Write Your Own Gameboy Emulator

Jeremy Cochoy

Contents

Chapter 1

Generalities on computers

Why writing a Game Boy emulator

Why would you write a Nintendo Game Boy emulator? I mean, it's a really old device. Nintendo released it in 1989 in japan. Most of the game on this console are now outclassed by today's game, and only few old nostalgic only play those games. Furthermore, one can see more than two hundred project of such emulators on github. Knowing that there already exist really good Game Boy and Game Boy Color emulators like BGB Gambatte that aim at reproducing accurately the game boy hardware's behavior, why would you join this run years after the start was given?

The reason become clear, once you ask the right question. What is the right question? Let me state it black on white:

> Why would you write a 8-bit computer emulator?

Because, this is exactly what is a Nintendo Game Boy. It's a computer, with all the components you can find in today's computer. It is just a little bit older, and a lot simpler and easier to understand than today's computer. And since our computers are a lot more faster, you don't have to worry too much about speed and performance of a Game Boy emulator implementation. To state it simply, a Game Boy emulator is one of the best project – if not the best – to jump in the world of computer emulation.

If you like learning, if you want to understand deeper what is the GPU for, how the CPU schedule all the hardware components, and the importance of memory mapping, welcome to this book. You are about the start one of the most intense journey of you life. Take some coffee, and sit comfortably. Let's start.

Prerequisites

Through this book, I assume you are already familar with a programming language (preferably an imperative one). If you know C, Rust, C++, or even Java, C#, JS or python you are fine. Strongly typed functionnal languages (like Haskell) can easily make this exercice a lot harder (debugguing becomes hard whereas the type system do not provide an easy way to enforce the right comportement through function types).

This book contain code samples written in Rust. You don't need to know this language to follow this book. Examples can be easily adapted to languages previously cited, and Rust remain very close to C by it's syntax, which was then inherited by many today's languages.

Although it is usually not recommended learning a new language while writing an emulator, it is indeed what many peoples actually does. If you want to learn Rust while building your emulator, it is something realizable, and have already been done by some persons. Just ask yourself if you feel confident enough to learn a language at the same time, or if you are new to the world of emulator and want to learn thing once at a time. If all the points that will be treated in this section seams new to you, you might prefer to stick to a language you already know.

Hexadecimal and binary systems

Let's recall a few things about numbers and numeric systems.

The number twenty-three is written by two digits, a `2` and a `3` from left to right. Since we use the *decimal* system, it means that the first digit count how many packets of 10 objects we have, and the second how objects we have. Formally `23` states for *two times ten and 3*. Since we use the decimal system every day, we forget (or just didn't know) this details.

But what happen if we want to count in packet of 16, 8, or 2? We name this new counting systems respectively hexadecimal, octal and binary. To count in hexadecimal, the first digits are `0`, `1`, `2`, …, `9`, `A`, `B`, `C`, `D`, `E`, `F`. We introduce the six missing symbols in order to reach the number 15. And so, what do we do once we reached `F`? Like in the decimal system, we add a new column, on the left. In hexadecimal, we have : `F`, `10`, `11`, `12`, …, `19`, `1A`, `1B`…

That is, you know how to count in hexadecimal. But wait, if you see `10`, does it mean ten in decimal, or sixteen written in hexadecimal?

Only the person who wrote this number can tell you. In order to tell people the quantity is an hexadecimal number, we usually prefix hexadecimal numbers by `0x`. This way, we know that `0x10` stands for sixteen and `10` stands for ten.

But why using this strange numbering while we have our good old decimal system? Before answering this question, let me show you the binary system.

In binary, we only need two digits. We choose to keep `0` and `1`. Counting goes: `0`, `1`, `10`, `11`, `100`, `101`, `110`, …

Not really economic in symbol. But the trick with this numbering system is that the value of each column can be represented by an electric wire with a voltage of zero (`0`) or a positive voltage (`1`).

That is actually how numbers are stored in computer. Usually, we like to pack binary digits by height. A digit in a binary number is called a bit. As before, we add the prefix `0b` before binary number to distinguish them from decimals. For example, `0b00000111` means seven.

