Unveiling the Game Boy: Architecture and Web-Based Emulation with TypeScript

1. Introduction to the Game Boy: A Pocket-Sized Powerhouse

The Nintendo Game Boy, first released in 1989 and often referred to by its model number DMG (Dot Matrix Game), was a revolutionary handheld console that captured the hearts of millions worldwide. Its immense popularity was not just a testament to its game library but also to its robust and, in retrospect, relatively straightforward hardware design. This design struck a careful balance between capability, manufacturing cost, and power efficiency, making it a durable and accessible platform for both players and developers. This report aims to demystify the Game Boy's internal architecture and provide a clear guide on how its core functionalities can be emulated in a web-based environment using the TypeScript programming language.
At a high level, the Game Boy, like any computer, is composed of several key interacting components:

- **Central Processing Unit (CPU):** The "brain" that executes game logic.
- **Memory:** Various types of storage for game code, graphics data, working variables, and save files.
- **Picture Processing Unit (PPU):** The specialized hardware responsible for generating the visuals displayed on the screen.
- **Audio Processing Unit (APU):** The hardware that creates all the sounds and music.
- **Input System:** The mechanism for handling player button presses.

Emulating these components involves creating software models that replicate their behavior and their intricate interactions. The Game Boy's architecture, while simpler than modern consoles, presents an excellent learning platform for understanding the fundamentals of hardware emulation. Its limitations, such as the four-shade grayscale display and specific sound channel capabilities, define its unique character, which a faithful emulator strives to preserve.

2. The Heart of the Machine: The Sharp LR35902 CPU

The central processing unit of the Game Boy is a custom 8-bit chip manufactured by Sharp, designated the LR35902. It operates at a clock speed of approximately 4.19 MHz (specifically 4,194,304 Hz, or 2^{22} Hz). This processor was designed exclusively for the Game Boy platform.

The Game Boy's Custom CPU and its Z80/8080 Heritage

The LR35902 is often described as a hybrid, drawing characteristics from two popular microprocessors of its era: the Intel 8080 and the Zilog Z80. The Z80 itself was designed to be binary compatible with the 8080, meaning the 8080's instruction set is largely a subset of the

Z80's. While this lineage provides a useful starting point for understanding the LR35902, it's crucial to recognize that it is not a direct clone of either.
Key distinctions set the LR35902 apart. It omits the Z80's extended index registers (IX and IY). Furthermore, several features from the Intel 8080 instruction set are absent, such as the parity flag and about half of the conditional jump instructions. A significant architectural difference is how it handles Input/Output (I/O) operations. Unlike the 8080 and Z80 which often use dedicated I/O instructions and ports, the LR35902 performs I/O through memory-mapped registers; specific memory addresses are designated for communication with hardware peripherals. The CPU is an 8-bit processor, meaning its internal registers primarily handle data in 8-bit (1-byte) chunks. The design choices leading to these differences, such as removing certain Z80/8080 features and opting for memory-mapped I/O, were likely driven by a desire for cost-saving and simplification, tailoring the CPU specifically for the Game Boy's requirements as a mass-market handheld device where power efficiency and component cost were paramount. This specialization means an emulator must be precise about the features that *are* and *are not* present, rather than attempting to use a generic Z80 core.

Core CPU Functions

The LR35902 performs several fundamental tasks:

- **Instruction Execution:** It fetches instruction codes (opcodes) from memory, decodes them to understand the operation to be performed, and then executes that operation. This is the core of how game logic progresses.
- **Memory Interaction:** The CPU constantly reads data from and writes data to various memory locations. This is its primary method of obtaining instructions, accessing game data, and communicating with other hardware components like the PPU and APU.
- **Interrupt Handling:** The CPU can respond to "interrupts," which are signals generated by other hardware components (like the PPU signaling a vertical blank period, or the joypad signaling a button press). When an interrupt occurs, the CPU temporarily suspends its current task to execute a special interrupt service routine (ISR).

Key CPU Registers

The LR35902 contains a set of internal storage locations called registers, used to hold data temporarily and manage program execution :

- **General Purpose Registers:** There are seven 8-bit general-purpose registers: A (Accumulator), B, C, D, E, H, and L. The Accumulator (A) is often used for arithmetic and logical operations. These registers can be paired to form three 16-bit register pairs: BC, DE, and HL. These pairs are typically used for holding 16-bit data or memory addresses.
- **Program Counter (PC):** A 16-bit register that holds the memory address of the *next* instruction to be fetched and executed. As instructions are fetched, the PC is incremented.
- **Stack Pointer (SP):** A 16-bit register that points to the current top of the stack, an area in Work RAM used for temporary data storage, function call return addresses, and interrupt handling.
- **Flag Register (F):** An 8-bit register where individual bits act as status flags, reflecting the outcome of recent operations. The key flags are:
    - **Zero Flag (Z):** Set if the result of an operation is zero.
    - **Subtract Flag (N):** Set if the last operation was a subtraction.

- **Half Carry Flag (H):** Set if there was a carry from bit 3 to bit 4 during an 8-bit operation (useful for BCD arithmetic).
- **Carry Flag (C):** Set if an operation resulted in a carry out of the most significant bit (for additions) or a borrow (for subtractions). These flags are crucial for implementing conditional jumps and other logic that depends on the results of previous computations.

Emulating the CPU in TypeScript

Emulating the LR35902 in TypeScript involves creating a software model of its registers and its operational cycle.
Representing Registers: A class or interface can define the CPU's registers.

```
class CPURegisters {
    public a: number = 0; // Accumulator
    public f: number = 0; // Flag register

    public b: number = 0;
    public c: number = 0;

    public d: number = 0;
    public e: number = 0;

    public h: number = 0;
    public l: number = 0;

    public sp: number = 0; // Stack Pointer (16-bit)
    public pc: number = 0; // Program Counter (16-bit)

    // Helper to get 16-bit AF register pair
    get af(): number { return (this.a << 8) | this.f; }
    set af(value: number) {
        this.a = (value >> 8) & 0xFF;
        this.f = value & 0xF0; // Lower 4 bits of F are always 0
    }

    // Helper to get/set 16-bit BC register pair
    get bc(): number { return (this.b << 8) | this.c; }
    set bc(value: number) {
        this.b = (value >> 8) & 0xFF;
        this.c = value & 0xFF;
    }

    // Similar getters/setters for DE and HL
    get de(): number { return (this.d << 8) | this.e; }
    set de(value: number) {
        this.d = (value >> 8) & 0xFF;
        this.e = value & 0xFF;
    }
    get hl(): number { return (this.h << 8) | this.l; }
    set hl(value: number) {
        this.h = (value >> 8) & 0xFF;
        this.l = value & 0xFF;
    }

    // Flag manipulation methods
    public setZeroFlag(value: boolean): void {
        this.f = value? (this.f | 0x80) : (this.f & ~0x80);
    }
    public getZeroFlag(): boolean {
        return (this.f & 0x80)!== 0;
    }
    //... similar methods for N, H, C flags
}
```

The Fetch-Decode-Execute Cycle: The CPU operates in a continuous loop:

1. **Fetch:** Read the byte (opcode) from the memory address pointed to by the Program Counter (PC). Increment PC.
2. **Decode:** Identify the instruction corresponding to the fetched opcode. Determine if any additional bytes (operands) need to be fetched.
3. **Execute:** Perform the operations defined by the instruction, which might involve manipulating registers, reading/writing memory, or changing flags. This cycle is fundamental to CPU operation.

Basic Instruction Implementation Approach: A common way to implement the instruction set is using a large switch statement keyed by the opcode, or a Map that maps opcodes to their respective execution functions. Each case or function will contain the logic for a specific Game Boy instruction.

```
// Conceptual example within a CPU class
// Assume 'this.registers' is an instance of CPURegisters
// Assume 'this.mmu' is the Memory Management Unit instance

public step(): number {
    const opcode = this.mmu.readByte(this.registers.pc++);
    let cycles = 4; // Base cycles for fetch, will be adjusted by
instruction

    switch (opcode) {
        case 0x00: // NOP (No Operation)
            cycles = 4;
            break;
        case 0x3E: // LD A, n8 (Load immediate 8-bit value into A)
            const value = this.mmu.readByte(this.registers.pc++);
            this.registers.a = value;
            cycles = 8;
            break;
        //... many more opcodes
        default:
            console.error(`Unknown opcode: 0x${opcode.toString(16)} at
0x${(this.registers.pc - 1).toString(16)}`);
            // Handle error or halt
            break;
    }
    return cycles; // Return T-cycles consumed
}
```

Handling Interrupts: Interrupts allow peripherals to signal the CPU. The Game Boy has an Interrupt Master Enable (IME) flag, which globally enables or disables interrupt handling. Two memory-mapped registers are crucial:

- **Interrupt Enable (IE) Register:** Located at address $FFFF. Each bit corresponds to a specific interrupt source (e.g., VBlank, LCD STAT, Timer, Serial, Joypad). If a bit is set, that interrupt type is enabled.
- **Interrupt Flag (IF) Register:** Located at address $FF0F. When a peripheral requests an interrupt, it sets the corresponding bit in this register.

If the IME flag is true, and a bit is set in *both* the IE and IF registers for the same interrupt source, the CPU will:

1. Reset the IME flag (disabling further interrupts temporarily).
2. Push the current Program Counter (PC) onto the stack.
3. Jump to a predetermined memory address (interrupt vector) specific to that interrupt type (e.g., $0040 for VBlank, $0048 for LCD STAT).
4. The corresponding bit in the IF register must be manually cleared by the interrupt handler routine.

The RETI (Return from Interrupt) instruction is used at the end of an interrupt service routine. It pops the return address from the stack into PC and re-enables the IME flag.
The CPU's execution speed and the timing of its instructions (measured in machine cycles or M-cycles, and T-cycles, where 1 M-cycle = 4 T-cycles) are critical for the overall synchronization of the Game Boy system. While achieving perfect cycle-accuracy is a complex endeavor, understanding that different instructions consume different amounts of time is fundamental. This timing directly impacts how the PPU renders graphics and how the APU produces sound, as these units often rely on the CPU's clock progression for their own operations. Incorrect timing can lead to a wide array of visual and auditory glitches.

3. Memory Architecture: The Game Boy's Brain

The Game Boy's CPU interacts with the rest of the system primarily through its memory interface. Understanding how this memory is organized is key to understanding the console's operation.

