Emulation

Takanen Edoardo

March 26, 2025

Abstract

(still to be placed)
images from:

1. https://www.pastraiser.com/cpu/gameboy/gameboyopcodes.html

Contents

1   Introduction

As a kid, I used to play with some Nintendo (copyright symbol) consoles like the Wii or the DS and I've always been keen about the games they make. This passion for videogames grew on me so that I got interested in the making process of them, leading to game development. I never asked myself one question, though, until this year, which is how are these games able to run onto this consoles? and how can people make emulators so that I could play on my personal computer? Thus, I decided to embrace the unknown world of emulation, because I have always been fascinated by it and always took it for granted. Emulation is not well explained on the Internet. Mainly, the results you will find if you search for it are "the program pretends to be the console" or "you will be able to play old titles". I was not satisfied with these responses, I wanted to know more. Thus, my emulation journey started with looking for a full definition of this process. I will try to give my own definition of emulation, so that I can lay a starting point to a general knowledge that will be then deepened during the paper. With this being said, to "make an emulator" means to develop the software that will do exactly what the hardware of the console does, so that when plugging the game data, the program will know how to read and handle it.

Emulation can only happen when the machine in which we run the software is more powerful than the hardware we want to emulate. For example, if our console has 2Kb of memory, for sure we are not able to emulate it on a computer that has 2Kb or less, since we also have to consider that the host computer will have an operating system running (which uses some of the RAM). I chose to make a Game Boy emulator, because while looking for the retro consoles, it seemed the least difficult when talking about the hardware structure complexity, meaning a good way to start tackling this subject.

2 Premises

This emulator project is purely for understanding the concepts and the theory about how a machine like a console (or similarly a computer) is made (and also for fun). There are many better-developed Game Boy emulators online, and making one that could compete with the other popular ones is nowhere near my goals. Therefore, I could not achieve the level of knowledge I want to reach if I just looked at other people's codes, I want to **fully** understand the subject. Obviously though I have to start somewhere, I do not have the skills to reverse-engineer a real Game Boy (although it would be an extremely interesting challenge), for this reason I will only consult theory guides made by many passionate developers and hackers that already did the work of studying the Game Boy from scratch for us. For a better understanding, I used two sources for this project, in order to have a dual perspective on the study.
For anyone who would like to dig into this challenge, the guides are GBDev and GBEDG.

What this paper is not.
This paper is not and was not intended as a guide, I previously attached some real references. This document is a report of my journey throught the development of the emulator, made for understanding the fundamentals of what is around us, from personal computers to smartphones. It could also be a way for readers to get passionate about this topic and an inspiration for them to make their own emulators (or even better, their own consoles!).

3   General Structure

The first thing I want to cover is in what way we want to structure our emulator.

```
1  int main() {
2      // Hardware components definition
3      Memory mem;
4      CPU cpu;
5      // ...
6
7      // Components initialization
8      mem.init();
9      cpu.init();
10     cpu.load_boot();
11     // ...
12
13     while (true) {
14         cpu.exec_op(mem); // Executing an operation
15         // emulate all the other components
16     }
17
18     return 0;
19 }
```

Actually, when we look at the circuit inside the Game Boy, all the components are, on one side, all on their own, they all execute at the same time. The CPU could be executing a simple addition, while the PPU could be rendering graphics onto the screen, all of these things happen simultaneously. This **can** be done with software, but would mean more complexity. Hence we will pick a less complicated path, and decide to execute the components one at a time.

```
1  while (true) {
2      // Returns how many clock cycles the instruction took
3      int cycles = cpu.exec_op();
4
5      // Updating all the other components
6      timers.update(cycles);
7      ppu.update(cycles);
8      // ...
9  }
```

The real hardware is driven by the clock, while my implementation will be driven by how many clock cycles an instruction took.

4 Memory

Before looking at the main components that shape the Game Boy hardware, I would like to focus on how memory is subdivided inside the console.

