CPU RIA/1: The Adder

we will built our undestanding by starting with a minimal core. let's first construct an emulator that only supports a single instruction, ADDITION

- Becoming familiar with new terminology

- how to interpret opcodes

- undestanding the main loop

TERMS RELATED TO CPU EMULATION

dealing with CPUs and emulation involves learning some terms. take a moment to look at and undestand the following

- an operation (op) refers to procedures taat are supported natively by the system. you might also encounter equivalent phrases such as implemented in hardware

or intrinsic operation as yu explore further

- Registers are containers for data that the cpu accesses directly. for most operations, operands must be moved to registers for an operation to function.

for the CHIP-8, each register is a u8 value

- An opcode is a number that maps to an operation. on the CHIP-8 platform, opcodes include both the operation and the operand's registers

DEFINING THE CPU

the first operation we want to support is addition. the operation takes 2 registers (x and y) as operands and adds value stored in y to x.

so far the CPU is inert. to perform addition we will need to take the following steps, but there is no ability to store data in memory as yet.

1, initialize a CPU

2, load u8 values to registers

3, load the addition opcode into current\_operation

4, perform the operation

the constant "0x8014" - is the opcode that the CPU will interpret. to decode it, split it into four parts

- 8 signifies that the operation involves 2 registers

- 0 maps to cpu.registers[0]

- 1 maps to cpu.registers[1]

- 4 indicates addition

UNDERSTANDING THE EMULATOR' MAIN LOOP

now we have loaded the data. the CPU is almost able to do some work. the run() method performs the bulk of our emulator's work.

using the following steps, it emulates CPU-cycles

1, reads the opcode (eventully from memory)

2, decodes instructions

3, matches decoded instructions to known opcodes

4, dispatches execution of the operation to a specific function

HOW TO INTERPRET CHIP-8 OPCODES

it is important for our cpu to be able to interpret its opcode (0x8014). this section provides a throught explanation of the process used in the CHIP-8

and its naming conventions

CHIP-8 opcodes are u16 values madeup of 4 nibbles. A nibble is half of a byte(4 bit). b/c there is nota 4bit type in RUST, splitting u16 values into those parts is fiddly(complicated)

to make matters more complicated, CHIP-8 nibbles are often recombined to form either 8-bit or 12-bit values depending on the context

to simplify talking about the parts of each opcode, let's introduce some standard terminology. each opcode is made up of 2 bytes: the "high byte" and the "low byte".

And each byte is made up of 2 nibbles. the "high nibble" and the "low nibble", respectly.

0 x 7 3 E E

7 3 - high byte(u8) -> 7 - high nibble(u4) 3 -> low nibble(u4)

E E - high byte(u8) -> E - high nibble(u4) E -> low nibble(u4)

Opcode type: 7XNN → add immediate

X: 3 → V3

NN: 0xEE → 238 decimal

Instruction:

V3 = V3 + 0xEE (no carry flag)

Explanation:

Adds an immediate value (NN) to a register.

Unlike 8XYN, no second register is involved.

0 x 8 2 3 1

CHIP-8 uses nibble-based opcode encoding, and arithmetic/logical instructions are grouped under opcodes starting with 8 (8XYN). The last nibble (N) decides the specific operation.

Here’s the table for 8XYN again:

N Operation

0 Vx = Vy

1 Vx = Vx OR Vy

2 Vx = Vx AND Vy

3 Vx = Vx XOR Vy

4 Vx = Vx + Vy (carry)

5 Vx = Vx - Vy (borrow)

6 Vx = Vx SHR 1

7 Vx = Vy - Vx

E Vx = Vx SHL 1

\*\*\* To extract nibbles from bytes, we will ned to use the right shift(>>) and logical AND (&) bitwise operations.

for NNN and KK, you don’t need shifting at all, because they’re already sitting at the lowest bits of the opcode.

Why?

NNN = opcode & 0x0FFF

0x0FFF in binary = 0000 1111 1111 1111

Mask keeps the lowest 12 bits → address value

No shifting needed, because we want those bits as-is

KK = opcode & 0x00FF

0x00FF in binary = 0000 0000 1111 1111

Mask keeps the lowest 8 bits → immediate constant

No shifting needed, because KK already lives in the last two hex digits

Compare with X, Y, N

For these, the nibble is not aligned at the bottom — so you need a shift and a mask:

let x = (opcode >> 8) & 0xF; // second nibble

let y = (opcode >> 4) & 0xF; // third nibble

let n = opcode & 0xF; // last nibble

✅ Rule of thumb:

Use mask only for NNN and KK (since they’re already lowest bits).

Use shift + mask for X, Y, N (since they’re in the middle of the opcode).

THE ADDER -is simple, just add what we have in our registers and put it in the first register

THE MULTIPLIER - adder can execute a single instruction: addition. multiplier, can execute several instructions in sequence. the multiplier includes RAM, a working main loop,

and a variable that indicates which instruction to execute next that we'll call position\_in\_memory.

how it works???