Overview of the Memory Map

The Sharp LR35902 CPU features a 16-bit address bus. This allows it to access a total of 2^{16} unique memory locations, which translates to 64 Kilobytes (64KB) of addressable space. However, this 64KB space is not a monolithic block of RAM. Instead, it's a carefully designed **memory map**, where different address ranges are routed to various hardware components,

including ROM, RAM, and I/O device registers. Each component responds to accesses within its designated address range.
The following table summarizes the primary regions of the Game Boy's memory map, largely based on information from :

Address Range	Description	Size	Typical R/W Properties
$0000 - $00FF	Boot ROM (disabled after boot) / Cartridge ROM	256 B	R (Boot ROM), R (Cartridge)
$0100 - $014F	Cartridge Header	80 B	R
$0150 - $3FFF	Cartridge ROM Bank 0	~16 KB	R
$4000 - $7FFF	Cartridge ROM (Switchable Banks)	16 KB	R (Bank-dependent)
$8000 - $9FFF	Video RAM (VRAM)	8 KB	R/W (PPU access restrictions)
$A000 - $BFFF	External RAM (Cartridge SRAM, often battery-backed)	8 KB	R/W (If present)
$C000 - $CFFF	Work RAM (WRAM) Bank 0	4 KB	R/W
$D000 - $DFFF	Work RAM (WRAM) Bank 1-7 (Switchable in CGB)	4 KB	R/W
$E000 - $FDFF	Echo RAM (Mirror of $C000 - $DDFF)	< 8 KB	R/W (Mirrors WRAM)
$FE00 - $FE9F	Object Attribute Memory (OAM) / Sprite RAM	160 B	R/W (PPU access restrictions)
$FEA0 - $FEFF	Unusable Memory	96 B	None
$FF00 - $FF7F	I/O Registers	128 B	R/W (Hardware specific)
$FF80 - $FFFE	High RAM (HRAM)	127 B	R/W
$FFFF - $FFFF	Interrupt Enable (IE) Register	1 B	R/W

Concept of Memory-Mapped I/O (MMIO)

A crucial aspect of the Game Boy's architecture is Memory-Mapped I/O (MMIO). Instead of having separate instructions or a dedicated bus for communicating with hardware peripherals (like the PPU, APU, joypad, or timers), these devices are controlled by reading from and writing to specific memory addresses, primarily within the $FF00 - $FF7F range. When the CPU accesses an address in this range, it's not interacting with standard memory cells but rather with registers on the respective hardware components. For example, writing a value to $FF40 (LCDC register) changes how the PPU operates, and reading from $FF00 (JOYP register) retrieves the current state of the gamepad buttons.

Cartridge ROM and RAM (SRAM)

Game cartridges are the primary medium for game software. They contain:

- **Read-Only Memory (ROM):** This holds the actual game code and fixed data. The Game Boy can directly access 32KB of ROM (from $0000 to $7FFF).
- **Static RAM (SRAM):** Some cartridges include additional RAM, known as SRAM, typically used for saving game progress. This SRAM is often battery-backed to retain data when the Game Boy is powered off. This SRAM is usually mapped into the $A000 - $BFFF address range.

For games larger than 32KB of ROM or those requiring more complex memory management (like switchable SRAM banks), **Memory Bank Controllers (MBCs)** are used. These are special chips on the game cartridge itself that intercept memory accesses and can swap different banks of ROM or RAM into the CPU's addressable range. Common types include MBC1, MBC2, MBC3, and MBC5, each with its own set of control registers and capabilities. Emulating MBCs is an advanced topic but essential for broad game compatibility.

Emulating Memory in TypeScript

To emulate the Game Boy's memory system, a Memory Management Unit (MMU) class is typically created. This class acts as an intermediary for all memory accesses from the CPU and other components.

Representing Memory Regions: Uint8Array typed arrays are ideal for representing the different memory regions due to their efficiency and direct byte manipulation capabilities.

```
class GameBoyMemory {
    public romBank0: Uint8Array; // First 16KB of ROM, or current bank
0
    public switchableRomBank: Uint8Array; // Current switchable ROM
bank (16KB)
    public vram: Uint8Array = new Uint8Array(8 * 1024);         // 8KB
VRAM
    public externalRam: Uint8Array = new Uint8Array(8 * 1024);  // 8KB
Cartridge RAM (SRAM)
    public wramBank0: Uint8Array = new Uint8Array(4 * 1024);    // 4KB
WRAM Bank 0
    public switchableWramBank: Uint8Array = new Uint8Array(4 * 1024);
// 4KB Switchable WRAM (for CGB)
    public oam: Uint8Array = new Uint8Array(160);               // 160
bytes OAM
    public ioRegisters: Uint8Array = new Uint8Array(128);       //
$FF00 - $FF7F
    public hram: Uint8Array = new Uint8Array(127);              //
$FF80 - $FFFE
    public interruptEnableRegister: number = 0;                 //
$FFFF (single byte)

    private fullRomData: Uint8Array | null = null;
    //... MBC related properties (e.g., current ROM/RAM bank, MBC
type)

    constructor() {
        // Initialize with placeholder ROM data or leave for loadRom
method
        this.romBank0 = new Uint8Array(16 * 1024);
        this.switchableRomBank = new Uint8Array(16 * 1024);
    }

    public loadRom(romData: Uint8Array): void {
        this.fullRomData = romData;
        // Initialize romBank0 and switchableRomBank based on ROM data
and MBC type
        // For simplicity, let's assume a 32KB ROM with no banking for
now:
        this.romBank0.set(romData.slice(0, 16 * 1024));
        if (romData.length > 16 * 1024) {
            this.switchableRomBank.set(romData.slice(16 * 1024, 32 *
1024));
        }
        // Parse header to determine MBC type, RAM size etc.
    }
}
```

Implementing a Memory Management Unit (MMU) Class: The MMU class will orchestrate memory access, routing reads and writes to the appropriate physical memory or I/O device. It will hold references to other emulated components (PPU, APU, Timer, Joypad) to delegate MMIO operations.

```
// Forward declarations for PPU, APU, Timer, Joypad classes
// class PPU { /*... */ readRegister(address: number): number;
writeRegister(address: number, value: number): void; /*... */ }
// class APU { /*... */ readRegister(address: number): number;
writeRegister(address: number, value: number): void; /*... */ }
// class Timer { /*... */ readRegister(address: number): number;
writeRegister(address: number, value: number): void; /*... */ }
// class Joypad { /*... */ readRegister(address: number): number;
writeRegister(address: number, value: number): void; /*... */ }


class MMU {
    private memory: GameBoyMemory;
    // References to other components
    private ppu: IPPU; // Using interface defined earlier
    private apu: IAPU;
    private timer: ITimer;
    private joypad: IJoypad;
    //... other components like Serial, MBC handler

    constructor(memory: GameBoyMemory, ppu: IPPU, apu: IAPU, timer:
ITimer, joypad: IJoypad /*... other deps */) {
        this.memory = memory;
        this.ppu = ppu;
        this.apu = apu;
        this.timer = timer;
        this.joypad = joypad;
    }

    public readByte(address: number): number {
        address &= 0xFFFF; // Ensure 16-bit address

        if (address >= 0x0000 && address <= 0x3FFF) { // ROM Bank 0
            // Potentially handle boot ROM overlay here
            return this.memory.romBank0[address];
        }
        if (address >= 0x4000 && address <= 0x7FFF) { // Switchable
ROM Bank
            // Delegate to MBC if present, otherwise return from
current switchable bank
            return this.memory.switchableRomBank[address - 0x4000];
        }
        if (address >= 0x8000 && address <= 0x9FFF) { // VRAM
            return this.ppu.readVRAM(address - 0x8000); // PPU might
restrict access
        }
        if (address >= 0xA000 && address <= 0xBFFF) { // External RAM
(Cartridge)
            // Delegate to MBC if present, otherwise return from
external RAM
            return this.memory.externalRam[address - 0xA000];
        }
        if (address >= 0xC000 && address <= 0xCFFF) { // WRAM Bank 0
            return this.memory.wramBank0[address - 0xC000];
        }
        if (address >= 0xD000 && address <= 0xDFFF) { // Switchable
WRAM Bank
            return this.memory.switchableWramBank;
        }
        if (address >= 0xE000 && address <= 0xFDFF) { // Echo RAM
            return this.readByte(address - 0x2000); // Mirrors
$C000-$DDFF
        }
        if (address >= 0xFE00 && address <= 0xFE9F) { // OAM
            return this.ppu.readOAM(address - 0xFE00); // PPU might
restrict access
        }
        if (address >= 0xFEA0 && address <= 0xFEFF) { // Unusable
memory
            return 0xFF; // Or behavior specific to hardware (often
returns 0xFF)
        }
        if (address >= 0xFF00 && address <= 0xFF7F) { // I/O Registers
            if (address === 0xFF00) return
this.joypad.readRegister(address);
            if (address >= 0xFF04 && address <= 0xFF07) return
this.timer.readRegister(address);
            if (address >= 0xFF10 && address <= 0xFF3F) return
this.apu.readRegister(address);
            if (address >= 0xFF40 && address <= 0xFF4F) return
this.ppu.readRegister(address); // PPU I/O regs
            //... other I/O registers (Serial, etc.)
            return this.memory.ioRegisters[address - 0xFF00]; //
Fallback for unhandled I/O
        }
        if (address >= 0xFF80 && address <= 0xFFFE) { // HRAM
            return this.memory.hram[address - 0xFF80];
        }
        if (address === 0xFFFF) { // Interrupt Enable Register
            return this.memory.interruptEnableRegister;
        }
        return 0xFF; // Default for unmapped reads
    }

    public writeByte(address: number, value: number): void {
        address &= 0xFFFF;
        value &= 0xFF;

        // ROM regions are generally not writable (MBC might handle
writes for bank switching)
        if (address >= 0x0000 && address <= 0x7FFF) {
            // Delegate to MBC for bank switching registers
            // this.mbc?.writeByte(address, value);
            return;
        }
        if (address >= 0x8000 && address <= 0x9FFF) { // VRAM
            this.ppu.writeVRAM(address - 0x8000, value); // PPU might
restrict access
            return;
        }
        if (address >= 0xA000 && address <= 0xBFFF) { // External RAM
            // Delegate to MBC if present
            this.memory.externalRam[address - 0xA000] = value;
            return;
        }
        if (address >= 0xC000 && address <= 0xCFFF) { // WRAM Bank 0
            this.memory.wramBank0[address - 0xC000] = value;
            return;
        }
        if (address >= 0xD000 && address <= 0xDFFF) { // Switchable
WRAM Bank
            this.memory.switchableWramBank = value;
            return;
        }
        if (address >= 0xE000 && address <= 0xFDFF) { // Echo RAM
            this.writeByte(address - 0x2000, value); // Mirrors
$C000-$DDFF
            return;
        }
        if (address >= 0xFE00 && address <= 0xFE9F) { // OAM
            this.ppu.writeOAM(address - 0xFE00, value); // PPU might
restrict access
            return;
        }
        if (address >= 0xFEA0 && address <= 0xFEFF) { // Unusable
memory
            return; // Writes are ignored
        }
        if (address >= 0xFF00 && address <= 0xFF7F) { // I/O Registers
            if (address === 0xFF00) {
this.joypad.writeRegister(address, value); return; }
            if (address >= 0xFF04 && address <= 0xFF07) {
this.timer.writeRegister(address, value); return; }
            if (address >= 0xFF10 && address <= 0xFF3F) {
this.apu.writeRegister(address, value); return; }
            if (address >= 0xFF40 && address <= 0xFF4F) {
this.ppu.writeRegister(address, value); return; } // PPU I/O regs
            //... other I/O registers
            this.memory.ioRegisters[address - 0xFF00] = value; //
Fallback
            return;
        }
        if (address >= 0xFF80 && address <= 0xFFFE) { // HRAM
            this.memory.hram[address - 0xFF80] = value;
            return;
        }
        if (address === 0xFFFF) { // Interrupt Enable Register
            this.memory.interruptEnableRegister = value;
            return;
        }
    }
}
```

Handling Read/Write Specifics: The MMU must respect the unique characteristics of each memory region. ROM is generally read-only, though writes to certain ROM addresses might be intercepted by an MBC to control bank switching. Some I/O registers have bits that are read-only while others are writable, or trigger actions on read/write (e.g., the JOYP register for input). Access to VRAM and OAM can be restricted by the PPU during specific rendering

modes (Mode 2 and Mode 3). The MMU must coordinate with the PPU to enforce these restrictions.
The MMU is more than just a simple address decoder; it's a central hub that orchestrates communication between the CPU and all other hardware components. Its accuracy in routing memory accesses and correctly handling the specific behaviors of I/O registers or memory access restrictions imposed by peripherals like the PPU is fundamental to a stable and correct emulation. A subtle bug in the MMU can manifest as a wide variety of seemingly unrelated problems in any other part of the emulated system. For instance, if the MMU doesn't block CPU writes to OAM when the PPU is in Mode 2 (OAM Scan), a game performing an OAM DMA transfer (which itself blocks CPU access to external RAM) might transfer corrupted sprite data, leading to visual glitches.