4.1 Memory mapping

The address bus had 16 bits, meaning there could be 65'536 unique addresses (64Kb).
Since the Game Boy did not have a flash memory, those 64Kb were all the console could access (this includes all the different RAMs, the cartridge data, and the registers made to control various components). Internally inside the Game Boy, there is some logic that specifies what component will be activated based on the requesting address.
These are the regions into which the memory is split, along with a brief description of their use. Notice that some areas are marked as "Prohibited", though Nintendo has not provided an explanation for this.

Start	End	Description
0000	3FFF	16Kb cartridge ROM
4000	7FFF	16Kb cartridge ROM*
8000	9FFF	8Kb VRAM
A000	BFFF	8Kb External RAM
C000	CFFF	4Kb Work RAM
D000	DFFF	4Kb Work RAM
E000	FDFF	Prohibited area
FE00	FE9F	OAM
FEA0	FEFF	Prohibited area
FF00	FF7F	I/O Registers
FF80	FFFE	High RAM
FFFF	FFFF	IE register

*switchable

Table 1: Game Boy's memory mapping

[MOVE TO CARTRIDGE SECTION] Another interesting fact I found out when making this emulator is what is inside a cartridge. It is fascinating to know that some cartridges would not only include the ROM banks with the game code, but they could also supply their own additional SRAM (Static Random-Access Memory), as well as a battery to preserve the game saves. Due to limiting memory sizes, games could also have a Memory Bank Controller (MBC) to change what ROM should be pointed for memory region $4000 - 7FFF$.
Notice that all these additional components are **not** supported on my emulator, since the original Tetris DMG cartridge just had a 32 Kb ROM.

4.2 Choices for this project

For simplicity, I decided not to break down all these areas into different regions of memory in the emulator, but I opted for an easier solution, which is to create an array of 65'536 bytes, since the data bus was 8 bits-long, so every address will have exacly one byte of data.

There is a trade-off in choosing this approach tough. On one hand, it makes things simpler to manage, we can have all the memory in one place and it really helps when debugging, but on the other hand, it is not entirely correct. Each region of memory has different restrictions, cartridge memory should be readonly, some areas might not be fully readable and writable sometimes (we will see an example when implementing the PPU).

By choosing this option, we are making all the memory readable and writable for everyone, so tecnically, the game could edit its own code (and this actually happened when I was emulating Tetris!).

4.3 Implementation

```
1  struct Memory {
2  private:
3      static constexpr u32 MAX_MEM = 64 * 1024;
4      // Array of bytes to emulate all the Gameboy's addresses
5      Byte Data[MAX_MEM] = {};
6  public:
7      void init();
8
9      /**
10      * Functions used for setting and accessing memory as
11      * mem[addr] = value        to set
12      * Byte value = mem[addr]    to access
13      * */
14      Byte operator[](u32 addr) const;
15      Byte& operator[](u32 addr);
16
17      /**
18      * Dumps all the memory in a file,
19      * used for debugging purposes
20      * */
21      void dump(const char* filename);
22  };
```

This is the entire memory structure. Every component we create will have access to this structure in order to read from and write to memory using the two operator[] functions. I have also added a **dump** function that writes all the bytes to a binary file at the moment the function is called, for easier debugging.

5 CPU

The first component we likely want to implement is the Central Processing Unit (CPU). This component is the most important in our circuit, and is the one that coordinates the other components, executes the program we give to it etc. Thus, the first thing I had to implement were all the instructions that the processor could run.

5.1 Architecture and considerations

The Game Boy's CPU is a custom-made by Sharp Corporation (which had a close relationship with Nintendo at that time), it is often referred to as **DMG-CPU** or **Sharp SM83** and runs at around 4.19 MHz. When making the processor a lot of inspiration was taken from the Zilog 80 and the Intel 8080. Personally, I recently had the possibility to work with a real Z80, and over the past few months, I have gained hands-on experience with its architecture. Specifically, when studying the Zilog, I noticed some differences and similarities with the Game Boy's processor.