- Adds 4kb of memory

- includes a fully fledged main loop and stopping condition

at each step in the loop, memory at position\_in\_memory is accessed and decoded in to an opcode. position\_in\_memory is the incremented to the next memory address,

- removes the current\_instruction field of the CPU struct, which is replaced by section of the main loop that decodes bytes from memory

-writes the opcodes into memory

EXPANDING THE CPU TO SUPPORT MEMORY

we need to implement some modifications to make out CPU more useful. to start, the computer needs memory.

some features of the CPU are quite novel:

-having 16 registers means that a single hexadecimal number (0 to F) can address those. that allows all opcodes to be compactly represented

as u16 values

-the CHIP-8 only has 4096 bytes of RAM (0x1000 in hexadecimal). this allows CHIP-8's equivalent of a usize type to only be 12 bit wide: 2^12 = 4096.

those 12bits become the nnn variable discussed earlier

what I deviates from the standard practice in 2 ways

- what we call the "position\_in\_memory" is nomally reffered to as the "program counter"

- within the CHIP-8 specification, the first 512 bytes(0X200) are reserved for the system, while other bytes are available

for programs. this implementation relaxes that restrction.

READING OPCODES FROM MEMORY

with the addition of memory within the CPU, the read\_opcode() method requires updating.

HANDLING INTEGER OVERFLOW

within the CHIP-8, we use the last register as a carry flag. when set, this flag indicates that an operation has overflowed the u8

register size

THE CALLER

we have nearly built all of the emulator machinery. this section adds the ability for you to call functions. there is no a programming language support,

however so any programs stil need to be written in binary.in addition to implementing functions, this section validates an assertion made at the start - functions are also data

EXPANDING THE CPU TO INCLUDE SUPPORT FOR A STACK

to build functions, we need to implement some additional opcodes. these are as follows:

- the CALL opcode (0x2nnn, where nnn is memory address) sets position\_in\_memory to nnn, the address of the function.

- the RETURN opcode (0x00ee) sets the position\_in\_memory to the previous CALL opcode

to enable these to opcodes to work together, the CPU needs to have some specialized memory available for storing addresses.

this is known as STACK. each opcode adds an address to the stack by incrementing the stack pointer and writting nnn to that position in the stack.

each RETURN opcode removes the top address by decrementing the stack pointer.

DEFINING A FUNCTION AND LOADING IT INTO MEMORY

within computer science, a function is just a sequence of bytes that can be executed by a CPU. CPUs start at the first opcode, the make their way to the end.

1, define a function. our function performs two addition operation and then returns - modest, yet informative. it is 3 opcodes long. the function

internals look like this in a notation that resembles assembly language:

add\_twice:

0x8014

0x8014

0x00EE

2, convert opcodes into Rust data types. translating these 3 opcodes into Rust's array syntax involves wrapping them in square brackets and using a comma for each number.

the function has now become a [u16; 3]:

let add\_twice: [u16; 3] = [

0x8014,

0x8014,

0x00EE,

]

3, we want to able to deal with one byte in the next step, so we'll decompose the [u16; 3] array further into a [u8; 6] array

let add\_twice:[u8; 6]=[

0x80, 0x14,

0x80, 0x14,

0x00, 0xEE

]

4, load the function into RAM. assuming that we wish to load the function into memory address 0x100,

here are 2 options.

first, if we have ou function available as a slice, we can copy it accress to memory with the copy\_from\_slice() method.

an alternative approach that achieves the same effect within memory without requring a temporary array is to overwrite bytes directly

IMPLEMENTING THE CALL AND RETURN OPCODES

calling a function is three-step process:

1, Store the current memory location on the stack

2, Increment the stack pointer

3, set the current memory location to the intended address

returning from a function involves reversing the calling process

1, decrement the stack pointer

2, retrieve the calling memory address from the stack

3, set the current memory location to the intended memory address

/\*

GREAT ANSWER FROM CHATGPT

me : so call means push the memory of the next instruction to the stack and return means go back to that memory address??

ChatGPT said:

Exactly! ✅ You’ve got it.

Let me restate it clearly:

CALL instruction

- Take the address of the next instruction (after the CALL) — this is where you’ll return later.

- Push that address onto the stack.

- Jump to the function’s memory address to start executing it.

RET instruction

- Pop the address from the stack.

- Set the program counter (position\_in\_memory) to that address

- Continue execution from there — exactly where the CPU left off before the CALL.

🔹 Analogy

CALL → “Go do this function, but I’ll mark my place with a bookmark.”

RET → “Take the bookmark and continue from there.”

This is why in your example:

First CALL 0x100 pushes 0x004 → jumps to 0x100.

RET pops 0x004 → resumes at 0x004.

No infinite loop occurs because we return to the instruction after the CALL, not the CALL itself.

If you want, I can make a tiny diagram showing stack changes and PC for each instruction in your code, so this “push next instruction / pop return address” idea becomes super visual.

\*/