What is the biggest number you can write with four digits? It's fifteen, `0b1111` in binary or `0xF` in hexadecimal. If we look at pocket of heights bits, usually called bytes, we get two symbols in hexadecimal. Instead of writing `0b01110001` we can just write `0x71` which is a lot easier to read.

Binary operation like *and* means that you take your two numbers, you write them in binary one above the other, and you keep only the column that both have `1`.

Example:

```
     0b11000111
AND  0b01001010
=    0b01000010
```

We usually write this operation `0xC7 & 0x4A = 0x42`.

There is other binary operations like *or* (it is enough to have a one in a column to keep it) noted with the pipe symbol `|`. Some others are the negation (replacing `0` by `1` and `1` by `0`), binary shift (shift the digits on the right or on the left) and binary rotation (just rotate the bits like if they were written around a cylinder). A last one, widely used, is the *xor* of two numbers, noted `^`. It stands for *eXclusive or*, and means that you keep a `1` in a column only if the two numbers have different digit. If both have the same digit, you place a `0`.

Get familiar with this operations if its not already the case.

Architecture of an 8-bit computer

Before writing the first line of code, lets describe the elementary block of hardware you would find in the console, and how they relate to each other. Surely, this organization will influence the design of your implementation.

The CPU

The Central Process Unit, also known as CPU for short, is the heart (actually, the brain) of a computer. A processor does three things. First int read an instruction from memory. Then, it decodes this instruction. It means reading the right number of arguments of the instruction from memory. Finally, it executes the instruction. The instruction can be adding two numbers, loading some value from memory at a specified location, writing at this location, and many other elementary actions.

Those free actions are called together the fetch–decode–execute cycle, or instruction cycle.

The CPU keeps repeating these cycle from start to the end of world (or more realistically, until the end of your battery).

To perform his job, the CPU has access to a set of internal registers. Some store data, other address, and some are used to control the processor.

A list of the Game Boy CPU's register can be found in annex A. Let describe the ones you are likely to find in any computer.

Working registers

The *working registers* are small variables contained inside the CPU, that can accessed without cost. The size of the registers are linked to the architecture of the processor. An 8 bit processor have registers of size 8bits, whereas 32 bits and 64 bits architecture have respectively registers of size 4 bytes and 8 bytes. Usually, there is an *Accumulator* register, often called `A`, which is used to store the result of arithmetic and binary operations. Often we can also find a register `C` called the *Counter* which is often used for… yes, counting, you get it.

Program counter

The *program counter* is a special register which store the address of the next instruction to fetch. It can be controller thanks to instructions like `JUMP`, `JNZ`, `CALL` and `RET` which change its values for conditional branching or function call.

Stack pointer

The *stack pointer* is a register which store the address of the stack. The stack is a location in the memory where registers can be *pushed* (wrote) and *poped* (restored, and the value is thrown). The data structure is a stack, which means that you have to retrieve the values in the reverse order you placed them. The stack is used right after a function call in order to save the values of the registers the function is going to use. Before quitting the function, the values are retrieved from the stack. The stack can also be used for storing any kind of intermediate value when there is not enough registers.

Status register

Status registers contains two kinds of values. First, it contains values that allow to control the processor. An example is the enabling and disabling of interrupts, controlled through the interrupt flag (a flag is a one bit register). The seconds are values affected by the result of the previous instruction (carry, flag that indicate if the result was zero, …). The first type of values are controlled through dedicated instructions. For example, the instruction `EI` set the interrupt flag to on. The second type are used by some instructions like conditional jumps.

The memory

The memory is another central part of the hardware. The cpu communicate with the memory through the memory BUS (from the Latin word omnibus). The memory bus is a set of conductors (usually directly drawn on the circuit) that allow the communication between the memory chips and the CPU. To retrieving a specific byte, a number called the address is sent by the CPU to the memory chips. The memory allow storing the instruction executed by the processor, but also the graphics and divers data of the games.

IO

IO Stands for Input and Output. It's basically any communication from the processor to any other device including the screen, and the joypad. In the Game Boy console, IO are accessed through the strategy of memory mapping. Memory mapping is allowing other devices to communicate with the CPU by sharing the address buss with them. Specific ranges of addresses are routed to devices instead of the memory chips. For example, the memory of the GPU (Graphic Process unit) corresponding to the tiles displayed by the Game Boy is mapped to certain range of addresses. To change the display of the screen, all the CPU has to do is to write to this specific address at the right timing. For knowing which buttons of the joypad are pressed by the player, the CPU can read a byte from a specific address.