For example, the Nintendo processor lacks the IX and IY registers, which in the Zilog were used to set a base address that could be offset with the $LD(IX+d), r$ and $LD(IY+d), r$ to save instruction bytes. Instead, the DMG-CPU introduced a brand new load instruction, LDH (load from high memory), which always offsets from address **FF00**, pointing to **High-RAM** and the **I/O registers**.

I think custom-making their own CPU was the perfect choice for Nintendo, as it allowed them to implement changes like these to better suit their needs, save instruction bytes, and increase performace.

CPUs are the main core of every computer and what they do most of the time is execute instructions defined in some memory. In the next section, we will give a brief summary of the different types of instructions. What is most important is to understand that the majority of these operate on internal registers and external memory.

8-bit registers		16-bit pairs	16-bit register	Description
A*	F**	AF	SP	Stack Pointer
B	C	BC	PC	Program Counter
D	E	DE		
H	L	HL		

*Accumulator
**Flags

Table 2: Game Boy's registers

Registers are the fastest memory to access because it is already inside the processor. However their size is very limited, so we cannot have everything in

them. That is why the CPU has a set of instructions for loading data to and from larger memory.

5.2   The op-tables

The list of all instructions is usually arranged in tables, called opcode-tables, based on their identifier byte.
The Game Boy has 2 op-tables which are shown in Figure 1. Instructions with similar behaviors have been marked with the same color. Since a single byte (one op-table) was insufficient to cover all instructions, an additional table was made, which however uses 2-byte instructions, with the first one always being *0xCB* in hexadecimal, covering all bit operations.
By briefly examining the different instructions, you can see that most of them perform the following operations:

1. Loading values into registers

2. Adding and subtracting values between registers

3. Reading from and writing to memory

4. Comparing values and manipulating individual bits in registers

Obviously other instructions also do other kinds of operations but, as I have said above, most of them operate on the CPU's registers.

