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 usualy not recomanded to learn a new language while writing an emulator, it is indee what many peolples actually does. If you want to learn Rust while building your emulator, it is something completely realisable, and have already been done by beny person. Just ask yourself if you feel confident enougth 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 think that will be explain in this section seams new to you, you might prefer to stick to a language you already know.

Hexadecimal and binary systems

Lets also recall a few things about numbers and numeric systems.

The number twenty three is writen 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 packet 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 everyday, 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, th 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 means ten in decimal, or does it means sixteen written in hexadecimal?

Only the persone who wrote this number can tell you. In order to tell people the number is writen in exadecimal, 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 befor, we add the prefix `0b` before binary number to distinguish them from decimals. For example, `0b00000111` means seven.

What is the bigest number you can write with four digits? It's fiveteen, `0b1111` in binary or `0xF` in hexadecimal. If we look at pcket of heights bits, usually called bytes, we get two sybols 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 usualy write this operation `0xC7 & 0x4A = 0x42`.

There is other binary operations like *or* (it is enougth 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 (juste rotate the bits like if they was writen around a cylinder). A last one, widely used, is the *xor* of two numbers, noted `^`. It stand 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 a 8-bit computer

Before writing the first line of code, lets describe the elementary block of hardward you would find in the console, and how they relate to each others. Surely, this organisation 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 decode this instruction. It means reading the right number of arguments of the instruction from memory. Finaly, it execute the instruction. The instruction can be additionning 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 likelly to find in any computer.

Working registers

The *working registers* are small variables contained inside the CPU, that can acessed without cost. The size of the registers are linked to the architecture of the processor. A 8 bit processor have register 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 controler thanks to instructions like `JUMP`, `JNZ`, `CALL` and `RET` which change it's values for conditionnal 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 retreive 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. Befor quiting the function, the values are retrived from the stack. The stack can also be used for storing any kind of intermediate value when there is not enougth registers.

Status register

Status registers contains values that alow to control the processor (for example enabling or disabling interupts), and values that are affected by the result of the previous instruction (carry, flag that indicate if the result was zero, …). The first type of values are controled through dedicated instructions.

The memory

IO

Interupts

Chapter 2

The memory

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`.

Chapter 3

Loading the rom (Where to put this?)

Chapter 4

The CPU core

Emulating takes time

Registers described and first rust code line

Architecture (call ptr[opcode] or ptr[op], v1 = readbyte, v2 = readyte.)

Chapter 5

MMU

Chapter 6

MBC

Chapter 7

Annex A

CPU's registers # Annex B CPU Opcode table 1 CPU Opcode table 2