A Nintendo Game Boy Emulator

Or:

How I Learned to Stop Worrying and Love the Architecture

CSC 350 Final Project Report

April 2019

[Student Information Redacted]

Introduction

For our final project, we chose to design and implement an emulator of the Nintendo Game Boy handheld gaming console. The main goal of this project was to gain an understanding of the architecture of the Game Boy and produce an executable capable of running Game Boy games. We also extended the architecture by adding a two-stage pipeline. This report discusses the architecture of the Game Boy and related details of the emulator itself, use-case examples, and a brief user guide.

Project Scope

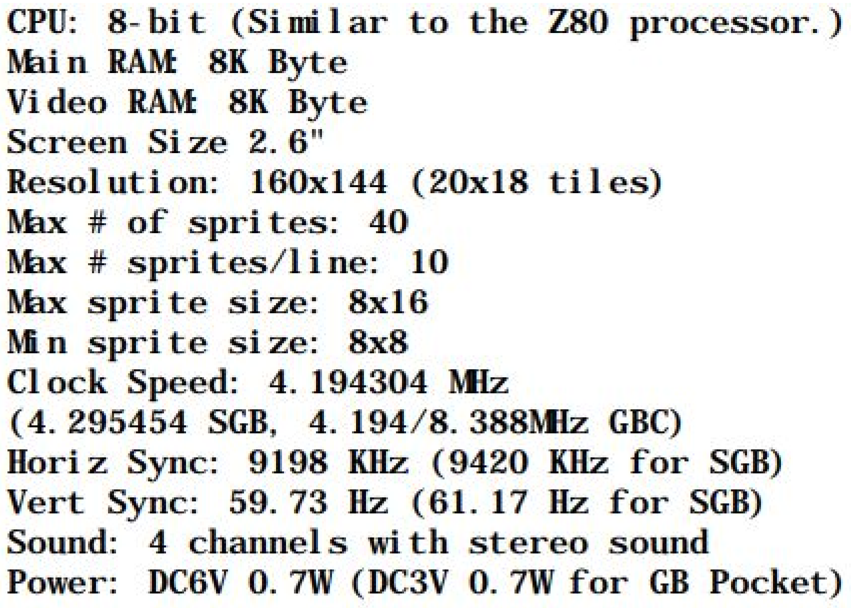
As stated above, the goal of this project was to study and emulate the architecture of the original Nintendo Game Boy. As an additional goal, we were to modify the processor to include a two-stage pipeline. There are limitations, however, in the emulation of a handheld system like the Game Boy, such as peripheral support, data transfer between systems etc. The focus of this emulator is implementing the primary functionality required to play most games. More precisely, we implemented the memory system, instruction set, graphics processing, joypad, and pipeline. We did not implement sound due to the complexity and its nonessential nature.

Brief History of the Game Boy

The Nintendo Game Boy, codenamed DMG (Dot Matrix Game), was the successor to Nintendo's previous handheld system, the Game & Watch. After its release in 1989, the Game Boy outperformed its competition despite its inferior hardware specs [3]. Over the years, the Game Boy line saw several iterations with lighter, more compact form-factors, higher-contrast displays, and colour displays [3]. Since its release, the Game Boy line has sold over 100 million units worldwide [4].

Architecture & Emulator Overview

Although the Game Boy CPU resembles the Intel 8080 and Zilog Z80, the Game Boy's CPU is technically a processor called a Sharp LR25902 [2]. The designers at Nintendo appear to have considered several factors into consideration when designing the Game Boy (e.g. power consumption, price, form factor). As mentioned above, the Game Boy was slightly underpowered compared to its competition, but the trade off between price, battery life, and processing power likely contributed to its success [3]. See Fig. 1 for general hardware specs. Here we present an overview of the Game Boy architecture. The following subsections briefly describe particularly important, interesting, or challenging elements of the architecture as they relate to our emulator.