8-bit opcodes

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NOP 1 4 ----	LD BC,d16 3 12 ----	LD (BC),A 1 8 ----	INC BC 1 8 ----	INC B 1 4 Z0H-	DEC B 1 4 Z1H-	LD B,d8 2 8 ----	RLCA 1 4 000C	LD (a16),SP 3 20 ----	ADD HL,BC 1 8 -0HC	LD A,(BC) 1 8 ----	DEC BC 1 8 ----	INC C 1 4 Z0H-	DEC C 1 4 Z1H-	LD C,d8 2 8 ----	RRCA 1 4 000C
1x	STOP 0 2 4 ----	LD DE,d16 3 12 ----	LD (DE),A 1 8 ----	INC DE 1 8 ----	INC D 1 4 Z0H-	DEC D 1 4 Z1H-	LD D,d8 2 8 ----	RLA 1 4 000C	JR r8 2 12 ----	ADD HL,DE 1 8 -0HC	LD A,(DE) 1 8 ----	DEC DE 1 8 ----	INC E 1 4 Z0H-	DEC E 1 4 Z1H-	LD E,d8 2 8 ----	RRA 1 4 000C
2x	JR NZ,r8 2 12/8 ----	LD HL,d16 3 12 ----	LD (HL+),A 1 8 ----	INC HL 1 8 ----	INC H 1 4 Z0H-	DEC H 1 4 Z1H-	LD H,d8 2 8 ----	DAA 1 4 Z-0C	JR Z,r8 2 12/8 ----	ADD HL,HL 1 8 -0HC	LD A,(HL+) 1 8 ----	DEC HL 1 8 ----	INC L 1 4 Z0H-	DEC L 1 4 Z1H-	LD L,d8 2 8 ----	CPL 1 4 -11-
3x	JR NC,r8 2 12/8 ----	LD SP,d16 3 12 ----	LD (HL-),A 1 8 ----	INC SP 1 8 ----	INC (HL) 1 12 Z0H-	DEC (HL) 1 12 Z1H-	LD (HL),d8 2 12 ----	SCF 1 4 -001	JR C,r8 2 12/8 ----	ADD HL,SP 1 8 -0HC	LD A,(HL-) 1 8 ----	DEC SP 1 8 ----	INC A 1 4 Z0H-	DEC A 1 4 Z1H-	LD A,d8 2 8 ----	CCF 1 4 -00C
4x	LD B,B 1 4 ----	LD B,C 1 4 ----	LD B,D 1 4 ----	LD B,E 1 4 ----	LD B,H 1 4 ----	LD B,L 1 4 ----	LD B,(HL) 1 8 ----	LD B,A 1 4 ----	LD C,B 1 4 ----	LD C,C 1 4 ----	LD C,D 1 4 ----	LD C,E 1 4 ----	LD C,H 1 4 ----	LD C,L 1 4 ----	LD C,(HL) 1 8 ----	LD C,A 1 4 ----
5x	LD D,B 1 4 ----	LD D,C 1 4 ----	LD D,D 1 4 ----	LD D,E 1 4 ----	LD D,H 1 4 ----	LD D,L 1 4 ----	LD D,(HL) 1 8 ----	LD D,A 1 4 ----	LD E,B 1 4 ----	LD E,C 1 4 ----	LD E,D 1 4 ----	LD E,E 1 4 ----	LD E,H 1 4 ----	LD E,L 1 4 ----	LD E,(HL) 1 8 ----	LD E,A 1 4 ----
6x	LD H,B 1 4 ----	LD H,C 1 4 ----	LD H,D 1 4 ----	LD H,E 1 4 ----	LD H,H 1 4 ----	LD H,L 1 4 ----	LD H,(HL) 1 8 ----	LD H,A 1 4 ----	LD L,B 1 4 ----	LD L,C 1 4 ----	LD L,D 1 4 ----	LD L,E 1 4 ----	LD L,H 1 4 ----	LD L,L 1 4 ----	LD L,(HL) 1 8 ----	LD L,A 1 4 ----
7x	LD (HL),B 1 8 ----	LD (HL),C 1 8 ----	LD (HL),D 1 8 ----	LD (HL),E 1 8 ----	LD (HL),H 1 8 ----	LD (HL),L 1 8 ----	HALT 1 4 ----	LD (HL),A 1 8 ----	LD A,B 1 4 ----	LD A,C 1 4 ----	LD A,D 1 4 ----	LD A,E 1 4 ----	LD A,H 1 4 ----	LD A,L 1 4 ----	LD A,(HL) 1 8 ----	LD A,A 1 4 ----