Interupts

Interrupts are signal, either from the CPU itself (when an internal counter overflow) or external devices (like the LCD screen of the Game Boy) that tell the CPU to stop the normal flow of execution When the Game Boy's processor receive an interruption, it finishes his instruction cycle, push the current value of the Pointer Counter register on the stack, and jump to a specific address corresponding to the interruption handler for this instruction. This block of code, named the interruption handler, is executed until the processor reach a specific instruction that tells him to go back to the normal flow of execution.

Chapter 2

The Memory Unit

Address Space

The memory bus of the Game Boy allows to address up to 65 536 bytes. Some part of these addresses correspond to read only memory (ROM), others to read and write memory (RAM, for random access memory) and some are memory mapped devices.

Bellow you can find a short scheme summarizing the distribution through the address space.

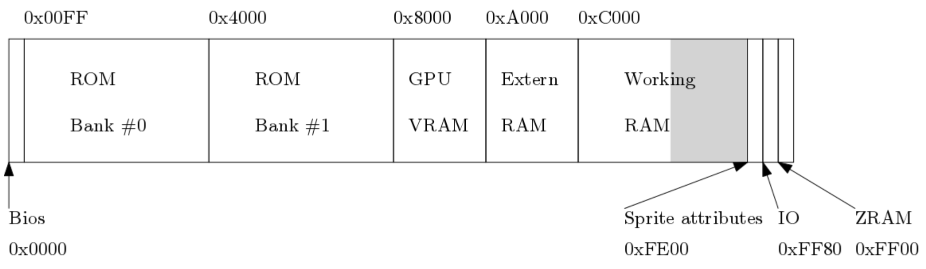

Figure 2.1: The memory strip

Many of those names might sounds strange or even meaningless. We describe each of these blocks, and give a quick summary for their use.

- **0x0000-0x00FF** BIOS:

The bios is the sequence of instructions executed immediately after the Game Boy's power on. It displays the Nintendo Logo, check for the signature of the cartridge, and initialize divers registers. Once the Pointer Counter reach the address 0X0100, the bios is not accessible anymore, and the address range is then mapped to the memory 0x0000-0x0100 of the cartridge's ROM.

- **0x0000-0x3FFF** and **0x4000-0x7FFF** ROM Bank:

The ROM is the read only part of the cartridge's memory. It contains the code of the game (starting at address 0x0100), and the graphics that will be loaded into the Game Boy's memory.

- **0x8000-0x9FFF** Video RAM:

The Video RAM is the memory used by the GPU for drawing tiles and divers graphics on the Game Boy's screen.

- **0xA000-0xBFFF** Cartridge's RAM:

  There is only few memory in the Game Boy console. Since some games might require more RAM, this address range can refer to RAM physically stored in the cartridge. It can be read and write at almost any time by the CPU.

- **0xC000-0xDFFF** Working RAM:

The Game Boy's internal RAM. Can be write and read at almost any time by the CPU.

- **0xE000-0xFDFF** Shadow RAM:

Due to the internal wiring, the Working RAM is also available on this address space. Reading and writing in this address range have the same effect as reading or writing 8k bytes below. Notice that the last 512 bytes of the Working RAM are not duplicated.

- **0xFE00-0xFE9F** Object Attribute Memory (OAM):

The game boy can handle up to 40 sprites located at different locations of the Game Boy screen, for rendering characters, falling bricks in Tetris, etc. Their information is stored in this memory.

- **0xFF00-FF7F** I/O:

The state of the console pad, and many values controlling the hardware behavior (sound, graphics, link cable...) are available at these addresses.

- **0xFF80-0xFFFE** High RAM:

The high RAM, also referred as the Zero Page, is a high speed RAM area. Most of the interactions between the hardware and the program occurs in this address range.

The MMU

Now we are happy, because we have an idea of the use of each block of this address space. But how should we implement the read and write access to this memory? Remember that we also have to memory mapped devices which require a specific code. As you can imagine, reading the state of the joypad is a bit different that just storing bytes in an array.