4. Bringing Games to Life: The Picture Processing Unit (PPU)

The Picture Processing Unit (PPU) is the specialized hardware component within the Game Boy responsible for generating all the visuals displayed on its 160x144 pixel, monochrome Liquid Crystal Display (LCD). This display is capable of showing four distinct shades of gray. The PPU is a complex piece of hardware, often considered more intricate than the CPU itself.

Key Graphics Concepts

The Game Boy's graphics are built upon several core concepts:

- **Tiles:** The fundamental graphical units are 8x8 pixel tiles. All graphics, including backgrounds, windows, and sprites, are constructed from these tiles.
- **Pixel Data (2bpp):** Each pixel on the Game Boy screen is represented by 2 bits, allowing for one of four colors (shades of gray). A single horizontal row of 8 pixels within a tile is encoded using two bytes of data. The first byte stores the least significant bit (LSB) for each of the 8 pixels' color index, and the second byte stores the most significant bit (MSB) for each pixel. For example, consider two bytes for a pixel row: Byte1 = 0xA5 (binary 10100101) and Byte2 = 0xC3 (binary 11000011).
  - The leftmost pixel (bit 7) gets its LSB from Byte1 (1) and MSB from Byte2 (1). Combined (MSB first), this is binary 11, or color index 3.
  - The next pixel (bit 6) gets LSB from Byte1 (0) and MSB from Byte2 (1). Combined, this is binary 10, or color index 2. This continues for all 8 pixels in the row.
- **Tile Data Storage (Tile Set):** The actual pixel patterns for the tiles are stored in Video RAM (VRAM), specifically in the address range $8000 - $97FF. This area can hold data for up to 384 unique 8x8 tiles (16 bytes per tile).
- **Background (BG):** The Game Boy can display a large background layer composed of a 32x32 grid of tiles, resulting in a total background size of 256x256 pixels. The actual LCD screen shows a 20x18 tile (160x144 pixel) "viewport" into this larger background.
  - **Background Maps (Tile Maps):** The arrangement of tiles on the background grid is defined by one of two selectable background tile maps, also stored in VRAM. These maps are located at $9800 - $9BFF or $9C00 - $9FFF. Each byte in a tile map is an index pointing to a tile in the Tile Data Storage.
  - **SCX/SCY Registers:** The Scroll X ($FF43) and Scroll Y ($FF42) registers control which portion of the 256x256 pixel background is visible in the 160x144 pixel

viewport. These registers allow for smooth scrolling effects.

- **Window:** An optional overlay layer, also constructed from tiles using one of the tile maps, can be displayed on top of the background. Its position is controlled by the WX (Window X position, register $FF4B) and WY (Window Y position, register $FF4A) registers. Notably, the effective WX coordinate for placing the window at the screen's left edge is 7; values less than 7 can cause parts of the window to be drawn off-screen to the left.
- **Sprites (Objects):** The Game Boy can display up to 40 sprites simultaneously on screen, with a hardware limitation of rendering a maximum of 10 sprites per individual scanline. Sprites can be either 8x8 pixels or 8x16 pixels in size.
- **Object Attribute Memory (OAM):** Sprite attributes are stored in a special area of memory called OAM, located from $FE00 to $FE9F. This 160-byte region holds data for 40 sprites, with each sprite using 4 bytes:
    1. Y-position (screen row + 16)
    2. X-position (screen column + 8)
    3. Tile Index (selects a tile from Tile Data Storage)
    4. Attributes (flags for X/Y flip, priority relative to background, and palette selection for CGB).

Scanlines and PPU Modes

The PPU constructs the image on the LCD one horizontal line (scanline) at a time, progressing from the top (scanline 0) to the bottom (scanline 143). The LY register ($FF44) stores the number of the scanline currently being processed by the PPU.
During the drawing of each scanline (LY 0-143), the PPU transitions through several distinct operational modes. Each mode dictates what the PPU is doing and which memory areas (VRAM, OAM) are accessible by the CPU :

- **Mode 2 (OAM Scan):** Duration: approximately 80 PPU clock cycles.
    - At the beginning of each drawable scanline, the PPU enters Mode 2.
    - During this mode, the PPU scans the Object Attribute Memory (OAM) to identify which sprites are visible on the current scanline (LY). A sprite is considered for rendering on the current line if its X-position is greater than 0, LY + 16 is within the sprite's vertical range, and no more than 10 sprites have already been selected for this line.
    - The CPU is **denied access** to OAM during Mode 2.
- **Mode 3 (Drawing Pixels / VRAM Transfer):** Duration: variable, typically 172 to 289 PPU clock cycles.
    - The PPU fetches pixel data from VRAM for the background and window layers, and from VRAM (via OAM tile indices) for the selected sprites.
    - It then combines these pixels based on priority rules and "sends" the final pixel data to the LCD rendering pipeline.
    - The CPU is **denied access** to both VRAM and OAM during Mode 3.
- **Mode 0 (Horizontal Blank / H-Blank):** Duration: variable, typically 87 to 204 PPU clock cycles. The total duration for one scanline (Modes 2 + 3 + 0) is always 456 PPU clock cycles.
    - After drawing all 160 pixels for a scanline, the PPU enters H-Blank. It is effectively idle during this period, waiting for the LCD's horizontal retrace.
    - The CPU has **full access** to VRAM and OAM during Mode 0.
- **Mode 1 (Vertical Blank / V-Blank):** Duration: 4560 PPU clock cycles (10 scanlines, LY

144-153).

  - After scanline 143 is completed, the PPU enters V-Blank mode. The LY register continues to increment from 144 up to 153.
  - During V-Blank, the PPU is not drawing to the screen. This period is critical for games, as it provides a safe time to update VRAM, OAM, palettes, or perform other graphics-related tasks without causing visual artifacts.
  - The CPU has **full access** to VRAM and OAM during V-Blank.
  - A V-Blank interrupt can be generated at the start of Mode 1 (when LY becomes 144), signaling the CPU that it's safe to perform graphics updates.

The PPU's operational timing and its control over VRAM/OAM access are not merely for organizing the rendering process; they are fundamental aspects of the hardware that game developers relied upon. For example, games often perform critical graphics data updates during V-Blank or H-Blank when the CPU is guaranteed access. If an emulator does not correctly simulate these access restrictions, it can lead to a variety of visual errors, from flickering sprites to corrupted backgrounds, or even game crashes if a game attempts to write to VRAM or OAM when the PPU should be blocking such access. This tight coupling between CPU memory access permissions and PPU operational state is a characteristic feature of older console systems and a significant challenge in achieving accurate emulation.

Furthermore, while a deep dive into the PPU's internal "pixel FIFO" (First-In, First-Out) pipeline is beyond a simplified introduction, its existence is hinted at by terms like "Pushing Pixels to the LCD". This internal pipeline is responsible for fetching, mixing, and shifting out pixels sequentially. This explains why Mode 3's duration is variable—it depends on the complexity of the scene (number of sprites, window presence, scrolling) which affects the pipeline's workload. It also underlies how certain advanced graphical effects, like mid-scanline changes to scroll registers (SCX/SCY), can be achieved.

LCD Control Register (LCDC - $FF40)

The primary control over the PPU's operation is exercised through the LCD Control (LCDC) register, located at memory address $FF40. This 8-bit register contains several flags that enable or configure different PPU features:

- **Bit 7: LCD Display Enable:** (0 = Off, 1 = On). Turning this off blanks the screen and halts PPU operation.
- **Bit 6: Window Tile Map Display Select:** (0 = $9800-$9BFF, 1 = $9C00-$9FFF). Selects the VRAM area for the window's tile map.
- **Bit 5: Window Display Enable:** (0 = Off, 1 = On).
- **Bit 4: BG & Window Tile Data Select:** (0 = $8800-$97FF, 1 = $8000-$8FFF). Selects the VRAM area for background and window tile patterns. Method 0 uses signed tile indices.
- **Bit 3: BG Tile Map Display Select:** (0 = $9800-$9BFF, 1 = $9C00-$9FFF). Selects the VRAM area for the background's tile map.
- **Bit 2: OBJ (Sprite) Size:** (0 = 8x8, 1 = 8x16 pixels).
- **Bit 1: OBJ (Sprite) Display Enable:** (0 = Off, 1 = On).
- **Bit 0: BG & Window Display Enable/Priority:** (0 = Off, 1 = On). In DMG, this bit also influences sprite priority over the background.

Emulating the PPU in TypeScript & Rendering to Canvas

Emulating the PPU involves managing its state, simulating its rendering pipeline on a per-scanline basis, and drawing the resulting image to an HTML canvas.

Data Structures:

- The existing vram: Uint8Array and oam: Uint8Array within the MMU/Memory component will be used.
- An internal buffer might be needed to store data for the up to 10 sprites selected for the current scanline during Mode 2.
- A **framebuffer** to hold the pixel data for the entire 160x144 screen. This is typically a Uint8ClampedArray of size 160 * 144 * 4 to store RGBA values for each pixel, compatible with the HTML Canvas API's ImageData object.

PPU State Machine: The PPU emulation will need to track:

- currentMode: The current PPU mode (0, 1, 2, or 3).
- currentScanline (LY): The current scanline being processed (0-153).
- ppuClocksThisScanline: Accumulator for PPU clock cycles within the current scanline.
- LCDC, STAT, SCY, SCX, WY, WX, LYC registers.

A core method, step(cpuCycles: number), will advance the PPU's state. Since the PPU clock runs at the same speed as the CPU clock on the original Game Boy (4.19 MHz), cpuCycles can be directly used as PPU cycles.

