Ensc 351 Group 13 Project Write-up

Noah Kremler	nka71@sfu.ca	301456957
Dan Renardson	drr2@sfu.ca	301463430
Andrew Ye	jya236@sfu.ca	301464553
Shao Chen Zhou	scz2@sfu.ca	301338458

System Explanation

The goal of this project was to deploy an emulator for the Nintendo GameBoy on the BeagleY-AI hardware. Functional requirements included loading GameBoy ROM files from the disk, input control with external joystick and button hardware, and displaying the GameBoy screen to an LCD display via micro-HDMI output, as shown in the system overview diagram below:

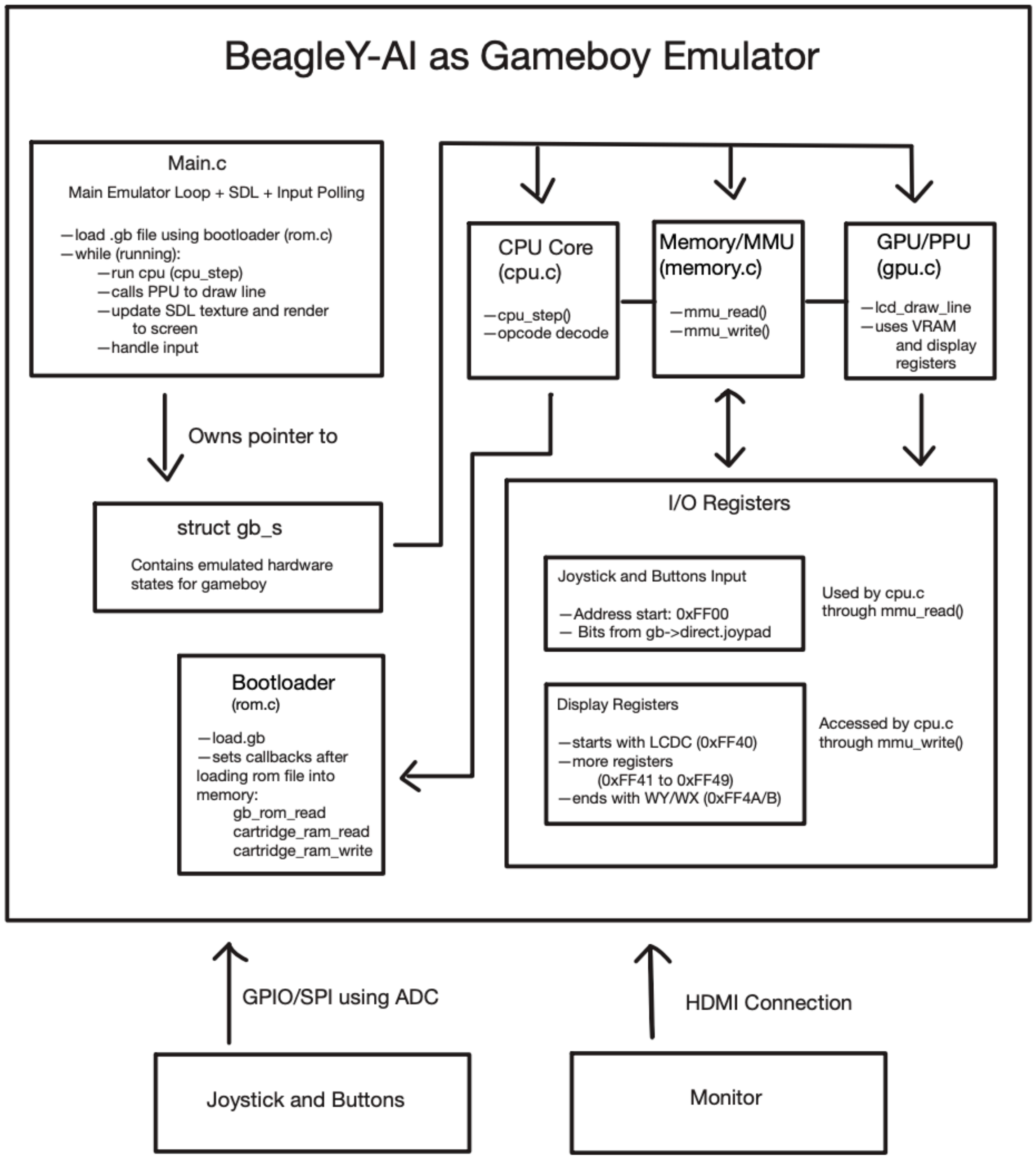
BeagleY-AI as Gameboy Emulator
Main.c
Main Emulator Loop + SDL + Input Polling
—load .gb file using bootloader (rom.c)
—while (running):
—run cpu (cpu_step)
—calls PPU to draw line
—update SDL texture and render to screen
—handle input
Owns pointer to
struct gb_s
Contains emulated hardware states for gameboy
Bootloader
(rom.c)
—load.gb
—sets callbacks after loading rom file into memory:
gb_rom_read
cartridge_ram_read
cartridge_ram_write
CPU Core
(cpu.c)
—cpu_step()
—opcode decode
Memory/MMU
(memory.c)
—mmu_read()
—mmu_write()
GPU/PPU
(gpu.c)
—lcd_draw_line
—uses VRAM and display registers
I/O Registers
Joystick and Buttons Input
—Address start: 0xFF00
— Bits from gb->direct.joypad
Used by cpu.c through mmu_read()
Display Registers
—starts with LCDC (0xFF40)
—more registers (0xFF41 to 0xFF49)
—ends with WY/WX (0xFF4A/B)
Accessed by cpu.c through mmu_write()
GPIO/SPI using ADC
HDMI Connection
Joystick and Buttons
Monitor

