo.

El `case` determina qué componente maneja esa dirección. Por ahora solo tenemos RAM, pero acá es donde después vamos a agregar PPU, APU, cartuchos, etc.

Y los métodos internos que aplican el 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
```

El `& RAM_MASK` es donde ocurre la magia del mirroring. `0x0800 & 0x07FF = 0x0000`. Cualquier dirección en el rango de mirrors se convierte automáticamente en su dirección real.

El Emulator

El Emulator coordina todo:

```
# 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
```

Nada loco: crea un Bus, crea una CPU conectada a ese Bus.

El método `step` ejecuta un paso de la emulación:

```
# src/nes/emulator.cr

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

Por ahora solo avanza la CPU. Cuando tengamos la PPU, acá es donde vamos a hacer que corra 3 ciclos por cada ciclo de CPU.

Y un helper para cargar programas de prueba:

```
# 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` toma un array de bytes y los escribe en memoria a partir de una dirección. Así podemos cargar programitas de prueba sin necesitar un cartucho todavía.

Debugging

Para ver qué está pasando internamente, vamos a agregar un módulo de debugging al 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` imprime el estado completo de la CPU en una línea. Lo vas a usar después de cada `step` para ver cómo cambian los registros.

Los helpers para formatear valores en hexadecimal:

```
# 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` formatea un byte como `$0A`, `hex16` formatea una dirección como `$8000`. El `$` es convención de assembler 6502 para indicar hexadecimal.

Y para mostrar los flags de forma legible:

```
# 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
```

Mayúscula = flag activo, minúscula = flag inactivo. `nvUbdIzc` significa que solo U e I están prendidos (el estado inicial de la CPU).

El módulo se incluye en el Emulator con `include Debugging`, así que podés llamar `emu.debug` directamente.

Tests

Vamos a tener un test por instrucción. Se pueden correr individualmente o todos juntos.

El runner principal:

```
# bin/test_cpu.cr

require "./cpu/test_lda"

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

require "./cpu/test_inx"
```

Cada `require` ejecuta el archivo de test correspondiente. Simple.

Veamos el test de LDA en **bin/cpu/test_lda.cr**. Primero la estructura:

```
# bin/cpu/test_lda.cr

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

puts "LDA Immediate"

all_passed = true
```

Cargamos el emulador y preparamos una variable para trackear si todos los tests pasan.

Un test típico se ve así:

```
# 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
```

Creamos un programa de 2 bytes: el opcode `0xA9` (LDA immediate) y el valor `0x42`. Lo cargamos, ejecutamos un step, y verificamos que el registro A tenga el valor correcto y que los flags estén bien.

El test completo también verifica el flag Zero (cargando `0x00`) y el flag Negative (cargando `0x80`). Misma estructura, distintos valores.

Al final:

```
puts all_passed ? "\n✅ LDA passed" : "\n❌ LDA failed"
```

El test de INX en **bin/cpu/test_inx.cr** es similar. Verifica:

1. Incremento simple (X pasa de 0 a 1)
2. Wrap around (X pasa de 255 a 0, flag Z se prende)
3. Flag Negative (X llega a 128, bit 7 prendido)

Aquí esta el link[2] al repositorio donde podes encontrar los archivos con los tests completos.

Corré `crystal run bin/test_cpu.cr` y deberías ver todos los tests pasando. Si no, tenés un bug. Bienvenido a la emulación.

Estructura final

Para asegurarte de que tenés todo, así debería verse tu proyecto:

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

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

Si algo no te funciona, podés comparar con el código del repositorio: Github[3]

Wrap up

Felicitaciones, tu CPU sabe hacer dos cosas: cargar un número en A y sumarle 1 a X. Es como tener un bebé que aprendió a decir “mamá” y “papá”. No es mucho, pero es honesto.

Ahora viene la parte divertida (o tediosa, depende cómo lo mires): implementar las otras 54 instrucciones. Yo personalmente me aburrí a la mitad, así que no te culpo si te pasa lo mismo.

Vamos a implementar juntos suficientes instrucciones como para que puedas implementar el resto vos solo. Mi recomendación: hacé al menos 10 o 15 por tu cuenta antes de copiar. Es donde realmente internalizás cómo funciona la CPU, y después el resto es más de lo mismo. Una vez que sientas que ya entendiste el patrón, te dejo el link al repositorio con todas las instrucciones implementadas para que puedas avanzar a la próxima sección.

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