Since the processor cannot make a distinction between reading from IO mapped device and simply the RAM, an efficient way is to implement two procedure. The first one, `read_byte` (`rb` for short) will allow reading from a specific memory address, wereas `write_byte` (`wb` for short) will give a way to send values through the memory bus.

We will also have to store the divers memories (VRAM, RAM, ROM…) that can be accessed. We will store this bunch of byte in arrays of bytes, all of them contain in a structure. To handle the different effect of reading to address below 0x0100 when the bios is active, we also add a boolean.

All this things together (this structure, and the read and write functions) will be called the memory unit, known as mmu for short.

Here is a rust code that give a basic skeleton for our MMU.

```rust
#[derive(PartialEq, Eq, Clone, Debug)]
// The MMU (memory)
struct Mmu {
    // GB Bios
    bios  : Vec<u8>,
    // 0000-3FFF    16KB ROM Bank 00
    rom   : Vec<u8>,
    // 4000-7FFF    16KB ROM Bank 01
    srom  : Vec<u8>,
    // 8000-9FFF   Video RAM
    vram  : Vec<u8>,
    // A000-BFFF    8KB External RAM
    eram  : Vec<u8>,
    // C000-CFFF    4KB Work RAM Bank 0 (WRAM)
    wram  : Vec<u8>,
    // D000-DFFF    4KB Work RAM Bank 1 (WRAM)
    swram : Vec<u8>,
    // FE00-FE9F    Sprite Attribute Table (OAM)
    oam   : Vec<u8>,
    // FF80-FFFE    High RAM (HRAM)
    hram  : Vec<u8>,
    // When true, reading below 0x100 access the bios.
    // Once the booting sequence is finished, the value is
    // turned to false. Then, rading below 0x100 read bytes from the rom field.
    bios_enabled : bool,
}
```

As you can see, type are given after the name of the field, separated by a colon. But the syntax remind close to the syntax from the `C` language. The derive instruction allow to inference automatically the list of traits mentionned. If you wonder what is a trait, you will find the answer in few lines.

We can give default values to this structure. For example, we can initialise `bios` to contain the instructions from the Game Boy's boot sequence. You can find the bios in Annex C, or many websites dedicated to Game Boy development.

In `C++`, default values are set thanks to a constructor. In rust, default values use the default trait. Traits are a way to define a collection of polymorphic functions (similar to Java's interface) that can be implemented (alltogether) for an arbitrary data type. They are a tool to implement *polymorphic functions*, and can also be saw as a way to accomplish *overloading*. Concretely, to create a sutructure with default values, you implement the `Default` trait for the type `MMU`. The default trait have only one function, namely `default()`. Once you implemented the trait, you can create a structure with the line

```
let my_struct : MMU = Default::default();
```

The choice of wich default function is called depend on the return type expected. Here, we ask explicitely for a type `MMU`.

We can implement the default trait il the following way:

```
impl Default for Mmu {
    fn default() -> Mmu { Mmu {
        bios : vec![
            0x31, 0xFE, ... // See Annex C
        ],
        rom   : empty_memory(0x0000..0x4000),
        srom  : empty_memory(0x4000..0x8000),
        vram  : empty_memory(0x8000..0xF000),
        eram  : empty_memory(0xA000..0xC000),
        wram  : empty_memory(0xC000..0xD000),
        swram : empty_memory(0xD000..0xE000),
        oam   : empty_memory(0xFE00..0xFEA0),
        hram  : empty_memory(0xFF80..0xFFFF),
        ier   : Default::default(),
        ifr   : Default::default(),
        bios_enabled : true,
    }
    }
}

// Replace the each element of the list by a null byte
```

```
fn empty_memory<I : Iterator>(range : I) -> Vec<u8> {
    range.map(|_| 0).collect()
}
```

Here, the `empty_memory` function simple compute an empty vector of the length of the list it receive, and `0..100` simply compute the list of 100 elements `0, 1, ..., 99`. It is just a fancy way to create a vector of a specific length without having to compute explicitely his length, which is the difference of the two numbers.