Scanline Rendering Logic: The step method will manage transitions between PPU modes based on accumulated cycles:

- **Mode 2 (OAM Scan):** When entering a new scanline (LY 0-143):
    - Switch to Mode 2.
    - Perform OAM scan: Iterate through the 40 OAM entries. For each sprite, check if sprite.x > 0, LY + 16 >= sprite.y, and LY + 16 < sprite.y + (spriteHeight). If so, and fewer than 10 sprites are already selected for this line, add it to a temporary list of sprites for this scanline.
    - Mode 2 lasts for 80 cycles.
- **Mode 3 (Drawing Pixels):** After Mode 2 completes:
    - Switch to Mode 3.
    - Iterate from pixel X = 0 to 159 for the current LY:
        1. **Background Pixel:** If BG is enabled (LCDC Bit 0), determine the tile and pixel data for the current (SCX + X, SCY + LY) coordinate on the 256x256 BG grid. Decode the 2bpp tile data to get a 4-shade color index.
        2. **Window Pixel:** If Window is enabled (LCDC Bit 5) and the current pixel (X, LY) is within the window's active area (defined by WX, WY), determine the tile and pixel data for the window. Decode its 2bpp color.
        3. **Sprite Pixels:** Iterate through the sprites selected in Mode 2. If a sprite is visible at the current X coordinate and has priority, decode its 2bpp tile data. Sprite pixels with color index 0 are transparent.
        4. **Mixing:** Combine the BG, Window, and Sprite pixels based on their color indices, transparency, and priority rules (e.g., LCDC Bit 0 for BG priority, sprite attribute bits for sprite priority).
        5. Convert the final 4-shade color index to an RGBA value using a predefined palette (e.g., white, light gray, dark gray, black).
        6. Write this RGBA value into the correct position in the framebuffer ((LY * 160 + X) * 4).
    - Mode 3 duration is variable (e.g., 172 cycles base + cycles for sprites).
- **Mode 0 (H-Blank):** After Mode 3 completes for a scanline:
    - Switch to Mode 0.

    - PPU is idle. Mode 0 lasts until a total of 456 cycles have elapsed for the current scanline.
    - An H-Blank interrupt can be requested if enabled in STAT.
- **Scanline Increment:** After 456 cycles, increment LY.
    - If LY reaches 144, transition to Mode 1 (V-Blank).
    - Otherwise, loop back to Mode 2 for the new scanline.
- **Mode 1 (V-Blank):** When LY = 144:
    - Switch to Mode 1.
    - Request a V-Blank interrupt (set bit 0 in IF register).
    - This is the ideal time to transfer the contents of the framebuffer to the HTML canvas for display.
    - V-Blank lasts for 10 scanlines (4560 cycles). When LY reaches 153 and 456 cycles pass for that line, LY resets to 0, and the PPU goes back to Mode 2 (or Mode 0 briefly if LCD is just turned on).

Decoding 2bpp Pixel Data: A function is needed to convert the 2-byte-per-tile-row format into usable color indices.

```
// Palette: array of 4 RGBA colors, e.g., [, ,...]
function decodePixelColor(byte1: number, byte2: number,
pixelIndexInRow: number): number {
    // pixelIndexInRow is 0 (leftmost) to 7 (rightmost)
    // Bits are typically read from left (MSB of byte) to right (LSB
of byte)
    const bitPosition = 7 - pixelIndexInRow;
    const lsb = (byte1 >> bitPosition) & 1;
    const msb = (byte2 >> bitPosition) & 1;
    return (msb << 1) | lsb; // Returns color index 0-3
}
```

The Game Boy's classic 4-shade grayscale palette can be represented as: Color 0: (White) Color 1: (Light Gray) Color 2: (Dark Gray) Color 3: (Black)

Rendering to Canvas:

1. In your HTML, include a <canvas> element: <canvas id="gameboyScreen" width="160" height="144"></canvas>.
2. In TypeScript, get the 2D rendering context:
```
const canvas = document.getElementById('gameboyScreen') as
HTMLCanvasElement;
const ctx = canvas.getContext('2d');
let imageData: ImageData | null = null;
if (ctx) {
    imageData = ctx.createImageData(160, 144);
}
```
3. During V-Blank (typically once per emulated frame), after the PPU has filled its internal framebuffer (which is a Uint8ClampedArray):
```
// Assuming 'ppuFrameBuffer' is the Uint8ClampedArray[160*144*4]
from PPU
if (ctx && imageData && ppu.isFrameComplete()) { //
isFrameComplete is a hypothetical PPU method
```

```
        imageData.data.set(ppu.getFrameBuffer()); // Copy pixel data
        ctx.putImageData(imageData, 0, 0);
    }
```

This putImageData call draws the emulated screen to the visible canvas.

Main Rendering Loop Integration: The PPU's step function will be called from the main emulator loop, driven by requestAnimationFrame for smooth browser rendering.

```
// Simplified main loop structure
function gameLoop(timestamp: number) {
    // Calculate cycles to run based on time elapsed since last frame
    //...

    // Run CPU for calculated cycles, which calls MMU, which calls
PPU.step(), APU.step(), Timer.step()
    // emulator.runCycles(numCyclesToRun);

    // PPU.step() internally updates its scanline, mode, and internal
framebuffer.
    // If PPU signals VBlank (frame ready):
    if (ppu.isVBlank()) { // Or a flag set by PPU
         renderToCanvas(); // Function that does the
imageData.data.set and ctx.putImageData
    }

    requestAnimationFrame(gameLoop);
}
requestAnimationFrame(gameLoop); // Start the loop
```

5. The Sound of Nostalgia: The Audio Processing Unit (APU)

The Audio Processing Unit (APU) of the Game Boy is responsible for generating all the distinctive sounds and music that form a core part of its nostalgic appeal. It features four independent sound channels whose outputs are mixed together. The APU's behavior is controlled via a set of memory-mapped registers located from $FF10 to $FF3F.

The Four Sound Channels

1. **Channel 1 (Pulse/Square Wave 1 - NR1x registers):**
   - Generates a rectangular waveform, often called a pulse wave or square wave.
   - Offers four selectable **duty cycles**: 12.5%, 25%, 50% (a perfect square wave), and 75%. The duty cycle defines the ratio of the "high" portion to the "low" portion of one wave cycle, affecting the timbre of the sound.
   - Controlled by registers for **frequency** (pitch), a **volume envelope** (for dynamic volume changes like fade-ins/outs), and a **length counter** (to automatically silence the channel after a set duration).
   - Uniquely features a hardware **frequency sweep** unit (controlled by register NR10).

This allows the channel's frequency to automatically slide up or down over time, useful for sound effects like lasers, sirens, or pitch bends.

2. **Channel 2 (Pulse/Square Wave 2 - NR2x registers):**
    - Functionally very similar to Channel 1, producing a pulse wave with selectable duty cycles, frequency control, a volume envelope, and a length counter.
    - The main difference is that Channel 2 **lacks the frequency sweep** unit.
3. **Channel 3 (Waveform RAM / Custom Wave - NR3x registers):**
    - This channel can play arbitrary waveforms defined by the programmer. The waveform data is stored in a small 16-byte area of RAM called Wave Pattern RAM, located at addresses $FF30 - $FF3F.
    - Each byte in this RAM stores two 4-bit samples, allowing for a total of 32 four-bit samples to define one complete waveform cycle.
    - It has frequency control and a length counter. Volume control is simpler than the pulse channels, offering direct selection of output levels (0%, 25%, 50%, or 100% of the sample value) via register NR32.
4. **Channel 4 (Noise Generator - NR4x registers):**
    - Produces pseudo-random noise, typically used for percussive sounds (like drums or cymbals) and sound effects (like explosions or static).
    - The noise is generated using a **Linear Feedback Shift Register (LFSR)**.
    - It has controls for the LFSR's clock frequency (divisor and shift clock frequency), a volume envelope, and a length counter. The LFSR can operate in 15-bit or 7-bit mode (affecting the character of the noise).

Key APU Mechanisms

Several underlying mechanisms govern the behavior of these channels:

- **Frame Sequencer:** This is a critical timing component within the APU. It is driven by a 512 Hz clock (derived from the system clock). The Frame Sequencer steps through an 8-step sequence. At specific steps in this sequence, it generates clock signals for other APU units :
    - **Length Counters:** Clocked at 256 Hz (steps 0, 2, 4, 6).
    - **Volume Envelopes:** Clocked at 64 Hz (step 7).
    - **Frequency Sweep (Channel 1):** Clocked at 128 Hz (steps 2, 6). The precise timing of the Frame Sequencer is fundamental to the characteristic attack, decay, and duration of Game Boy sounds. If its timing or the way it triggers these sub-units is not emulated correctly, sounds will be flat, have incorrect durations, or lack dynamic effects.
- **Volume Envelopes:** Found on Channels 1, 2, and 4. These allow the volume of a channel to change automatically over time. Key parameters, set via NRx2 registers, include:
    - Initial Volume (4 bits, 0-15).
    - Envelope Direction (0 = Decrease, 1 = Increase).
    - Envelope Period/Step Time (3 bits, 0-7). A period of 0 means no automatic volume change. If non-zero, the volume is adjusted by 1 (up or down) every Period * (1/64) seconds.
- **Length Counters:** Found on all four channels. These can automatically disable a channel after a programmed duration. When a channel is triggered with its length counter enabled (via a bit in NRx4), the counter is loaded with a value (64-L for channels 1, 2, 4 from

NRx1; 256-L for channel 3 from NR31). It then decrements at each 256 Hz clock from the Frame Sequencer. When it reaches zero, the channel is silenced.

- **Timers and Frequency:** Each channel has a frequency timer that dictates its fundamental pitch or playback rate. These timers are typically implemented as down-counters. A frequency value written to the channel's registers (NRx3 and NRx4) is used to calculate a period. The timer counts down from this period. When it reaches zero, it generates an output clock (which advances the channel's waveform generator) and reloads itself with the period. A common hardware pattern is that changes to the frequency registers only take effect when the timer reloads. Emulating this "deferred update" is important for accuracy, especially with rapid frequency changes.
- **Digital-to-Analog Conversion (DAC) and Mixing:** Each of the four channels generates a stream of 4-bit digital sample values (0-15). Conceptually, each channel has its own DAC that converts these digital values into an analog audio signal. These four analog signals are then mixed together. The Game Boy provides stereo output via the headphone jack (controlled by panning registers NR50 and NR51) and a mono output via its internal speaker (where left and right are typically summed).

APU Channel Register Summary

The following table gives a conceptual overview of the key registers controlling the APU channels, based on. x refers to the channel number (1, 2, 3, or 4).

Register	Address	Channel(s)	Key Bits / Function
NR10	$FF10	1	Sweep: Time (bits 6-4), Direction (bit 3), Shift (bits 2-0)
NR11/NR21	$FF11/$FF16	1, 2	Duty Cycle (bits 7-6), Length Load (bits 5-0, value is 64-L)
NR12/NR22	$FF12/$FF17	1, 2	Envelope: Initial Volume (7-4), Direction (3), Period (2-0)
NR13/NR23	$FF13/$FF18	1, 2	Frequency LSB (all 8 bits)
NR14/NR24	$FF14/$FF19	1, 2	Trigger (bit 7), Length Enable (bit 6), Frequency MSB (bits 2-0)
NR30	$FF1A	3	DAC Power (bit 7)
NR31	$FF1B	3	Length Load (all 8 bits, value is 256-L)
NR32	$FF1C	3	Output Level Select (bits 6-5: 00=0%, 01=100%, 10=50%, 11=25%)
NR33	$FF1D	3	Frequency LSB (all 8 bits)
NR34	$FF1E	3	Trigger (bit 7), Length

Register	Address	Channel(s)	Key Bits / Function
			Enable (bit 6), Frequency MSB (bits 2-0)
NR41	$FF20	4	Length Load (bits 5-0, value is 64-L)
NR42	$FF21	4	Envelope: Initial Volume (7-4), Direction (3), Period (2-0)
NR43	$FF22	4	Noise: Clock Shift (bits 7-4), Width Mode (bit 3), Divisor Code (bits 2-0)
NR44	$FF23	4	Trigger (bit 7), Length Enable (bit 6)
NR50	$FF24	Master	Vin Panning (bit 7, 3), Master Volume Right (bits 6-4), Master Volume Left (bits 2-0)
NR51	$FF25	Master	Channel Panning: CH4-CH1 to Right (bits 7-4), CH4-CH1 to Left (bits 3-0)
NR52	$FF26	Master	Sound Enable (bit 7), Channel Status CH4-CH1 (bits 3-0, Read-Only)
$FF30-$FF3F	-	3 (WaveRAM)	16 bytes for 32 x 4-bit samples

Emulating the APU in TypeScript & Web Audio API

Emulating the APU involves modeling each channel's sound generation process and using the Web Audio API to play back the mixed audio.