8x	ADD A,B 1 4 Z0HC	ADD A,C 1 4 Z0HC	ADD A,D 1 4 Z0HC	ADD A,E 1 4 Z0HC	ADD A,H 1 4 Z0HC	ADD A,L 1 4 Z0HC	ADD A,(HL) 1 8 Z0HC	ADD A,A 1 4 Z0HC	ADC A,B 1 4 Z0HC	ADC A,C 1 4 Z0HC	ADC A,D 1 4 Z0HC	ADC A,E 1 4 Z0HC	ADC A,H 1 4 Z0HC	ADC A,L 1 4 Z0HC	ADC A,(HL) 1 8 Z0HC	ADC A,A 1 4 Z0HC
9x	SUB B 1 4 Z1HC	SUB C 1 4 Z1HC	SUB D 1 4 Z1HC	SUB E 1 4 Z1HC	SUB H 1 4 Z1HC	SUB L 1 4 Z1HC	SUB (HL) 1 8 Z1HC	SUB A 1 4 Z1HC	SBC A,B 1 4 Z1HC	SBC A,C 1 4 Z1HC	SBC A,D 1 4 Z1HC	SBC A,E 1 4 Z1HC	SBC A,H 1 4 Z1HC	SBC A,L 1 4 Z1HC	SBC A,(HL) 1 8 Z1HC	SBC A,A 1 4 Z1HC
Ax	AND B 1 4 Z010	AND C 1 4 Z010	AND D 1 4 Z010	AND E 1 4 Z010	AND H 1 4 Z010	AND L 1 4 Z010	AND (HL) 1 8 Z010	AND A 1 4 Z010	XOR B 1 4 Z000	XOR C 1 4 Z000	XOR D 1 4 Z000	XOR E 1 4 Z000	XOR H 1 4 Z000	XOR L 1 4 Z000	XOR (HL) 1 8 Z000	XOR A 1 4 Z000
Bx	OR B 1 4 Z000	OR C 1 4 Z000	OR D 1 4 Z000	OR E 1 4 Z000	OR H 1 4 Z000	OR L 1 4 Z000	OR (HL) 1 8 Z000	OR A 1 4 Z000	CP B 1 4 Z1HC	CP C 1 4 Z1HC	CP D 1 4 Z1HC	CP E 1 4 Z1HC	CP H 1 4 Z1HC	CP L 1 4 Z1HC	CP (HL) 1 8 Z1HC	CP A 1 4 Z1HC
Cx	RET NZ 1 20/8 ----	POP BC 1 12 ----	JP NZ,a16 3 16/12 ----	JP a16 3 16 ----	CALL NZ,a16 3 24/12 ----	PUSH BC 1 16 ----	ADD A,d8 2 8 Z0HC	RST 00H 1 16 ----	RET Z 1 20/8 ----	RET 1 16 ----	JP Z,a16 3 16/12 ----	PREFIX CB 1 4 ----	CALL Z,a16 3 24/12 ----	CALL a16 3 24 ----	ADC A,d8 2 8 Z0HC	RST 08H 1 16 ----
Dx	RET NC 1 20/8 ----	POP DE 1 12 ----	JP NC,a16 3 16/12 ----		CALL NC,a16 3 24/12 ----	PUSH DE 1 16 ----	SUB d8 2 8 Z1HC	RST 10H 1 16 ----	RET C 1 20/8 ----	RETI 1 16 ----	JP C,a16 3 16/12 ----		CALL C,a16 3 24/12 ----		SBC A,d8 2 8 Z1HC	RST 18H 1 16 ----
Ex	LDH (a8),A 2 12 ----	POP HL 1 12 ----	LD (C),A 2 8 ----			PUSH HL 1 16 ----	AND d8 2 8 Z010	RST 20H 1 16 ----	ADD SP,r8 2 16 00HC	JP (HL) 1 4 ----	LD (a16),A 3 16 ----				XOR d8 2 8 Z000	RST 28H 1 16 ----
Fx	LDH A,(a8) 2 12 ----	POP AF 1 12 ZNHC	LD A,(C) 2 8 ----	DI 1 4 ----		PUSH AF 1 16 ----	OR d8 2 8 Z000	RST 30H 1 16 ----	LD HL,SP+r8 2 12 00HC	LD SP,HL 1 8 ----	LD A,(a16) 3 16 ----	EI 1 4 ----			CP d8 2 8 Z1HC	RST 38H 1 16 ----