We can now implement our read and write memory functions.

```
// Read a byte from MMU
fn rb(addr : u16, mmu : &Mmu) -> u8 {
    let addr = addr as usize;
    match addr {
        0x0000...0x00FF => if mmu.bios_enabled {mmu.bios[addr]}
        else {
            mmu.rom[addr]
        },
        0x0100...0x3FFF => mmu.rom[addr],
        0x4000...0x7FFF => mmu.srom[addr - 0x4000],
        0x8000...0x9FFF => mmu.vram[addr - 0x8000],
        0xA000...0xBFFF => mmu.eram[addr - 0xA000],
        0xC000...0xCFFF => mmu.wram[addr - 0xC000],
        0xD000...0xDFFF => mmu.swram[addr - 0xD000],
        0xE000...0xEFFF => mmu.wram[addr - 0xE000],
        0xF000...0xFDFF => mmu.swram[addr - 0xF000],
        0xFE00...0xFE9F => mmu.oam[addr - 0xFE00],
        0xFF80...0xFFFE => mmu.hram[addr - 0xFF80],
        // Otherwise, return 0
        _ => 0,
    }
}

// Write a byte to the MMU at address addr
fn wb(addr : u16, value : u8, mmu : &mut Mmu) {
    let addr = addr as usize;
    match addr {
        0x0000...0x7FFF => return, // ROM is Read Only
        0x8000...0x9FFF => mmu.vram[addr - 0x8000] = value,
        0xA000...0xBFFF => mmu.eram[addr - 0xA000] = value,
        0xC000...0xCFFF => mmu.wram[addr - 0xC000] = value,
        0xD000...0xDFFF => mmu.swram[addr - 0xD000] = value,
        0xE000...0xEFFF => mmu.wram[addr - 0xE000] = value,
        0xF000...0xFDFF => mmu.swram[addr - 0xF000] = value,
        0xFE00...0xFE9F => mmu.oam[addr - 0xFE00] = value,
        0xFF80...0xFFFE => mmu.hram[addr - 0xFF80] = value,
        // Otherwise, do nothing
        _ => return,
    }
}
```

Notice that if we want to read 16 bit, we can just call twice the `rb` method and glue the values with the following binary operation:

```
// Combine two input bytes h and l into a 16bit integer containing h:l
fn w_combine(h : u8, l : u8) -> u16 {
    (h as u16) << 8 | (l as u16)
}

// Read a word (2 bytes) from MMU at address addr
pub fn rw(addr : u16, mmu : &Mmu) -> u16 {
    let l = rb(addr, mmu);
    let h = rb(addr + 1, mmu);
    w_combine(h, l)
}
```

Symmetrically, for writing a 16bits register, we can first break it in two 8bits values:

```
// Break the higher and lower part of the input 16bit integer into h:l
fn w_uncombine(hl : u16) -> (u8, u8) {
    ((hl >> 8) as u8, hl as u8)
}
```

and then call `wb` on each.

```
// Write a word (2 bytes) into the MMU at adress addr
fn ww(addr : u16, value : u16, mmu : &mut Mmu) {
    let (h, l) = w_uncombine(value);
    wb(addr, l, mmu);
    wb(addr + 1, h, mmu);
}
```

Loading the rom

Rom files are backup of the data stored in the Nintendo Game Boy's cartridge. At this time, memory was very expensive. It was both difficult to have huge amount of memory and memory chips was costly. Therefor, cartridge has diferent sizes. The smallest ones, like tetris, had a single "memory bank". As a start, we will only load the lightweigh games like Tetris which are usualy stored in a file with extension `.gb`.

The following code open a `.gb` file, check it's size and if it seams compatible, load it.

```
// Load a .gb file into the Mmu struct
fn mmu_from_rom_file(filename : String) -> Result<Mmu> {
    let mut file = try!(File::open(filename));

    let mut contents : Vec<u8> = Vec::new();

    let number_of_bytes = try!(file.read_to_end(&mut contents));

    match number_of_bytes {
        0x8000 => {
            let mmu = Mmu {
                rom : contents[0x0000..0x4000].to_vec(),
                srom : contents[0x4000..0x8000].to_vec(),
                .. Default::default()
            };
            return Ok(mmu);
        }
        _ => return Err(Error::new(ErrorKind::Other, "Wrong file size"))
    }
}
```

Here, we need to explain what is this `try!` macro and the `Ok`/`Err` constructors. The `File::open` method return a `Result<File>`. A `Result<T>`, where `T` is any arbitrary type, is either an `Ok(value : T)` or a `Err(error : Error)`. Depending on the success or failure of `File::open`, we deal with the first or second one. The `try!` macro is then only a test on wich value was returned, which extract the value from `Ok`, or return from the function by returning the error. This abstract construction for handling error is also known as the Either monad (because we have either a result or an error).