CPU: 8-bit (Similar to the Z80 processor.)
Main RAM: 8K Byte
Video RAM: 8K Byte
Screen Size 2.6"
Resolution: 160x144 (20x18 tiles)
Max # of sprites: 40
Max # sprites/line: 10
Max sprite size: 8x16
Min sprite size: 8x8
Clock Speed: 4.194304 MHz
(4.295454 SGB, 4.194/8.388MHz GBC)
Horiz Sync: 9198 KHz (9420 KHz for SGB)
Vert Sync: 59.73 Hz (61.17 Hz for SGB)
Sound: 4 channels with stereo sound
Power: DC6V 0.7W (DC3V 0.7W for GB Pocket)

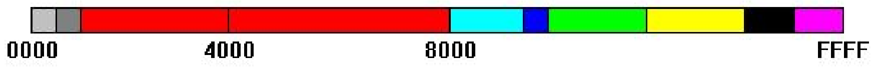
Fig. 1. [1] Game Boy hardware specs

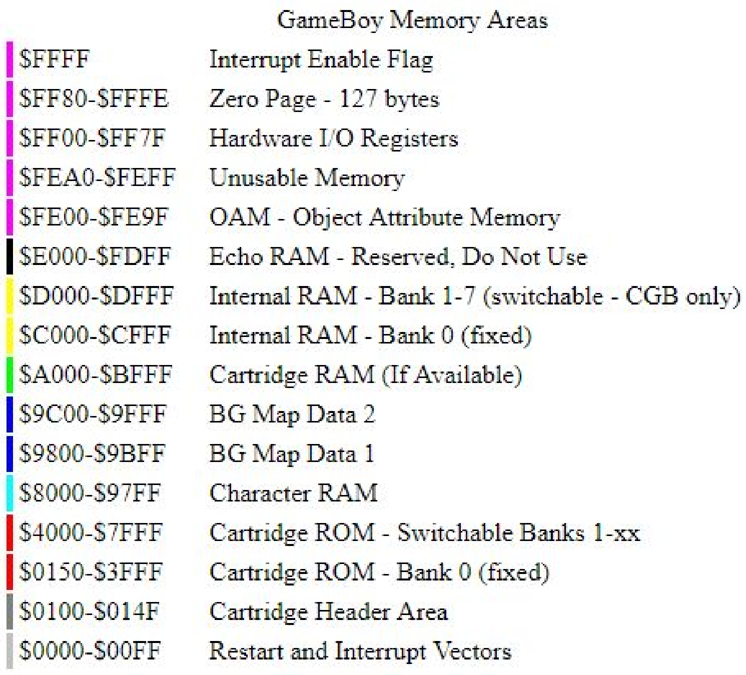
Processor

The processor is an 8-bit CPU based off the 8080 and Z80 as mentioned above [1]. The processor is clocked at 4.194304 MHz. Our implementation of the CPU does the usual business of fetching, decoding, and executing instructions, albeit in an indirect way. The emulator’s handling of instructions timing is elaborated on below.

Memory Map

One integral component of this system is the memory layout. The Game Boy has a 16-bit address space, where system and video RAM; cartridge ROM; game data; and hardware are mapped. See Fig. 2 for a general overview of the memory map.





GameBoy Memory Areas

🟪	$FFFF	Interrupt Enable Flag
🟪	$FF80-$FFFE	Zero Page - 127 bytes
🟪	$FF00-$FF7F	Hardware I/O Registers
🟪	$FEA0-$FEFF	Unusable Memory
🟪	$FE00-$FE9F	OAM - Object Attribute Memory
⬛	$E000-$FDFF	Echo RAM - Reserved, Do Not Use
🟨	$D000-$DFFF	Internal RAM - Bank 1-7 (switchable - CGB only)
🟨	$C000-$CFFF	Internal RAM - Bank 0 (fixed)
🟩	$A000-$BFFF	Cartridge RAM (If Available)
🟦	$9C00-$9FFF	BG Map Data 2
🟦	$9800-$9BFF	BG Map Data 1
🟦	$8000-$97FF	Character RAM
🟥	$4000-$7FFF	Cartridge ROM - Switchable Banks 1-xx
🟥	$0150-$3FFF	Cartridge ROM - Bank 0 (fixed)
◼	$0100-$014F	Cartridge Header Area
◻	$0000-$00FF	Restart and Interrupt Vectors

Fig. 2. [5] Game Boy memory map

The Cartridge Header Area starting at 0100 is where the Game Boy begins execution. There is typically a jump to 0150, the real beginning of the game ROM [5]. The header contains various kinds data about the cartridge including a Nintendo logo, which is loaded onto the screen at startup. If this logo does not match up with that in internal ROM, the Game Boy halts execution [1].