16-bit opcodes, where the first 8 bits are 0xCB

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	RLC B 2 8 Z00C	RLC C 2 8 Z00C	RLC D 2 8 Z00C	RLC E 2 8 Z00C	RLC H 2 8 Z00C	RLC L 2 8 Z00C	RLC (HL) 2 16 Z00C	RLC A 2 8 Z00C	RRC B 2 8 Z00C	RRC C 2 8 Z00C	RRC D 2 8 Z00C	RRC E 2 8 Z00C	RRC H 2 8 Z00C	RRC L 2 8 Z00C	RRC (HL) 2 16 Z00C	RRC A 2 8 Z00C
1x	RL B 2 8 Z00C	RL C 2 8 Z00C	RL D 2 8 Z00C	RL E 2 8 Z00C	RL H 2 8 Z00C	RL L 2 8 Z00C	RL (HL) 2 16 Z00C	RL A 2 8 Z00C	RR B 2 8 Z00C	RR C 2 8 Z00C	RR D 2 8 Z00C	RR E 2 8 Z00C	RR H 2 8 Z00C	RR L 2 8 Z00C	RR (HL) 2 16 Z00C	RR A 2 8 Z00C
2x	SLA B 2 8 Z00C	SLA C 2 8 Z00C	SLA D 2 8 Z00C	SLA E 2 8 Z00C	SLA H 2 8 Z00C	SLA L 2 8 Z00C	SLA (HL) 2 16 Z00C	SLA A 2 8 Z00C	SRA B 2 8 Z00C	SRA C 2 8 Z00C	SRA D 2 8 Z00C	SRA E 2 8 Z00C	SRA H 2 8 Z00C	SRA L 2 8 Z00C	SRA (HL) 2 16 Z00C	SRA A 2 8 Z00C
3x	SWAP B 2 8 Z000	SWAP C 2 8 Z000	SWAP D 2 8 Z000	SWAP E 2 8 Z000	SWAP H 2 8 Z000	SWAP L 2 8 Z000	SWAP (HL) 2 16 Z000	SWAP A 2 8 Z000	SRL B 2 8 Z00C	SRL C 2 8 Z00C	SRL D 2 8 Z00C	SRL E 2 8 Z00C	SRL H 2 8 Z00C	SRL L 2 8 Z00C	SRL (HL) 2 16 Z00C	SRL A 2 8 Z00C
4x	BIT 0,B 2 8 Z01-	BIT 0,C 2 8 Z01-	BIT 0,D 2 8 Z01-	BIT 0,E 2 8 Z01-	BIT 0,H 2 8 Z01-	BIT 0,L 2 8 Z01-	BIT 0,(HL) 2 16 Z01-	BIT 0,A 2 8 Z01-	BIT 1,B 2 8 Z01-	BIT 1,C 2 8 Z01-	BIT 1,D 2 8 Z01-	BIT 1,E 2 8 Z01-	BIT 1,H 2 8 Z01-	BIT 1,L 2 8 Z01-	BIT 1,(HL) 2 16 Z01-	BIT 1,A 2 8 Z01-
5x	BIT 2,B 2 8 Z01-	BIT 2,C 2 8 Z01-	BIT 2,D 2 8 Z01-	BIT 2,E 2 8 Z01-	BIT 2,H 2 8 Z01-	BIT 2,L 2 8 Z01-	BIT 2,(HL) 2 16 Z01-	BIT 2,A 2 8 Z01-	BIT 3,B 2 8 Z01-	BIT 3,C 2 8 Z01-	BIT 3,D 2 8 Z01-	BIT 3,E 2 8 Z01-	BIT 3,H 2 8 Z01-	BIT 3,L 2 8 Z01-	BIT 3,(HL) 2 16 Z01-	BIT 3,A 2 8 Z01-
6x	BIT 4,B 2 8 Z01-	BIT 4,C 2 8 Z01-	BIT 4,D 2 8 Z01-	BIT 4,E 2 8 Z01-	BIT 4,H 2 8 Z01-	BIT 4,L 2 8 Z01-	BIT 4,(HL) 2 16 Z01-	BIT 4,A 2 8 Z01-	BIT 5,B 2 8 Z01-	BIT 5,C 2 8 Z01-	BIT 5,D 2 8 Z01-	BIT 5,E 2 8 Z01-	BIT 5,H 2 8 Z01-	BIT 5,L 2 8 Z01-	BIT 5,(HL) 2 16 Z01-	BIT 5,A 2 8 Z01-
7x	BIT 6,B 2 8 Z01-	BIT 6,C 2 8 Z01-	BIT 6,D 2 8 Z01-	BIT 6,E 2 8 Z01-	BIT 6,H 2 8 Z01-	BIT 6,L 2 8 Z01-	BIT 6,(HL) 2 16 Z01-	BIT 6,A 2 8 Z01-	BIT 7,B 2 8 Z01-	BIT 7,C 2 8 Z01-	BIT 7,D 2 8 Z01-	BIT 7,E 2 8 Z01-	BIT 7,H 2 8 Z01-	BIT 7,L 2 8 Z01-	BIT 7,(HL) 2 16 Z01-	BIT 7,A 2 8 Z01-