Notice that the `.gb` file also cointain divers informations about the cartridge format, the title of the game, or even the manufacturer.

Here is a small code sample for extracting the name and manufacturer of the game in trust.

```
let title = read_string(&mmu.rom[0x0134..], 0x0F);
let manufacturer = read_string(&mmu.rom[0x013F..], 0x0F);

// Construct a string from a sequence of bytes
pub fn read_string(memory : &[u8], max_len : usize) -> String {
    let mut string = String::new();
    let mut idx = 0;
    while (idx < max_len) && memory[idx] != 0 {
        string.push(memory[idx] as char);
        idx += 1;
    }
    return string
}
```

The title of the game is located at address `0x0134` and is at most 15 bytes long. Unused bytes are filled with 0. The manufacturer is located at address `0x013F`, with the same buffer length.

In a language like `C` or `C++`, one could build strings directly by casting a pointer to the byte area into a `char*`, and using `strcpy` to create a null terminqted `C` string. Here we build a copy in a `String` than can be freely manipulated.

Chapter 3

The CPU core

Now, you can load a game into memory, but once the gamme is in the memory, nothing happen. You need to emulate the heart of the Game Boy: the GPU. This section is devoted to structuring and writing the CPU core, the part of the emulater responsible for fetching instructions, and reproducing the behavior of a single instructions on our memory.

The CPU core is, without a doubt, the place the more <> to bugs. CPU cores are hard to debbug and often made of a big bunch of code (easily more than 1 000 lines). If you don't feel confortable with developping one, you can allow yourself to reuse an existing one from a known emulator. If you want to write yours – because it's an <> experience, and you'll probably learn a lot – allow yourself to look at other implementations. Also, remember you don't need a fully functionnal CPU core to emulate a specific game. For example, tetris require only a restricted set of instructions, and their execution don't have to be perfectly faithfull.

States and Registers

The execution of a program is basically repeating three actions : fetching instructions from memory into the CPU core, decoding the instructions – associating to a binary number a block of code – and finaly runing the associated block of code.

Those three operations, called the instruction cycle, are repeated indefinitely until the `STOP` instruction is executed, or the power is cut.

Between two instruction cycles, what change is the internal states of the CPU. The CPU have a small set of variables, called registers, that are modified by

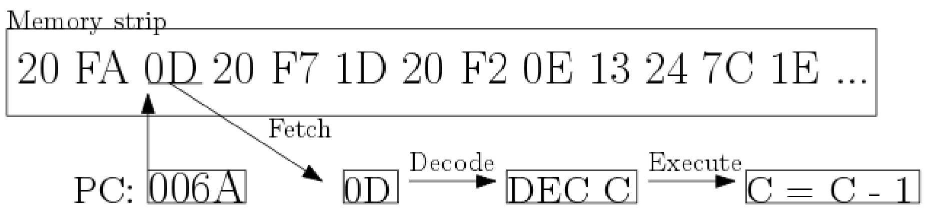

Figure 3.1: The fetch-decode-execute cycle

executing instructions. Some registers are directly accessibles, and others are modified thanks to specific instructions.

Our first step in writing a CPU core is then defining the structure that is going to contain all this states. We need the registers bits register a, b, c, d, e, h, l and f, the stack pointer and the program counter.

```
#[derive(PartialEq, Eq, Default, Debug)]
struct Cpu {
    // CPU's registers
    a : u8,
    b : u8,
    c : u8,
    d : u8,
    e : u8,
    h : u8,
    l : u8,
    f : u8,

    // Program counter
    pc : u16,
    // Stack pointer
    sp : u16,
}
```

Since instruction will have to acess both memory and the CPU, it is easier to give them both through a structure repraisenting a *virtual machine*.

```
#[derive(PartialEq, Eq, Default, Debug)]
struct Vm {
    cpu : Cpu,
    mmu : Mmu,
}
```

Design of the CPU core

The center of the CPU core is the dispatch loop. This loop fetch the next instruction from the memory located at the address stored by the **programme counter**, look for the associated block of code to execute, and then count the amount of time enlapsed.