Fig. 1: *System overview diagram*

The emulator program is written in C and contains app modules for CPU, GPU, I/O, main emulator loop, ROM and RAM, and timers. There are two HAL modules, one each for the joystick and buttons. The base code for this project was adapted from the *peanut_gb.h* file from the PeanutGB git repository [1]. This open-source gameboy emulator provided us with the declaration for the main emulator context structure that holds arrays for the ROM, RAM, CPU registers, and other configurations and flags. This data structure was heavily modified to suit this project's needs, and code modularization was greatly improved based on the coding practices we learned in the ENSC 351 course.

Program execution is done by passing in an argument of the path to the ROM file in the command line. Initially, the program will call the bootloader function to configure the gameboy context structure; this gives us the authentic initialization and behaviour of the original Game Boy, instead of skipping the sequence altogether like some emulators do. This function loads the ROM file from the path given to the program and puts its data into dynamically allocated memory such that it can be accessed without reading from the disk. Every GameBoy ROM contains an identical Nintendo logo graphic at a defined address range; this is verified to check to ensure the ROM file is intact. The bootloader then checks if the ROM is for the GameBoy Color or GameBoy Advance, both of which are out-of-scope and unsupported by this project. The bootloader also initializes RAM and VRAM before jumping to the game.

Original Gameboy ROMs can have RAM integrated on the cartridge hardware. This RAM and the ROM are both organized into 8 kB and 16 kB banks (pages) that can be switched to occupy the same address spaces [2]. This is controlled by hardware known as the Memory Bank Controller (MBC); there are multiple versions of this chip and this project only supports MBC0 and MBC1. This is by design, as emulating additional MBC chips is outside of the MVP (Minimum Viable Product) scope and does not greatly improve our learning or showcase our understanding of the hardware.

The Sharp LR35902 processor on the real GameBoy uses a pipelined architecture, whereas this project emulates a single cycle architecture; one instruction must finish executing before the next one begins. Some instructions will store values in the VRAM to write to the display. The pixel processing unit (PPU) is responsible for converting tile and sprite data in memory into the final pixels shown on the LCD. It fetches background and sprite tiles from VRAM as it scans across each line of the screen. Instructions keep executing and some will store values in the VRAM to write to the display. VRAM is organized into two pages, one each for the background and the sprites. Each page is organized into 8-pixel square tiles.

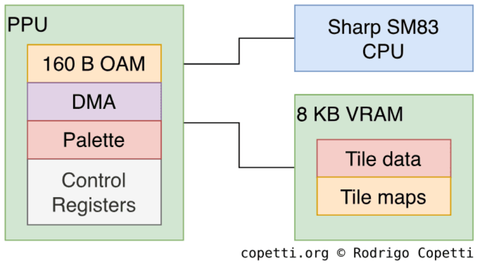

Fig. 2: *Memory architecture of the PPU and VRAM [3]*

After 144 horizontal lines have been written to VRAM, an entire frame is ready for display. SDL3 is an open-source graphics library used to handle display output on the micro-HDMI port on the BeagleY-AI. The tiles from the VRAM are buffered into a 160 by 144 array of RGB pixels. The GameBoy's LCD display has a 2-bit color space and therefore it assigns one of the four colors in the palette (which are shades of gray) to each of the pixels in the array. These pixels are compiled into a format suitable for the SDL3 display function and written to the display device. After a frame is displayed, a 16 millisecond delay is applied such that the program renders at around 60 frames per second.