8x	RES 0,B 2 8 ----	RES 0,C 2 8 ----	RES 0,D 2 8 ----	RES 0,E 2 8 ----	RES 0,H 2 8 ----	RES 0,L 2 8 ----	RES 0,(HL) 2 16 ----	RES 0,A 2 8 ----	RES 1,B 2 8 ----	RES 1,C 2 8 ----	RES 1,D 2 8 ----	RES 1,E 2 8 ----	RES 1,H 2 8 ----	RES 1,L 2 8 ----	RES 1,(HL) 2 16 ----	RES 1,A 2 8 ----
9x	RES 2,B 2 8 ----	RES 2,C 2 8 ----	RES 2,D 2 8 ----	RES 2,E 2 8 ----	RES 2,H 2 8 ----	RES 2,L 2 8 ----	RES 2,(HL) 2 16 ----	RES 2,A 2 8 ----	RES 3,B 2 8 ----	RES 3,C 2 8 ----	RES 3,D 2 8 ----	RES 3,E 2 8 ----	RES 3,H 2 8 ----	RES 3,L 2 8 ----	RES 3,(HL) 2 16 ----	RES 3,A 2 8 ----
Ax	RES 4,B 2 8 ----	RES 4,C 2 8 ----	RES 4,D 2 8 ----	RES 4,E 2 8 ----	RES 4,H 2 8 ----	RES 4,L 2 8 ----	RES 4,(HL) 2 16 ----	RES 4,A 2 8 ----	RES 5,B 2 8 ----	RES 5,C 2 8 ----	RES 5,D 2 8 ----	RES 5,E 2 8 ----	RES 5,H 2 8 ----	RES 5,L 2 8 ----	RES 5,(HL) 2 16 ----	RES 5,A 2 8 ----
Bx	RES 6,B 2 8 ----	RES 6,C 2 8 ----	RES 6,D 2 8 ----	RES 6,E 2 8 ----	RES 6,H 2 8 ----	RES 6,L 2 8 ----	RES 6,(HL) 2 16 ----	RES 6,A 2 8 ----	RES 7,B 2 8 ----	RES 7,C 2 8 ----	RES 7,D 2 8 ----	RES 7,E 2 8 ----	RES 7,H 2 8 ----	RES 7,L 2 8 ----	RES 7,(HL) 2 16 ----	RES 7,A 2 8 ----
Cx	SET 0,B 2 8 ----	SET 0,C 2 8 ----	SET 0,D 2 8 ----	SET 0,E 2 8 ----	SET 0,H 2 8 ----	SET 0,L 2 8 ----	SET 0,(HL) 2 16 ----	SET 0,A 2 8 ----	SET 1,B 2 8 ----	SET 1,C 2 8 ----	SET 1,D 2 8 ----	SET 1,E 2 8 ----	SET 1,H 2 8 ----	SET 1,L 2 8 ----	SET 1,(HL) 2 16 ----	SET 1,A 2 8 ----
Dx	SET 2,B 2 8 ----	SET 2,C 2 8 ----	SET 2,D 2 8 ----	SET 2,E 2 8 ----	SET 2,H 2 8 ----	SET 2,L 2 8 ----	SET 2,(HL) 2 16 ----	SET 2,A 2 8 ----	SET 3,B 2 8 ----	SET 3,C 2 8 ----	SET 3,D 2 8 ----	SET 3,E 2 8 ----	SET 3,H 2 8 ----	SET 3,L 2 8 ----	SET 3,(HL) 2 16 ----	SET 3,A 2 8 ----
Ex	SET 4,B 2 8 ----	SET 4,C 2 8 ----	SET 4,D 2 8 ----	SET 4,E 2 8 ----	SET 4,H 2 8 ----	SET 4,L 2 8 ----	SET 4,(HL) 2 16 ----	SET 4,A 2 8 ----	SET 5,B 2 8 ----	SET 5,C 2 8 ----	SET 5,D 2 8 ----	SET 5,E 2 8 ----	SET 5,H 2 8 ----	SET 5,L 2 8 ----	SET 5,(HL) 2 16 ----	SET 5,A 2 8 ----
Fx	SET 6,B 2 8 ----	SET 6,C 2 8 ----	SET 6,D 2 8 ----	SET 6,E 2 8 ----	SET 6,H 2 8 ----	SET 6,L 2 8 ----	SET 6,(HL) 2 16 ----	SET 6,A 2 8 ----	SET 7,B 2 8 ----	SET 7,C 2 8 ----	SET 7,D 2 8 ----	SET 7,E 2 8 ----	SET 7,H 2 8 ----	SET 7,L 2 8 ----	SET 7,(HL) 2 16 ----	SET 7,A 2 8 ----