First, we need two functions for reading byte at the address pointed by the **programme counter**, and incrementing the counter.

```
/// Read a byte from the memory pointed by PC, and increment PC
fn read_program_byte(vm : &mut Vm) -> u8 {
    let byte = mmu::rb(vm.cpu.pc, &mut vm.mmu);
    vm.cpu.pc += 1;
    return byte;
}

/// Read a word (2bytes) from the memory pointed by PC, and increment PC
fn read_program_word(vm : &mut Vm) -> u16 {
    let word = mmu::rw(vm.cpu.pc, &mut vm.mmu);
    vm.cpu.pc += 2;
    return word;
}
```

Notice that in rust, the `+ 1` operator can create an integer overflow. An integer overflow is what happen when the variable already contains the maximum number it can, and you want to add some positive value to it. If you compile in debug mode, a such overflow will cause a crash of your application with an error message. But if you compile in release mode, the overflow occuring in the previous lines will just reset the variable back to `0`, which is the behavior expected from the processor.

Now, we can write the dispatch loop:

```
// The dispatch loop
while true {
    execute_one_instruction(vm);
}

// Execute exactly one instruction by the CPU
//
// The function load the byte pointed by PC, increment PC,
// and call dispatch with the opcode to run the instruction.
// It returns the time this instruction would have taken on
// the real hardware.
fn execute_one_instruction(vm : &mut Vm) -> Clock {
    // Disable bios once executed
    if vm.cpu.pc >= 0x100 {
        vm.mmu.bios_enabled = false;
    }

    // Fetch the instruction
    let opcode = read_program_byte(vm);

    // Run opcode
    let clock = dispatch(opcode)(vm);

    return clock;
}
```

The `dispatch` function is actually particuliar. It get an opcode – a binary number describing the instruction to execute – and return a function, which is the code block we have to run in oder to produce the expected behavior.

Executing instructions

CPU core, because of their design, are in a certain sence inherently slower than the original ardware. Emulating the behavior of a processor means teller your own CPU to execute a certain set of instruction that aim at reproducing what **one** instruction of the targeted hardware can do. If Game Boy's emulater can run actually faster than original harware is only because our processor are now a lot faster. Where the original Game Boy CPU was doing near one million instructions per second, today's computer do more than two billion instructions per seconds.

Even if today's computer are fast, try to keep emulated functions simple, as simplicity is more likely to induce a short nuber of instructions executed by your host CPU. An implementation of an instruction will look at the values given as argument, change the internal states of the vm (the cpu's states and eventually the memory) and return a description of the time this instruction should have take if executed by the Game Boy's CPU. To describe the time, we introduce the structure `Clock`. We count the time in a unit which is a *clock cycle*. In the Game Boy their is actually a small quartz crystal oscillating at a frequency of 4.194304 MHz called the clock. One of this oscillation is then called a *clock cycle*. Notice that since an opcode can have an operand following it in memory, like an number for an add instruction, an instruction (opcode and it's operands) can actually be more than one byte. Though their is not direct use of this knowledge, we also track this value for debuging.

```
struct Clock {
    /// Length in byte of the last instruction
    m : u64,
    /// Duration in clock cycles
    t : u64,
}
```

Most of the instructions affect the *flag register* `F`. This register contains different bytes activated or unactivated depending on the result of the last instruction. Each of such bites are called flags. They are used in conditionnal jump to know if a value is greather than the other, if two values are equals, and so on. Here is a table reasuming the different flags.

Table 3.1: Flag register `F`

Bit 7 : Z	Bit 6 : N	Bit 5 : H	Bit 4 : C
Zero bit	Negative bit	Half Carry bit	Carry bit
Set to 0 if the result is null.	Set if the operation use negative numbers.	Set if a carry appear in the middle of the number.	Set if a carry overflow the register.

The description aren't precise and exact, but they summarise the meaning of these flags. You can find the details of how each operation affect the flag register in annexe .