The next area is for cartridge ROM. In what is referred to as ROM bank 0 is 16Kb of fixed memory. After bank 0 is an area of switchable cartridge ROM memory. Game Boy cartridges may contain a *Memory Bank Controller* (MBC) which allow banks of ROM (and/or RAM) stored on the cartridge to be mapped into the memory space and thereby allowing for larger ROMs than with a fixed mapping [5]. There are several types of MBCs, and each type supports different ROM/RAM sizes, so our implementation needed to detect the MBC and emulate its operation.

Addresses 8000 - DFFF and FE00 - FE9F map onto general purpose and video memory. Areas of video memory are defined for *tiles*, described in the following section; background map; and OAM (Object Attribute Memory, i.e. sprite RAM). One curious area is the Echo RAM at address E000 - FE00. It holds a copy of internal (aka work) RAM [1]. The exact purpose of this area aside is not apparent, but most sources do not comment on its purpose or simply state it is not to be used[x, w, v].

The higher address space is used for I/O registers and High RAM (HRAM). The I/O registers include joypad, timer, interrupt flag registers, and other hardware control registers. HRAM allows quicker access than internal RAM, and can be used when faster loads are desired [5].

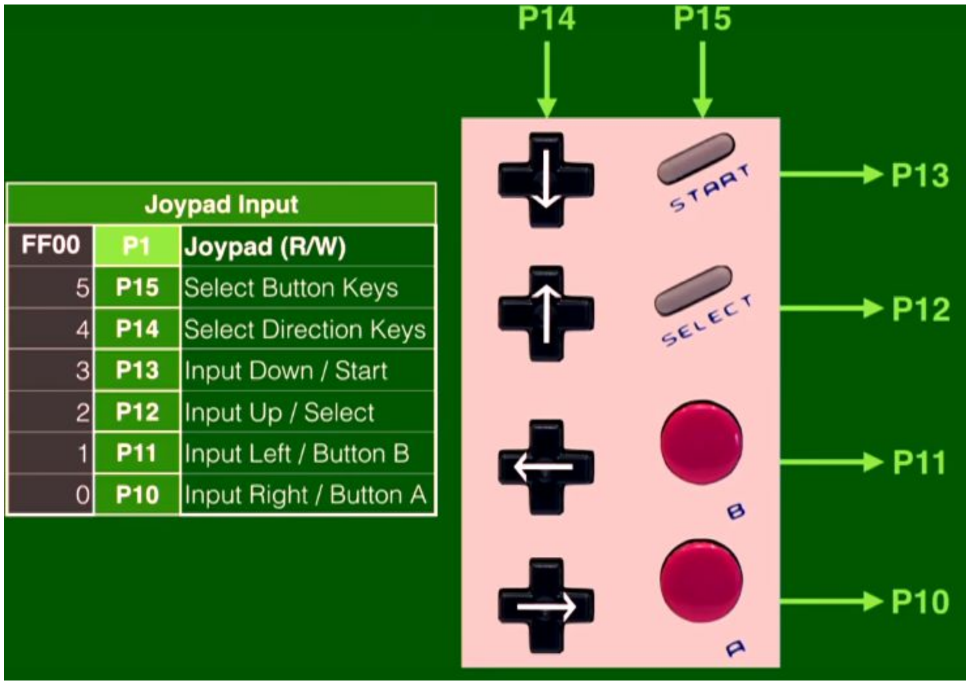
Out emulator essentially treats memory as one large array. The emulator's MMU provides an interface through which reading and writing is controlled. Furthermore, MMU *observers* allow other I/O modules to specify how memory is written and read to their respective areas. This adds another layer of abstraction, allowing easy access to, for example, hardware I/O registers.