Figure 1: Game Boy's opcode-tables

It is important to say that some operations depend of results coming from previous instructions. These results are saved in the so called *flags*. Each flag would be represented by a single bit, which is set to 1 when active, and all the flags are stored together inside the F register. Later, we will see that for simplicity I chose to use a separate variable for each flag, instead of using a single F variable.

These flags are:

Bit*	Name	Description
7	zf	Zero flag
6	n	Add/sub flag
5	h	Half carry flag
4	cy	Carry flag
3-0	-	Not used

*bit position inside the F register

Table 3: DMG-CPU's flags

1. **Zero flag**
   Set if the result of an operation is 0
   Used for conditional jumps

2. **Add/sub flag**
   1 if the previous operation was an addition, 0 if it was a subtraction
   Used for DAA instructions only

3. **Half carry flag**
   Set when there is a carry between the lower 4 bits of the operands during an arithmetic operation. It indicates that the lower nibble (4 bits) has overflowed.

4. **Carry flag**
   Set when an arithmetic operation causes a carry beyond the most significant bit of a byte (either the first or the second one in 16-bit operations) in addition, or a borrow when subtracting. Also set when a rotate/shift operation has shifted out a 1.

Instructions also can take different amount of clock cycles to execute. (TODO)

5.3 Implementation

Thus, the first task I had to do was to implement every single instruction shown above, so that my virtual CPU would imitate the original Game Boy's processor behavior.

```cpp
struct CPU {
    Byte A, B, C, D, E, H, L;

    Word SP;
    Word PC;

    // flags
    Byte z, n, h, c, IME;

    Byte fetch_byte(u32& cycles, Memory& mem);
    Word fetch_word(u32& cycles, Memory& mem);
    Byte read_byte(Word addr, u32& cycles, Memory& mem);
    void write_byte(Word addr, Byte data, u32& cycles, Memory& mem)
    ;
    void write_word(Word addr, Word data, u32& cycles, Memory& mem)
    ;

    // ... Functions to execute different bit manipulations

    // ... List of all instructions written as
    // static constexpr Byte INS_[INSTRUCTION] = [OP-CODE];

    // ... Functions to handle interrups (we will discuss them
    later)

    // Executes an instruction
    void exec_op(u32&, Memory&);

    // Gets called by the main loop
    u32 execute(Memory& mem);
}
```