We define a list of this flags:

```
#[derive(PartialEq, Eq, Copy, Clone, Debug)]
/// List of flags
pub enum Flag {
    Z = 7,
    N = 6,
    H = 5,
    C = 4,
}
```

We allow easy modification of the `F` register through this functions:

```
// Set the specified flag to the value given
fn set_flag(vm : &mut Vm, flag : Flag, value : bool) {
    if value {
        vm.cpu.f |= 1 << flag as usize
    }
    else {
        vm.cpu.f &= !(1 << flag as usize)
    }
}

// Get the state of a flag
```

```
fn get_flag(vm : &Vm, flag : Flag) -> bool {
    0x01 & vm.cpu.f >> (flag as usize) == 0x01
}

// Reset the flags of the Vm (set all flags to 0)
fn reset_flags(vm: &mut Vm) {
    vm.cpu.f = 0
}
```

Lets look at few example of simulated instructions. The first example is the `ADD A, r` operation.

```
// Implement adding a register reg to the register A,
// and update the flag register F.
fn i_add_r(reg : u8, vm : &mut Vm) -> Clock {
    // Aliases for shorter expressions
    let a = vm.cpu.a;
    let b = reg;

    // Compute the sum
    let sum = vm.cpu.a + b;

    set_flag(vm, Flag::Z, sum == 0);
    set_flag(vm, Flag::N, false);
    set_flag(vm, Flag::H, (0x0F & a) + (0x0F & b) > 0xF);
    set_flag(vm, Flag::C, (b as u16) + (a as u16) > 0xFF);

    // Store the result in the register A
    vm.cpu.a = sum;

    Clock { m:1, t:4 }
}
```

Notice that settings the flags can be tricky, especially the half carry flag `H`.

The second is the `JP addr` operation, which jump to the address stored in the next two bytes of memory.

```
// Read the next two bytes and jump to the address
fn i_jp(vm : &mut Vm) -> Clock {
    vm.cpu.pc = read_program_word(vm.mmu);
    Clock { m:3, t:16 }
}
```

As you can see, the previous functions are generic in the sence that they need to be specialised for specific registers. For example, we would like to implement both `ADD A, B` and `ADD A, C` as a specialisation of the previous functions.

Dispatching instructions

Once we have implemented the behavior for one instruction, we have to associate at each opcode the corresponding block of code. This work is doone by the `dispatch` function.

Usually, in `C`, we would use an array indexed by the different value that can take a byte (0 to 255) which alement are pointer on fuction (namely the adresses of block of code). Using the keyword `case` on a value of a byte in C would be converted to an array of function pointer automatically by the compiler, allowing to have a really fast condition branching (instead of comparing the value of the byteode through 256 different if block, we just read the function to execute as a value in an array which is way faster). In `rust`, we can use the `match` keyword and expect a similar optimisation.

instead of directly calling the right block of code, we actually return this block of instruction. This allow more flexibility in case you would like to implement a disassembler, or a debugguer in your emulator. Since writing the CPU core is the part of emulator more propice to expect bugs, adding a real-time debugguer – a way to see which instruction are executed and what are the current state of the CPU while running – to your implementation is a good way to help you discover bugs in your code.

The return type of the dispatch function is a bit particular, it is a `Box<Fn(&mut Vm) -> Clock>`. This type allow us to take any function or closure that take a mutable reference on a `Vm` and produce a `Clock`.

A closure is an anonymous function that can take references or copies of local variables in it's body. The variables, called free variables, are said to ben *closed over* when used in the body of such an anonymous function, hence the name closure. In our specific case closure are just normal anonymous function. Here is an example of closure:

```
let plus_one = |x : int| {
    // Closure body
    return x + 1;
}
```

We know have a look at the dispatch function.

```
fn dispatch(opcode : u8) -> Box<Fn(&mut Vm) -> Clock> {
    match opcode {
        // NOP
        0x00 => Box::new(|$vm : &mut Vm| i_nop(vm)),

        // ...

        // ADD A, r
        0x80 => Box::new(|$vm : &mut Vm| i_addr(vm.cpu.b, vm)],
        0x81 => Box::new(|$vm : &mut Vm| i_addr(vm.cpu.c, vm)],
        0x82 => Box::new(|$vm : &mut Vm| i_addr(vm.cpu.d, vm)],

        // ...

        // JUMP
        0xC3 => Box::new(|$vm : &mut Vm| i_jp(vm)],

        // ...
    }
}
```

We store our closure in a "box", a heap allocated object to be able to return it like if it was just a function.