Video

The graphical output is powered by an 8-bit Pixel Processing Unit (PPU) clocked at the same 4 MHz as the CPU. The PPU manages it's own memory, with any writes from the CPU needing to pass through the PPU first. The screen supports a 2-bit colour palette, making for "4 bad shades of green" as Michael Steil put it [6].
The video system consists of three main concepts: background, window and sprites. The background is a set of 8 by 8 grids of pixels, tiled across the screen. The area tiles may be assigned to is larger than the actual display, allowing the unit to 'scroll' horizontally and vertically to give the illusion of a moving background. The window is similar to the background, even drawing it's pixel data from the same area of memory, but unlike the background, it is not affected by the PPU's scroll registers, and will remain stationary where positioned. This makes it ideal for small scoreboards along the right or bottom of the display. The final thing to draw is the sprites. These are very self-explanatory -- the blocks in Tetris, Mario, Donkey Kong -- all are sprites. These are the objects in the game that have to move independently of the background. They are drawn on top of the background (with a single exception) and are the only layer to support transparency. There may be up to 40 in use in a game at a time.

Input

The Game Boy joypad consists of a direction pad (up, down, left, right) and four buttons: a, b, start, and select. Instead of representing the joypad with eight bits, one for each button, the Game Boy represents the inputs with 6 bits as a 2x4 matrix. As shown in Fig. 3, the columns select whether the direction pad or buttons are considered. Games write to the joypad register to select the column, wait for a few cycles, and read from the register to get the input. The emulator represents the joypad as a struct which contains separate bytes for the selected column, direction pad, and buttons.Upon reads/writes to the joypad, the appropriate values are written to or returned. The main challenge of input to the emulator is to poll the keyboard and translate that to the joypad register. The emulator takes keyboard input using the SDL library by polling the keyboard in a loop. Keyboard events are passed to the joypad controller to set the appropriate values in the struct.



Joypad Input		
FF00	P1	**Joypad (R/W)**
5	P15	Select Button Keys
4	P14	Select Direction Keys
3	P13	Input Down / Start
2	P12	Input Up / Select
1	P11	Input Left / Button B
0	P10	Input Right / Button A

Fig. 3. [6] Joypad register

Instructions

The Game Boy instruction set is a mix of the instruction set of the Intel 8080 and Zilog Z80, with a few omissions and additions [1]. One interesting feature to note is the dual purpose LD instruction which handles both loads and stores depending on which argument is the memory address. Most of the instructions are as expected: arithmetic, logical, branch, and load/store instructions.The emulator fetches instructions from locations in the Game Boy’s memory, looks up the appropriate opcode from an array, and updates the CPU’s current instruction. The CPU “executes” the instructions by calling functions that implement the instructions.

Examples

The following figures are examples of running Tetris on Linux.