Setup AudioContext: This is the main interface to the Web Audio API.

```
const audioCtx = new (window.AudioContext |
| (window as any).webkitAudioContext)();
if (!audioCtx) {
    console.warn("Web Audio API is not supported in this browser.");
}

let masterGainNode: GainNode | null = null;
if (audioCtx) {
    masterGainNode = audioCtx.createGain();
    masterGainNode.gain.value = 0.5; // Initial volume
    masterGainNode.connect(audioCtx.destination);
}
```

AudioWorklets for Custom Sound Generation: For accurate emulation of the Game Boy's unique sound characteristics (duty cycles, custom waveforms, LFSR noise, precise timing of envelopes/sweeps), AudioWorkletProcessor is the recommended approach. An AudioWorkletNode runs in a separate audio thread, processing audio in blocks and allowing custom JavaScript/Wasm code for sound generation, which is ideal for low-latency, precise synthesis.
A separate AudioWorkletProcessor would be created for each of the four channels. The main emulator thread would communicate with these worklets (e.g., via port.postMessage()) to update their parameters (frequency, volume, duty cycle, wave RAM contents, etc.) when the game writes to APU registers.

- **Channel 1 & 2 (Pulse Waves) via AudioWorkletProcessor:**
    - The worklet's process method would generate the pulse waveform based on the current frequency, duty cycle (8-step pattern), volume envelope state, and sweep state (for Channel 1).
    - The frequency timer should be a down-counter, with frequency changes applied on reload.
    - Volume envelope and sweep logic will be updated based on "ticks" received from the main thread, corresponding to Frame Sequencer events.
    - The output buffer of the process method is filled with the generated samples.

Example structure for a PulseChannelWorklet.js (to be registered with audioCtx.audioWorklet.addModule):
```
// PulseChannelProcessor.js
(Illustrative)
class PulseChannelProcessor extends AudioWorkletProcessor {
    constructor() {
        super();
        // Initialize frequency, dutyCycle, envelope, sweep,
length counter states
        this.phase = 0;
        this.dutyCyclePattern = ; // Example 12.5%
        this.currentVolume = 15;
        //... other properties

        this.port.onmessage = (event) => {
            // Handle messages from main thread to update
parameters
            // e.g., event.data.type === 'SET_FREQUENCY',
event.data.value
            // e.g., event.data.type === 'FRAME_SEQUENCER_TICK',
event.data.unit ('envelope', 'length', 'sweep')
        };
    }

    static get parameterDescriptors() {
        return [{ name: 'frequency', defaultValue: 440, minValue:
0 }];
        // Actual frequency will be controlled internally based on
GB registers
    }
    process(inputs, outputs, parameters) {
        const output = outputs; // Output buffer for this channel
        const outputChannel = output; // Assuming mono output per
channel initially

        for (let i = 0; i < outputChannel.length; i++) {
            // 1. Advance internal frequency timer based on
sampleRate and GB frequency
            // 2. If timer clocks, advance duty cycle step
            // 3. Get current duty cycle output (0 or 1)
            // 4. Multiply by current envelope volume (0-15)
            // 5. Scale to -1.0 to 1.0 range for Web Audio
            // outputChannel[i] = calculatedSample;
        }
        return true; // Keep processing
    }
}
registerProcessor('pulse-channel-processor',
PulseChannelProcessor);
```

- **Channel 3 (Wave Channel) via AudioWorkletProcessor:**
  - The worklet will need access to the 16 bytes (32 4-bit samples) of Wave RAM. This can be sent from the main thread when the game writes to $FF30-$FF3F.
  - The process method steps through these 32 samples at a rate determined by the channel's frequency setting.
  - Volume shifting (0%, 25%, 50%, 100%) is applied to the 4-bit sample value.
- **Channel 4 (Noise Channel) via AudioWorkletProcessor:**
  - The worklet implements the 15-bit or 7-bit LFSR logic.
  - The LFSR is clocked by its internal frequency timer. The output is typically the inverted bit 0 of the LFSR.
  - Volume envelope logic is applied.
  - show examples of basic noise generation with AudioWorklets, though the Game Boy's LFSR is specific.

Frame Sequencer, Envelopes, Sweep, Length Counters:

- The main emulator loop advances a master APU cycle counter.
- When enough cycles have passed to correspond to a 512Hz Frame Sequencer step, the emulator determines which units (length, envelope, sweep) should be clocked based on the current Frame Sequencer step.
- It then sends messages to the respective channel worklets to trigger updates in their internal state for these units.
- Alternatively, if not using worklets for full logic, AudioParam scheduling methods like linearRampToValueAtTime or setValueAtTime can be used for simpler envelope effects, but they may not perfectly replicate Game Boy behavior without careful management.

Mixing and Output:

- In the main thread, create instances of your AudioWorkletNodes:
  ```
  // After
  audioCtx.audioWorklet.addModule('path/to/PulseChannelProcessor.js'
  ```

```
) etc.
if (audioCtx && masterGainNode) {
    const channel1Node = new AudioWorkletNode(audioCtx,
'pulse-channel-processor');
    const channel2Node = new AudioWorkletNode(audioCtx,
'pulse-channel-processor'); // Separate instance
    const channel3Node = new AudioWorkletNode(audioCtx,
'wave-channel-processor');
    const channel4Node = new AudioWorkletNode(audioCtx,
'noise-channel-processor');

    channel1Node.connect(masterGainNode);
    channel2Node.connect(masterGainNode);
    channel3Node.connect(masterGainNode);
    channel4Node.connect(masterGainNode);
}
```

Resampling Challenges: The Game Boy's APU generates audio at very high internal effective sample rates (e.g., a pulse channel's waveform step rate can be over 1 MHz). The Web Audio API's AudioContext typically runs at a much lower sample rate (e.g., 44.1 kHz or 48 kHz). Directly outputting samples at the Game Boy's rate and letting the browser downsample naively can lead to significant **aliasing** artifacts (unwanted frequencies, buzzing, or ringing sounds).

- Proper solutions involve band-limited synthesis techniques (which are complex) or applying a digital low-pass filter before downsampling.
- For a simpler emulator, one might accept some aliasing or try to mitigate the worst effects (e.g., by ensuring generated frequencies don't exceed Nyquist limits too badly). Some emulators use libraries like blip_buf (though these are typically C/C++ and would require WebAssembly for web use) to handle high-quality resampling.

6. Player Interaction: Handling Input

Player interaction with the Game Boy is primarily through its built-in joypad, which features a directional pad (D-pad: Up, Down, Left, Right) and four action buttons (A, B, Select, Start).

Reading Input via JOYP Register ($FF00)

The state of these 8 buttons is read by the game software through a single memory-mapped I/O register: JOYP, located at address $FF00. The Game Boy employs a multiplexing scheme to read these 8 buttons using fewer I/O lines than would be needed if each button had a dedicated line. This works as follows:

- **Selection Bits (Writable):**
    - **Bit 5 (P15):** When this bit is written as 0 by the game, the action buttons (A, B, Select, Start) are selected for reading. When 1, they are deselected.
    - **Bit 4 (P14):** When this bit is written as 0 by the game, the direction buttons (Right, Left, Up, Down) are selected for reading. When 1, they are deselected. The game can only select one group (action or direction) at a time by writing 0 to either bit 4 or bit 5. Writing 1 to both effectively deselects all buttons for reading via bits 0-3.

- **State Bits (Read-Only):**
    - **Bits 0-3:** These bits reflect the pressed state of the currently selected group of buttons. A value of 0 in one of these bits means the corresponding button is **pressed**, and 1 means it is **not pressed**.
        - **Bit 0:** Reads Right (if directions selected) OR A (if actions selected).
        - **Bit 1:** Reads Left (if directions selected) OR B (if actions selected).
        - **Bit 2:** Reads Up (if directions selected) OR Select (if actions selected).
        - **Bit 3:** Reads Down (if directions selected) OR Start (if actions selected).
- **Unused Bits:**
    - **Bits 7-6:** These bits are not used and always read as 1.

The game software typically reads the joypad by first writing to $FF00 to select either the direction keys (e.g., writing $20, which sets bit 5 to 1 and bit 4 to 0) or the action keys (e.g., writing $10, which sets bit 5 to 0 and bit 4 to 1). After a brief moment for the hardware lines to stabilize, the game then reads from $FF00. The returned value will have bits 4 and 5 reflecting what was written (or the current state if nothing was written recently to select), and bits 0-3 will reflect the states of the selected buttons.

It is crucial for an emulator to accurately model this behavior. Simply storing whatever value the game writes to $FF00 and returning that same value on a read will not work. The read operation must dynamically construct the result by combining the current selection state of bits 4 and 5 with the actual hardware button states for bits 0-3.

Emulating Input in TypeScript

An input handling class (e.g., Joypad.ts) will manage the state of the Game Boy's buttons and interface with the MMU for reads/writes to $FF00.