Inputs are handled by polling the joystick and buttons before the instruction is executed at each clock cycle. This polling will capture all edge events on the buttons that took place over the previous clock cycle and map the host input changes into changes in the I/O register bits. Buttons have hardware RC filtering and software debouncing. The current state of the joystick is read over SPI using the MCP3208 ADC. The joystick inputs are emulating the D-pad on the GameBoy hardware thus there is a threshold at which the joystick is considered to be deflected in any direction. These inputs update their corresponding registers in the GameBoy emulator context data structure.

All of these features come together to produce a playable gaming experience. While instruction timings are not perfect (discussed in the Feature Table below), each component otherwise works as expected. The user runs the executable program on the BeagleY-AI with a supported ROM and the game will appear on the screen connected to the LCD display monitor. The user can play the game using the controller hardware or the keyboard inputs, and can quit the game at any time by pressing the escape key. The entire project runs on the target BeagleY-AI; the host is necessary only for cross-compilation.

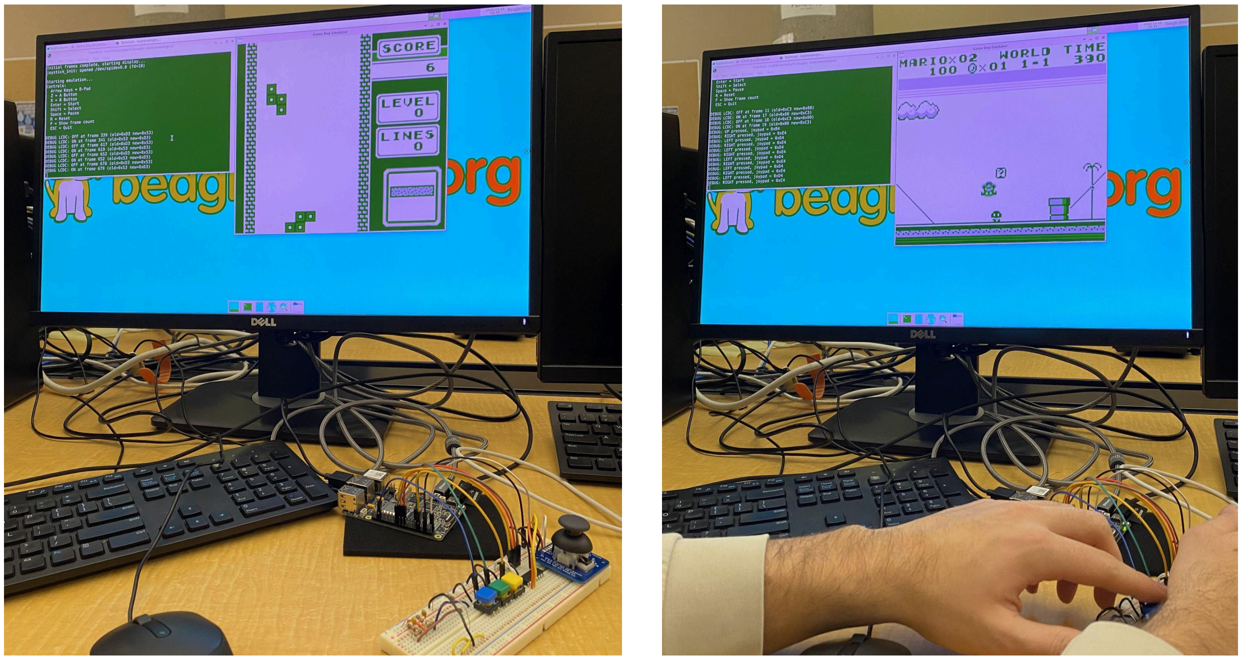

Fig. 3: *Fully functional Tetris (left) and Super Mario World (right)*

Feature table

Description	H/T	Comp	Code	Author	Notes
Emulator Backbone: structure of the overall emulator	T	5	C	Noah	Adapted from PeanutGB [1]
CPU Decoding & Operations: decodes the opcodes and executes the correct behaviors in the emulator context	T	4	C	Noah, Andrew, Chen	CPU behaviour is well-emulated, but instruction timing is slightly off, leading to a slower-than-intended gameplay
Memory Map and Bus Model: handles read and writes to the dedicated memory spaces for the emulator	T	5	C	Noah	Adapted from PeanutGB and assisted by AI
Cartridge and MBC Emulation: Detects cartridge type and chooses proper MBC model	T	5	C	Dan	Cross-referenced Game Boy Sharp LR35902 documentation
Emulator Bootloader: loads the game files and initializes the emulator	T	5	C	Dan	
Pixel Processing Unit: graphical calculations and pixel draws	T	5	C	Andrew	Cross referenced PPU states and behavior with documentation [3], Some sample code taken from PeanutGB [1], ~60% sample code
Video Timing: cycle-level timing accuracy critical for mid-frame effects	T	5	C	Noah	
Graphics Output: displays the pixel buffer	T	5	C	Noah, Andrew, Dan	Uses SDL3 to display the pixels and for frame pacing
Host Integration and UX: Video scaling and Pause, Frame Count, Reset features	T	5	C	Andrew, Noah	Referenced SDL3 documentation
External Hardware Inputs: joystick and buttons for game controls	T	5	C	Chen	Referenced course material
Main Emulator Loop	T	5	C	Noah, Dan	Cross-referenced emulator loop concepts with SDL documentation
Timers	T	5	C	Noah	Adapted from PeanutGB, AI-assisted

Extra Hardware and Software Used

