Game Boy emulator - Programming Documentation

Adrián Habušta

August 27, 2023

Contents

1 Specification

1.1 Summary

The project is meant to be a mostly-accurate emulator for the original Game Boy console produced by Nintendo.

1.2 Detailed

Game Boy games were distributed as a cartridge with, at the very least, some ROM that contains the code, graphics and audio of the game. The CPU works with this data and passes it to other components to produce output for the user. Our program needs to accept such a ROM as a file, and run it as the Game Boy would. Some games used more complicated memory patterns, such as having bigger ROMs and using banking to map them onto the limited size the CPU could access, and even sometimes having extra RAM. This RAM was sometimes backed with a battery to allow storing game saves.

2 Components

The emulator consists of classes representing each major component and an emulator class to tie them all together.

2.1 CPU

The processor is emulated by creating an instance of the `central_processing_unit` class, which contains registers and a jump table that executes instructions. Every instruction is cycle accurate, but not precisely hardware accurate. For example, some instructions that use 16-bit registers write to one half of the register during one cycle, and to the other during another cycle. This is not emulated, since nothing except the processor can see these internal registers anyway. The cycles in which the memory is accessed are machine cycle accurate, but they might not be t-cycle accurate (more info on machine cycles vs t-cycles) can be found in the Further Reading section.

The CPU uses a helper class to send information to the emulator that a STOP instruction has been encountered.

Interrupts to the CPU are sent by different components using callbacks that are passed to said components by the emulator at creation.

2.2 Memory mapper

The memory is separated into regions based on the Game Boy memory map. Writing/reading from these regions by the CPU initiates an m-cycle, which starts computing one m-cycle for each other component, before returning control to the CPU. This approach allows for fairly accurate emulation of component timings. The mapper doesn't have it's own memory, it only calls getters/setters for specific regions/registers.

2.3 Cartridge memory controller

This component is unfinished. It contains the basic logic to support different types of Cartridge MBCs (Memory Bank Controllers) but only the simplest type (No MBC) is implemented as of yet. This component allows reading from game ROM, and later will also allow sending commands to the MBC (through writing to the ROM), and interacting with any potential RAM in the emulated cartridge.

2.4 PPU

An acronym for the Pixel Processing Unit. This component is something like a GPU. It is not emulated 100% correctly, because the timing logic of the PPU is very complex, and few games need to have it emulated precisely. Every t-cycle the ppu draws one pixel to a framebuffer using a fairly complex method to decide what to draw. This framebuffer is then rendered to the screen during the VBlank period. The different periods of the PPU are emulated using a simple state machine.

3 Rendering

The screen is rendered at a framerate of around 59.7 fps. At the start of a VBlank period, the PPU framebuffer is pushed to an SDL texture and immediately rendered. All of the emulator logic before this is done as fast as possible. Since the screen can't be affected during VBlank, this is a safe way to render. After all component logic is computed at the last cycle of VBlank, the emulator sleeps for time necessary to stay at 59.7 fps. This is done by counting the amount of cycles that have passed since the last wait period, and the time that it occured. If the LCD is turned off by the emulated game, the screen renders a blank texture and never updates until the LCD is turned back on.

4 Closing thoughts

I enjoyed working on this project, but it is not yet fully finished. The entire APU and serial port are left out. Programming the audio processor would probably take a very long time because it is a complicated system. The serial port would probably require doing some inter-process communication which I have never attempted. Since the rest of the project was already complex, I decided to leave these features out, since I thought they wouldn't be necessary for games to function. The serial port is a bit differnt thought, because one specific game randomly refused to work despite my CPU implementation passing all test ROM tests, and I think it's because the serial port isn't implemented.

As well as these, no MBCs (memory bank controllers) are implemented. In real cartridges, these take care of mapping bigger ROMs to the small range offered by the Game Boy. They also take care of any potential RAM. Leaving these out makes many games unplayable, and they would be the first thing I would implement if I ever came back to this project. I didn't do it now because I couldn't think of an elegant way to implement them, so now I only have a skeleton that supports no MBC.

Overall though, I really enjoyed working on this project. My two previous project were an emulator of the 6502 processor. This project felt like a natural progression from this, since it involved emulating a CPU, and integrating it into a full system. The other components were harder to crack, since I never did anything like them before, but I managed to get it all working in the end.

5 Further Reading

Pandocs, a great resource for most Game Boy knowledge.
Gekkio Game Boy docs, more in-depth information, mostly about the CPU
Official Nintendo Game Boy Programming Manual