Internal Joypad State: Maintain an object or boolean flags to store the current pressed state of all 8 Game Boy buttons, as reported by the host system (keyboard or gamepad).

```
// Inside Joypad.ts
export class Joypad implements IJoypad { // Assuming IJoypad interface
    // Internal state of actual button presses (true = pressed)
    private buttonStates: {
        right: boolean; left: boolean; up: boolean; down: boolean;
        a: boolean; b: boolean; select: boolean; start: boolean;
    } = {
        right: false, left: false, up: false, down: false,
        a: false, b: false, select: false, start: false
    };

    // Represents the value of the P1/JOYP register ($FF00)
    // Bits 0-3 are input lines (read-only from CPU perspective,
reflect button state)
    // Bits 4-5 are output lines (writable by CPU to select button
group)
    // Bits 6-7 are unused (always read as 1)
    private joypRegister: number = 0xFF; // Initial state, all lines
high (unselected, unpressed)

    // Reference to MMU or Interrupt Controller to request Joypad
```

```
interrupt
    private mmu: IMemory | null = null; // Or a dedicated interrupt
handler

    public connectMMU(mmu: IMemory): void {
        this.mmu = mmu;
    }

    // Called by MMU on write to $FF00
    public writeRegister(address: number, value: number): void {
        if (address === 0xFF00) {
            // Only bits 4 and 5 are writable by the game
            this.joypRegister = (this.joypRegister & 0xCF) | (value &
0x30);
        }
    }

    // Called by MMU on read from $FF00
    public readRegister(address: number): number {
        if (address === 0xFF00) {
            let result = (this.joypRegister & 0xF0) | 0xC0; // Start
with selection bits + unused bits (always 1)

            if (!((this.joypRegister >> 5) & 1)) { // Bit 5 is 0:
Action buttons selected
                if (this.buttonStates.a)      { result &= ~0x01; }
else { result |= 0x01; }
                if (this.buttonStates.b)      { result &= ~0x02; }
else { result |= 0x02; }
                if (this.buttonStates.select) { result &= ~0x04; }
else { result |= 0x04; }
                if (this.buttonStates.start)  { result &= ~0x08; }
else { result |= 0x08; }
            } else if (!((this.joypRegister >> 4) & 1)) { // Bit 4 is
0: Direction buttons selected
                if (this.buttonStates.right) { result &= ~0x01; } else
{ result |= 0x01; }
                if (this.buttonStates.left)  { result &= ~0x02; } else
{ result |= 0x02; }
                if (this.buttonStates.up)    { result &= ~0x04; } else
{ result |= 0x04; }
                if (this.buttonStates.down)  { result &= ~0x08; } else
{ result |= 0x08; }
            } else {
                // Neither group selected, or both selected (which
acts like neither for input bits)
                // Bits 0-3 should remain high (1) indicating no
buttons pressed in this case.
```

```
                result |= 0x0F;
            }
            return result;
        }
        return 0xFF; // Should not happen if MMU routes correctly
    }

    // --- Host Input Handling ---
    public setButtonState(buttonName: keyof Joypad, pressed: boolean):
void {
        const oldState = this.buttonStates[buttonName];
        this.buttonStates[buttonName] = pressed;

        if (pressed &&!oldState) { // Button just pressed
            // Check if the currently selected group in JOYP would
read this button
            const isActionSelected =!((this.joypRegister >> 5) & 1);
            const isDirectionSelected =!((this.joypRegister >> 4) &
1);
            let relevantPress = false;

            if (isActionSelected) {
                if ((buttonName === 'a' |
| buttonName === 'b' |
| buttonName === 'select' |
| buttonName === 'start')) {
                    relevantPress = true;
                }
            }
            if (isDirectionSelected) {
                 if ((buttonName === 'right' |
| buttonName === 'left' |
| buttonName === 'up' |
| buttonName === 'down')) {
                    relevantPress = true;
                }
            }

            if (relevantPress && this.mmu) {
                // Request Joypad interrupt (Bit 4 of IF register
0xFF0F)
                const ifReg = this.mmu.readByte(0xFF0F);
                this.mmu.writeByte(0xFF0F, ifReg | 0x10);
            }
        }
    }
}
```

Mapping Keyboard Events: Standard browser events keydown and keyup can be used to capture keyboard input. Map specific keyboard keys (e.g., arrow keys for D-pad, 'Z' for A, 'X' for B, 'Enter' for Start, 'Space' for Select) to update the buttonStates in the Joypad instance.

```
// In main emulator setup or UI interaction file
const gameboyJoypad = new Joypad();
//... connect gameboyJoypad to MMU...

window.addEventListener('keydown', (event: KeyboardEvent) => {
    mapKeyEvent(event.key, true);
});
window.addEventListener('keyup', (event: KeyboardEvent) => {
    mapKeyEvent(event.key, false);
});

function mapKeyEvent(key: string, pressed: boolean): void {
    switch (key.toUpperCase()) { // Using toUpperCase for
case-insensitivity
        case 'ARROWUP':    gameboyJoypad.setButtonState('up',
pressed); break;
        case 'ARROWDOWN':  gameboyJoypad.setButtonState('down',
pressed); break;
        case 'ARROWLEFT':  gameboyJoypad.setButtonState('left',
pressed); break;
        case 'ARROWRIGHT': gameboyJoypad.setButtonState('right',
pressed); break;
        case 'Z':          gameboyJoypad.setButtonState('a', pressed);
break;
        case 'X':          gameboyJoypad.setButtonState('b', pressed);
break;
        case 'ENTER':      gameboyJoypad.setButtonState('start',
pressed); break;
        case 'SHIFT': // Or ' ' for Space, 'CONTROL' for Select etc.
        case ' ':          gameboyJoypad.setButtonState('select',
pressed); break;
    }
}
```

Using the Gamepad API: For supporting game controllers, the Web Gamepad API is used.

1. Listen for gamepadconnected and gamepaddisconnected events on the window object.
2. Inside the main game loop (driven by requestAnimationFrame), call navigator.getGamepads() to get an array of connected Gamepad objects.
3. For each connected gamepad, inspect its buttons array (for A, B, Start, Select) and axes array (for D-pad directions). The buttons array elements have a pressed boolean property. Axes values range from -1.0 to 1.0; a threshold is used to determine D-pad presses (e.g., axes < -0.5 for Left).
4. Update the buttonStates in the Joypad instance based on the gamepad input.

```
// Inside main game loop or a dedicated input update function
function pollGamepads(): void {
    const gamepads = navigator.getGamepads();
    for (const gp of gamepads) {
        if (gp) {
            // Standard mapping (may vary by controller/OS)
            // D-Pad often on axes or buttons 12-15
            gameboyJoypad.setButtonState('a', gp.buttons?.pressed |
| false);      // Typically 'A' or Cross
            gameboyJoypad.setButtonState('b', gp.buttons?.pressed |
| false);      // Typically 'B' or Circle
            gameboyJoypad.setButtonState('select', gp.buttons?.pressed
|
| false); // Select/Back
            gameboyJoypad.setButtonState('start', gp.buttons?.pressed
|
| false);  // Start/Options

            // D-pad from buttons (common)
            gameboyJoypad.setButtonState('up', gp.buttons?.pressed |
| false);
            gameboyJoypad.setButtonState('down', gp.buttons?.pressed |
| false);
            gameboyJoypad.setButtonState('left', gp.buttons?.pressed |
| false);
            gameboyJoypad.setButtonState('right', gp.buttons?.pressed
|
| false);

            // Or D-pad from axes (less common for D-pad itself but
good for analog sticks)
            // if (gp.axes < -0.5) gameboyJoypad.setButtonState('up',
true); else...
            //...
            break; // Process first connected gamepad
        }
    }
}
// Call pollGamepads() in your requestAnimationFrame loop
```

Joypad Interrupt: When a button transitions from an unpressed state to a pressed state, a Joypad interrupt (Interrupt Flag register $FF0F, bit 4) should be requested if the game is currently interested in reading that type of button (i.e., if the corresponding selection bit 4 or 5 in $FF00 is 0). This signals the CPU that new input is available.

7. Loading Game Cartridges (ROMs)

Game Boy games are distributed as ROM (Read-Only Memory) files, typically with .gb (for original Game Boy) or .gbc (for Game Boy Color) extensions. Loading these files into the emulator is the first step to playing a game.

Understanding the ROM Header

The first part of a Game Boy ROM file contains a **header** region, spanning from address $0100 to $014F within the cartridge's memory map. This header provides metadata about the game and the cartridge hardware. Key fields include:

- **Entry Point ($0100 - $0103):** These four bytes typically contain a NOP instruction followed by a JP (jump) instruction to the main starting address of the game's code.
- **Nintendo Logo ($0104 - $0133):** A specific 48-byte bitmap representing the Nintendo logo. The Game Boy's internal boot ROM verifies this logo during startup; if it doesn't match, the console usually won't boot the game. Emulators may or may not enforce this check.
- **Title ($0134 - $0143):** The game's title, stored as ASCII characters. The length varies; for older games, it can be up to 16 characters. For Game Boy Color games or later monochrome games, this area is shorter to accommodate other GBC-specific flags.
- **Manufacturer Code ($013F - $0142):** (Newer ROMs) Identifies the game's manufacturer.
- **CGB Flag ($0143):** Indicates Game Boy Color compatibility. $80 typically means GBC compatible (but can run on DMG), $C0 means GBC only.
- **New Licensee Code ($0144 - $0145):** A two-character code for the game's licensee.
- **SGB Flag ($0146):** Indicates Super Game Boy support.
- **Cartridge Type ($0147):** A crucial byte specifying the type of Memory Bank Controller (MBC) used by the cartridge, and whether it includes RAM and/or a battery for saving. Examples:
    - $00: ROM ONLY
    - $01: MBC1
    - $02: MBC1+RAM
    - $03: MBC1+RAM+BATTERY
    - (Many other types exist for MBC2, MBC3, MBC5, etc.)
- **ROM Size ($0148):** A code indicating the total size of the ROM on the cartridge. The actual size is typically 32KB * (1 << value_at_0148).
- **RAM Size ($0149):** A code indicating the size of any external SRAM on the cartridge (e.g., for save files).
- **Destination Code ($014A):** (00 = Japanese, 01 = Non-Japanese).
- **Old Licensee Code ($014B):** An older, single-byte licensee code.
- **Mask ROM Version Number ($014C):** Usually $00.
- **Header Checksum ($014D):** An 8-bit checksum calculated from the bytes in the header from $0134 to $014C. The boot ROM verifies this.
- **Global Checksum ($014E - $014F):** A 16-bit sum of all bytes in the ROM (excluding these two checksum bytes themselves). This is rarely checked by actual hardware but can be used by emulators to verify ROM integrity.

While some header information (like ROM size declared in the header) might not be strictly enforced by real hardware (which determines size by the physical chip capacity), the Cartridge Type byte is extremely important for emulators. It dictates which MBC needs to be emulated for the game to function correctly, including accessing ROM beyond the initial 32KB banks and managing SRAM. Failure to support or correctly emulate the specified MBC will render many games unplayable or unable to save progress.