Aside from the standard C libraries and course materials, this project used the SDL3 library to display the rendered pixels, capture user hardware inputs, and for frame pacing. This open source module binds well with the project as it is a common and simple low level solution to both displaying graphics and capturing hardware inputs. It was chosen also due to its high level of compatibility on different devices such as the target device BeagleY-AI.

Also separate from the course material were the three square button switches we used to control the ‘A,’ ‘B,’ and ‘Select’ button inputs necessary for our emulator. We set up a simple breadboard circuit connected to the BeagleY-AI GPIO pins to get these buttons working. This specific hardware was chosen because a group member already had some from a previous electronics project, as well as the buttons’ simplicity.

Fig. 4: *Square button switches [5]*

Project Significance

This project went far beyond the scope of this course in terms of hardware-accurate timing and synchronization, as well as full-stack integration from low-level hardware emulation to a polished user-visible system. It is also impressive because of the depth and difficulty of test-driven validation it required. Matching not only the Game Boy’s functional behaviour but also its timing behaviour proves that this project is a strong example of a soft real-time system, since it must keep up with external time and missing deadlines will degrade user experience, even if it doesn’t cause catastrophic failure.

Throughout the course of this project, our group deepened our understanding of real-time timing and scheduling by learning how to simulate CPU instructions, PPU scanlines, timers, and interrupts from the ~4.19MHz Game Boy clock. We also learned how to synchronize concurrently the aforementioned hardware. In order to deliver predictable performance under variable load, we had to ensure a stable frame rate even when the CPU load changes.

That being said, our emulator's timing is the one piece of our project that lacks polish. If we had more time to work on this project, we would spend our hours debugging the small instruction timing issues in order to get this emulator running at a speed even closer to the original Game Boy. Though we did use test ROMs and manual framerate testing to verify the program speed (we achieved ~40fps, but were aiming for close to 60fps), the next step to improve this project would be to take a deep dive into the CPU code to find performance bottlenecks.

References

[1] deltabeard, "Peanut-GB," GitHub repository (source file: peanut_gb.h), Available: https://github.com/deltabeard/Peanut-GB/blob/master/peanut_gb.h]. [Accessed: December 1, 2025]

[2] Felber, P., Robson, P., and Korth, M., "Game Boy CPU Manual", [Online]. Available: [http://marc.rawer.de/Gameboy/Docs/GBCPUman.pdf], [Accessed: November 30, 2025]

[3] A. Vivace, C. Sandlin, and J. Frohwein, *Pan Docs*. https://gbdev.io/pandocs/. Accessed: Dec. 01, 2025.

[4] R. Copetti, "Game Boy / Color architecture – A practical analysis: Graphics," Copetti.org. Available: https://www.copetti.org/writings/consoles/game-boy/#graphics. Accessed: Dec. 3, 2025.

[5] Colorful 12mm Square Tactile Button Switch Assortment – 15 pack, The Pi Hut. Available: https://thepihut.com/products/colorful-square-tactile-button-switch-assortment-15-pack. [Accessed: Dec. 3, 2025].