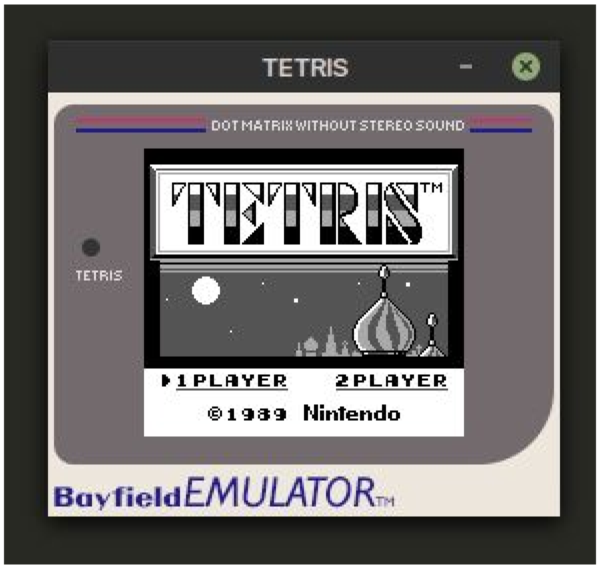


Fig. 4. Tetris start screen

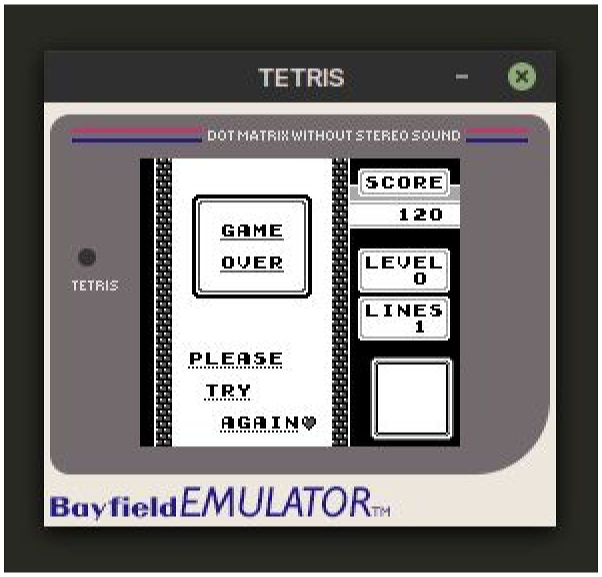


Fig. 5. Tetris game over

Fig. 6 shows the Tetris start screen with the classic, green-gray display of the original Game Boy.

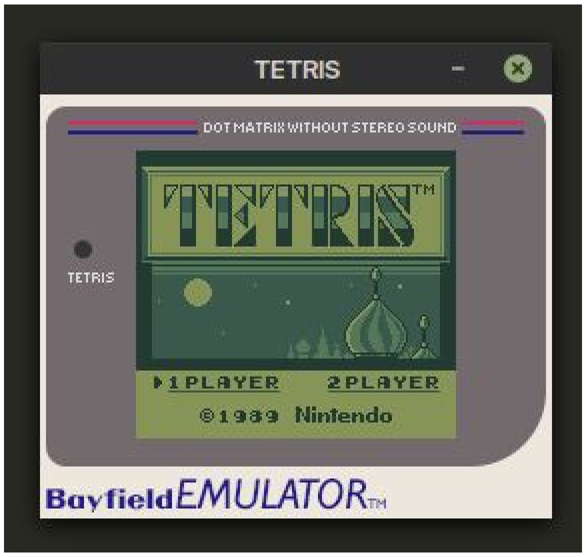


Fig. 6. Tetris start screen on classic display

User Guide

Build Guide

The following is a guide to building the emulator on Mac OS or Linux.

Mac OS

- Install the SDL2 development library from Homebrew: `brew install sdl2`
- Type `make` in the bayfield folder.

Linux

- We recommend using Clang as your compiler.
- Install SDL2 from your distribution's package manager.
- Type `make CC=clang CXX=clang++` in the bayfield folder.

Controls

The Game Boy had a simple interface which we implemented as the following:

- Direction

    - Up - Up arrow key
    - Down - Down arrow key
    - Left - Left arrow key
    - Right - Right Arrow key
- Operation
    - A button - Z key
    - B button - X key
    - Select - Backspace
    - Start - Return

References

[1] DP. *Game Boy™ CPU Manual.* Accessed: Apr. 2, 2019. [Online]. Available: http://marc.rawer.de/Gameboy/Docs/GBCPUman.pdf

[2] “Game Boy Technical Data”. gbdev.gg8.se. http://gbdev.gg8.se/wiki/articles/Game_Boy_Technical_Data (accessed Apr. 4, 2019).

[3] R. Mongenel. “ASMSchool - Introduction”. gameboy.mongenel.com. http://gameboy.mongenel.com/dmg/lessonintro.html (accessed Apr. 4, 2019)

[4] “Dedicated Video Game Sales Units,” Nintendo Co., Ltd., Japan, Dec. 31, 2018. Accessed: Apr. 4, 2019. [Online]. Available: https://www.nintendo.co.jp/ir/en/finance/hard_soft/

[5] R. Mongenel. “GameBoy Memory Map”. gameboy.mongenel.com. http://gameboy.mongenel.com/dmg/asmmemmap.html (accessed Apr. 4, 2019)

[6] Michael Steil. *The Ultimate Game Boy Talk (33c3)*. (Dec. 30, 2016). Accessed: Apr. 4, 2019. [Online Video]. Available: https://www.youtube.com/watch?v=HyzD8pNlpwI