Loading ROMs in a Web Environment

Web browsers provide APIs to allow users to select local files and for JavaScript to read their contents.
File Input Element: An HTML <input type="file"> element is the standard way to let users choose a ROM file from their computer.

```
<label for="romFile">Load Game Boy ROM:</label>
<input type="file" id="romFile" accept=".gb,.gbc" />
```

FileReader API: Once a file is selected, the FileReader API is used to read its content as an ArrayBuffer. This ArrayBuffer contains the raw binary data of the ROM file.

```
// In your emulator's UI handling code
const romFileInput = document.getElementById('romFile') as
HTMLInputElement | null;

if (romFileInput) {
    romFileInput.addEventListener('change', (event: Event) => {
        const target = event.target as HTMLInputElement;
        const file = target.files?.;

        if (file) {
            const reader = new FileReader();
            reader.onload = (loadEvent: ProgressEvent<FileReader>) =>
{
                if (loadEvent.target?.result instanceof ArrayBuffer) {
                    const arrayBuffer = loadEvent.target.result;
                    // Pass this arrayBuffer to the emulator's ROM
loading function
                    // e.g., myGameBoyEmulator.loadRom(arrayBuffer);
                    console.log(`ROM loaded: ${file.name}, size:
${arrayBuffer.byteLength} bytes`);
                } else {
                    console.error("Failed to read file as
ArrayBuffer.");
                }
            };
            reader.onerror = () => {
                console.error("Error reading file:", reader.error);
            };
            reader.readAsArrayBuffer(file);
        }
    });
}
```

Parsing Basic ROM Header Information in TypeScript: After obtaining the ArrayBuffer, create a Uint8Array view to easily access individual bytes. Then, extract information from the header addresses.

```
// Inside your emulator class, e.g., GameBoy.ts
// Assume this.mmu is the Memory Management Unit instance, which has a
GameBoyMemory instance.
public loadRom(romArrayBuffer: ArrayBuffer): void {
    const romData = new Uint8Array(romArrayBuffer);

    // Store the full ROM data, potentially in the MMU or a Cartridge
class
    this.mmu.getMemory().loadRom(romData); // Assuming GameBoyMemory
has a method to handle this

    // --- Parse Header Information ---
    // Nintendo Logo (0x0104 - 0x0133) - can be used for verification
    // const nintendoLogo = romData.slice(0x0104, 0x0134);

    let title = "";
    // Title can be up to 16 chars, or less if CGB flag is set
    const cgbFlag = romData[0x0143];
    const titleEndAddress = (cgbFlag === 0x80 |
| cgbFlag === 0xC0)? 0x013F -1 : 0x0143; // Adjust for GBC title
length
    for (let i = 0x0134; i <= titleEndAddress; i++) {
        if (romData[i] === 0x00) break; // Null terminator
        title += String.fromCharCode(romData[i]);
    }
    this.gameTitle = title.trim(); // Store for display or save key
    console.log("Game Title:", this.gameTitle);

    this.cartridgeType = romData[0x0147];
    console.log("Cartridge Type: 0x" +
this.cartridgeType.toString(16).toUpperCase());
    // Based on this.cartridgeType, instantiate the correct MBC
handler

    const romSizeCode = romData[0x0148];
    this.romSize = (32 * 1024) * (1 << romSizeCode); // 32KB <<
romSizeCode
    console.log("ROM Size:", this.romSize / 1024, "KB");

    const ramSizeCode = romData[0x0149];
    // Map ramSizeCode to actual SRAM size (e.g., 00=0KB, 02=8KB,
03=32KB)
    // this.sramSize = mapRamSizeCodeToBytes(ramSizeCode);
    // console.log("SRAM Size:", this.sramSize / 1024, "KB");

    // Initialize/reset CPU, PPU, APU, etc. to start the game
    this.reset();
}
```

The parsed ROM data, particularly the initial banks, is then loaded into the emulated memory regions managed by the MMU. The Cartridge Type is used to configure the appropriate MBC emulation if the game requires one.

8. Structuring Your TypeScript Emulator Project

A well-structured project is crucial for managing the complexity of an emulator. TypeScript's features like interfaces and classes help in creating modular and maintainable code.

Defining Interfaces for Core Components

Using interfaces to define the contracts for each major hardware component (CPU, PPU, APU, MMU, Timer, Joypad, etc.) is a good practice. This promotes loose coupling, making it easier to develop, test, and replace components independently.

```
// Example interfaces (can be more detailed)

// CPURegisters class would be defined elsewhere
// class CPURegisters { /*... */ }

interface ICPU {
    registers: CPURegisters;
    step(): number; // Executes one instruction, returns T-cycles
consumed
    requestInterrupt(interruptType: number): void; // e.g., VBLANK_INT
= 0, LCD_STAT_INT = 1
    halted: boolean;
    ime: boolean; // Interrupt Master Enable
    reset(): void;
}

interface IPPU {
    step(cpuCycles: number): void; // Advances PPU state by given CPU
T-cycles
    readRegister(address: number): number;
    writeRegister(address: number, value: number): void;
    readVRAM(address: number): number; // VRAM addresses are
0x0000-0x1FFF relative to VRAM start
    writeVRAM(address: number, value: number): void;
    readOAM(address: number): number;  // OAM addresses are 0x00-0x9F
relative to OAM start
    writeOAM(address: number, value: number): void;
    getFrameBuffer(): Uint8ClampedArray; // For rendering to canvas
    isFrameReady(): boolean; // Indicates a new frame is complete
    clearFrameReady(): void;
    reset(): void;
}
```

```
interface IAPU {
    step(cpuCycles: number): void;
    readRegister(address: number): number;
    writeRegister(address: number, value: number): void;
    // Method to get audio samples or manage AudioWorkletNodes
    reset(): void;
}

interface ITimer {
    step(cpuCycles: number): void;
    readRegister(address: number): number;
    writeRegister(address: number, value: number): void;
    reset(): void;
}

interface IJoypad {
    readRegister(address: number): number;
    writeRegister(address: number, value: number): void;
    setButtonState(buttonName: string, pressed: boolean): void; //
e.g., 'up', 'a'
    reset(): void;
}

interface IMemory { // This would be the MMU
    readByte(address: number): number;
    writeByte(address: number, value: number): void;
    readWord(address: number): number;  // Helper for 16-bit reads
    writeWord(address: number, value: number): void; // Helper for
16-bit writes
    getMemory(): GameBoyMemory; // Access to raw memory regions if
needed by Cartridge/MBC
    loadBootRom?(bootRomData: Uint8Array): void;
    reset(): void;
}
```

Organizing Code into Modules/Classes

Each emulated hardware component should reside in its own TypeScript file (module) and be implemented as a class that conforms to its defined interface. For example:

- cpu.ts (implements ICPU)
- ppu.ts (implements IPPU)
- apu.ts (implements IAPU)
- mmu.ts (implements IMemory)
- timer.ts (implements ITimer)
- joypad.ts (implements IJoypad)
- cartridge.ts (handles ROM data, header parsing, MBC logic)

- gameboy.ts (the main emulator class that orchestrates everything)

The main GameBoy class will instantiate these components and wire them together (e.g., passing the MMU instance to the CPU, PPU, APU so they can read/write memory and I/O registers) [shows a Gameboy class with properties like cpu, gpu (PPU), apu, memory].

The Main Emulator Loop and Component Synchronization

The heart of the emulator is its main loop, which drives the simulation forward. In a web browser, this loop is best implemented using requestAnimationFrame for smooth rendering synchronized with the browser's display refresh rate.

The synchronization strategy between the CPU, PPU, and APU is critical. These components on real hardware operate based on a shared clock. In an emulator, their execution must be interleaved correctly to maintain relative timing. A common and effective method is the "catch-up" approach :

1. **CPU Execution:** The CPU is allowed to execute a certain number of instructions or run for a certain number of clock cycles.
2. **Peripheral Catch-Up:** The number of clock cycles consumed by the CPU is then passed to the other timed peripherals (PPU, APU, Timers). These peripherals then advance their own internal states by that many cycles.

Main Loop Structure:

```
// Inside GameBoy.ts or a similar main orchestrator class
private cpu: ICPU;
private ppu: IPPU;
private apu: IAPU;
private timer: ITimer;
private mmu: IMemory; // MMU instance
private joypad: IJoypad;

private running: boolean = false;
private lastFrameTime: number = 0;
private accumulatedCycles: number = 0;
// Target cycles per frame: GameBoy CPU is ~4.19 MHz. Screen refresh
is ~59.7 Hz.
// Cycles per frame = 4194304 / 59.7 ~= 70224 cycles
private readonly CYCLES_PER_FRAME = 70224; // Or a fraction for
smaller time slices

constructor() {
    // Instantiate all components and their dependencies
    //...
}

public run(): void {
    if (!this.romLoaded) { // Assume a romLoaded flag
        console.error("No ROM loaded.");
        return;
    }
    this.running = true;
    this.lastFrameTime = performance.now();
    requestAnimationFrame(this.gameLoop.bind(this));
}

public pause(): void {
    this.running = false;
}

private gameLoop(currentTime: number): void {
    if (!this.running) return;

    const deltaTime = currentTime - this.lastFrameTime;
    this.lastFrameTime = currentTime;

    // Calculate how many CPU cycles to emulate this frame/timeslice
    // More sophisticated timing can aim for CYCLES_PER_FRAME over
16.67ms
    const cyclesToRunThisStep = Math.min(this.CYCLES_PER_FRAME / 4, //
Run in smaller chunks for better responsiveness
                                       (deltaTime / 1000) * 4194304);


    this.accumulatedCycles = 0;
    while (this.accumulatedCycles < cyclesToRunThisStep) {
        if (this.cpu.halted &&!this.cpu.ime &&
(this.mmu.readByte(0xFFFF) & this.mmu.readByte(0xFF0F) & 0x1F) === 0)
{
            // CPU is HALTed and no pending enabled interrupts,
effectively skip cycles until next interrupt
            // This needs careful handling of interrupt checks.
            // For simplicity here, we just advance by a minimum cycle
count.
            const minCycles = 4;
            this.timer.step(minCycles);
            this.ppu.step(minCycles);
            this.apu.step(minCycles);
            this.accumulatedCycles += minCycles;
            // Check for interrupts that might wake the CPU
            this.handleInterrupts();
            if ((this.mmu.readByte(0xFFFF) & this.mmu.readByte(0xFF0F)
& 0x1F)!== 0) {
                 this.cpu.halted = false;
            } else {
                // Still halted, break and wait for real interrupt or
next frame
                // This part is tricky and needs robust HALT bug
emulation
                break;
            }
        }

        const cyclesConsumed = this.cpu.step(); // CPU executes one
instruction
        this.accumulatedCycles += cyclesConsumed;

        // Peripherals catch up
        this.timer.step(cyclesConsumed);
        this.ppu.step(cyclesConsumed);
        this.apu.step(cyclesConsumed);

        this.handleInterrupts(); // Check and dispatch interrupts to
CPU
    }


    // After running a batch of cycles, check if PPU has a frame ready
    if (this.ppu.isFrameReady()) {
        this.renderFrameToCanvas(); // Method that calls
getFrameBuffer and putImageData
        this.ppu.clearFrameReady();
    }

    // Poll gamepad input
    this.pollGamepadInput(); // Assuming a method that updates joypad
state

    requestAnimationFrame(this.gameLoop.bind(this));
}

private handleInterrupts(): void {
    if (this.cpu.ime) { // Only if Interrupt Master Enable is true
        const ie = this.mmu.readByte(0xFFFF); // Interrupt Enable
register
        const iflag = this.mmu.readByte(0xFF0F); // Interrupt Flag
register
        const pendingAndEnabled = ie & iflag & 0x1F; // Mask for 5
interrupt types

        if (pendingAndEnabled!== 0) {
            this.cpu.halted = false; // Wake CPU if halted
            if ((pendingAndEnabled & 0x01)!== 0) { // VBlank
                this.cpu.requestInterrupt(0); //
VBLANK_INT_VECTOR_ADDRESS_0x40
            } else if ((pendingAndEnabled & 0x02)!== 0) { // LCD STAT
                this.cpu.requestInterrupt(1); //
LCD_STAT_INT_VECTOR_ADDRESS_0x48
```

```
            } else if ((pendingAndEnabled & 0x04)!== 0) { // Timer
                this.cpu.requestInterrupt(2); //
TIMER_INT_VECTOR_ADDRESS_0x50
            } else if ((pendingAndEnabled & 0x08)!== 0) { // Serial
                this.cpu.requestInterrupt(3); //
SERIAL_INT_VECTOR_ADDRESS_0x58
            } else if ((pendingAndEnabled & 0x10)!== 0) { // Joypad
                this.cpu.requestInterrupt(4); //
JOYPAD_INT_VECTOR_ADDRESS_0x60
            }
        }
    }
}
// Other methods like renderFrameToCanvas, pollGamepadInput, reset,
etc.
```

shows a similar loop (jsGB.frame) that runs for a fixed number of total clock cycles (70224 for one full frame) and calls GPU.step() (equivalent to PPU step) within the CPU instruction processing loop. This ensures tight synchronization. The granularity of this synchronization (per CPU instruction, per small batch of cycles, or per larger time slice) impacts both emulation accuracy and browser performance. Running in smaller, more frequent batches tends to improve responsiveness and can be more accurate for time-sensitive PPU effects, but may have higher overhead.
If components operate on independent timers or are too loosely coupled, their internal states can diverge significantly from what would occur on actual hardware. For example, if the PPU runs too slowly relative to the CPU, the CPU might attempt to write to VRAM when the PPU should be actively using it for rendering, or CPU reads of the LY register (current scanline) might be stale, leading to incorrect game logic or graphical glitches. Similarly, if the APU is out of sync, sound effects will not align with on-screen actions, and music tempo can be wrong.

9. Performance Considerations and Next Steps

Building a Game Boy emulator that runs well in a web browser presents unique challenges related to performance and the nuances of web technologies.

Common Pitfalls in Web-Based Emulation

- **Input Latency:** This is the delay between a physical input (like a key press or gamepad button press) and the corresponding action appearing on the emulated screen. Web-based emulators can suffer from multiple sources of latency: browser event handling delays, the emulator's own processing time per frame, and the display's refresh rate. High input latency can make games feel unresponsive or "laggy," significantly degrading the player experience.
- **Audio Glitches and Latency:** Web Audio API, while powerful, can be tricky. Issues like clicks, pops, stuttering, or desynchronized audio are common if not handled carefully. These can arise from buffer underruns (not enough audio data ready when the browser needs it) or overruns, or if the APU emulation or main loop performance is inconsistent,

leading to irregular delivery of audio samples to the browser. AudioWorklet is designed to mitigate some of these issues by running audio processing in a dedicated thread, but it requires careful implementation to manage communication with the main emulator thread and maintain timing.

- **JavaScript Performance:** Emulation is computationally intensive, involving tight loops for CPU instruction execution, PPU rendering, and APU sound synthesis. JavaScript, even with modern JIT (Just-In-Time) compilation, can be a bottleneck. Inefficient algorithms, frequent memory allocations (garbage collection pressure), or poorly optimized code in critical paths can lead to slowdowns and frame drops. Heavy operations within the main requestAnimationFrame loop can also cause jank.

Achieving a "good feel" in an emulator, especially one running in the constrained environment of a web browser, is not solely about correctly implementing the hardware logic. It also heavily depends on actively managing these performance aspects and minimizing latency across input, audio, and video. These factors are often intertwined; for example, if the CPU emulation is too slow, it will impact PPU rendering rates and APU sample generation, potentially causing both visual and auditory problems.

Basic Optimization Tips for JavaScript/TypeScript

- **Profiling:** Use the browser's built-in developer tools (Performance tab) to profile the emulator and identify performance bottlenecks. This will show which functions are consuming the most CPU time.
- **Efficient Data Structures:** Utilize TypedArrays (like Uint8Array, Uint16Array, Float32Array) for memory regions, pixel buffers, and audio samples. They are much more memory-efficient and faster for numerical operations than standard JavaScript arrays.
- **Minimize Work in Hot Loops:** The core CPU execution loop and PPU rendering routines are "hot paths." Optimize these sections carefully. Avoid string manipulations, excessive object creation/destruction, or complex conditional logic within these loops if possible.
- **Reduce Garbage Collection:** Frequent creation and discarding of objects can lead to garbage collection pauses. Try to reuse objects or use object pooling where appropriate.
- **Web Workers (Cautiously):** While Web Workers can offload some tasks to background threads, the core emulation loop (CPU-PPU-APU synchronization) often needs to be tightly coupled, making it difficult to parallelize effectively without introducing complex synchronization mechanisms that might negate performance gains. However, auxiliary tasks like UI updates or complex audio post-processing could potentially be offloaded.
- **Bitwise Operations:** Use bitwise operators (&, |, ^, ~, <<, >>, >>>) for register manipulation, flag handling, and pixel decoding, as they are generally very fast.
- **Cache Pre-calculated Values:** If certain values are frequently recalculated but don't change often, consider caching them.

Further Areas of Study

Once a basic Game Boy emulator is up and running, there are many avenues for improvement and further learning:

- **Cycle Accuracy:** Striving for more precise emulation of the exact number of clock cycles each CPU instruction takes, and how PPU and APU operations align with these cycles. This is complex but crucial for compatibility with games that rely on very specific hardware timing for effects or game logic.

- **Hardware Quirks and Bugs:** Real Game Boy hardware has numerous undocumented behaviors, timing oddities, and even minor bugs (e.g., the OAM bug, specific PPU STAT interrupt conditions). Emulating these quirks is often necessary to make certain games run correctly. This often requires consulting detailed hardware tests and community documentation.
- **Memory Bank Controllers (MBCs):** As mentioned, most Game Boy games use MBCs (MBC1, MBC2, MBC3, MBC5 being common) to manage access to larger ROMs and SRAM. Each MBC type has its own set of registers mapped into the ROM address space and specific logic for bank switching. Implementing these is essential for broad game compatibility.
- **Game Boy Color (GBC) Emulation:** This is a significant step up, involving:
    - Color graphics (palettes for background and sprites).
    - Double-speed CPU mode.
    - Increased VRAM and WRAM.
    - Different PPU features and registers.
    - Infrared port.
- **Sound Improvements:** Advanced techniques for band-limited synthesis to reduce aliasing, or more accurate emulation of the APU's analog characteristics.
- **Debugging Tools:** Integrating features like a memory viewer, register inspector, CPU disassembler, PPU tile viewer, or breakpoint capabilities directly into the emulator can be invaluable for development and for understanding how games work.
- **Save States:** Allowing users to save and load the entire emulator state at any point.
- **Netplay:** Implementing multiplayer capabilities over the network.

The journey of emulator development is inherently iterative. It's common to start with basic functionality, focusing on getting the core components (CPU, a simple memory model, basic PPU output) working for a simple ROM. From there, accuracy and features are gradually added and refined. Test ROMs—small programs specifically designed to test particular aspects of the hardware—are invaluable tools throughout this process, providing clear pass/fail indicators for different functionalities.

10. Conclusion

The Nintendo Game Boy, with its relatively simple yet capable architecture, serves as an excellent platform for learning the intricacies of hardware emulation. Its core components—the custom Sharp LR35902 CPU, the mapped memory system, the tile-based PPU, the distinctive four-channel APU, and the straightforward input mechanism—each present unique and engaging challenges when translated into a software simulation.

Developing a Game Boy emulator in a web-based environment using TypeScript leverages modern web technologies like the Canvas API for rendering, the Web Audio API (particularly AudioWorklets) for sound synthesis, and standard JavaScript event handling for input. The process involves not only understanding the function of each hardware unit but also their precise timing and interactions. Key considerations include the CPU's fetch-decode-execute cycle, the MMU's role in routing memory accesses and handling I/O registers, the PPU's scanline-based rendering and mode-dependent memory access restrictions, the APU's frame sequencer and channel-specific sound generation, and the joypad's multiplexed input reading.

While a "simple" introduction can get a basic emulator running, achieving high compatibility and an authentic "feel" requires attention to detail, including nuanced timing, hardware quirks, and robust handling of web platform limitations, especially concerning performance and audio/visual latency. The path of emulator development is one of continuous refinement, often guided by test ROMs and a deeper dive into the Game Boy's extensive documentation and community knowledge. By breaking down the system into its constituent parts and tackling their emulation systematically, developers can gain a profound appreciation for the engineering behind this iconic handheld and the art of software preservation through emulation.

Works cited

1. DMG-01: How to Emulate a Game Boy, https://rylev.github.io/DMG-01/public/book/print.html
2. GameBoy-presentation.pdf,
http://www.cs.columbia.edu/~sedwards/classes/2019/4840-spring/reports/GameBoy-presentation.pdf 3. The Nintendo® Game Boy™, Part 1: The Intel 8080 and the Zilog Z80. | RealBoy,
https://realboyemulator.wordpress.com/2013/01/01/the-nintendo-game-boy-1/ 4. gameboy/README.md at main - GitHub,
https://github.com/raphamorim/LR35902/blob/main/README.md 5. GameBoy Emulation in JavaScript: Interrupts - Imran Nazar,
https://imrannazar.com/series/gameboy-emulation-in-javascript/interrupts 6. Emulation accuracy - Emulation General Wiki, https://emulation.gametechwiki.com/index.php/Emulation_accuracy 7. Writing axle's GameBoy emulator | Phillip Tennen,
https://axleos.com/writing-axles-gameboy-emulator/ 8. Gameboy: The Memory Question : r/EmuDev - Reddit,
https://www.reddit.com/r/EmuDev/comments/1g9wm45/gameboy_the_memory_question/ 9. Memory - GB ASM Tutorial - gbdev.io, https://gbdev.io/gb-asm-tutorial/part1/memory.html 10. Conware: Automated Modeling of Hardware Peripherals - PurS3 Lab,
https://purs3lab.github.io/files/conware.pdf 11. Joypad - Gameboy Emulation - codeslinger.co.uk, http://www.codeslinger.co.uk/pages/projects/gameboy/joypad.html 12. Gameboy OAM Scan Accesses : r/EmuDev - Reddit,
https://www.reddit.com/r/EmuDev/comments/1h9pwxf/gameboy_oam_scan_accesses/ 13. How to emulate GameBoy PPU : r/EmuDev - Reddit,
https://www.reddit.com/r/EmuDev/comments/1jf52vn/how_to_emulate_gameboy_ppu/ 14. GBEDG | The Gameboy Emulator Development Guide, https://hacktix.github.io/GBEDG/ppu/
15. How To Homebrew Game Boy Games - DigiKey,
https://www.digikey.com/en/maker/projects/how-to-homebrew-game-boy-games/508defd7091c4a2eb912647109097284 16. Question about the gameboy window x-coordinate register : r/EmuDev - Reddit,
https://www.reddit.com/r/EmuDev/comments/10orf0d/question_about_the_gameboy_window_xcoordinate/ 17. GameBoy Emulation in JavaScript: Integration - Imran Nazar,
https://imrannazar.com/series/gameboy-emulation-in-javascript/integration 18. Gameboy Tile rendering - 2 bytes per pixel - javascript - GitHub Gist,
https://gist.github.com/0648be7f9dcd55e55dd2a54e928b7da5 19. Gameboy emulator written in TypeScript - GitHub, https://github.com/roblouie/gameboy-emulator 20. CanvasRenderingContext2D: putImageData() method - Web APIs | MDN,
https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/putImageData
21. CanvasRenderingContext2D: getImageData() method - Web APIs | MDN,
https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/getImageData
22. The Game Loop | gablaxian.com,
https://gablaxian.com/articles/creating-a-game-with-javascript/the-game-loop/ 23. The Game

Boy APU | jsgroth's blog, https://jsgroth.dev/blog/posts/gb-rewrite-apu/ 24. Gameboy sound hardware - GbdevWiki - gg8, https://gbdev.gg8.se/wiki/articles/Gameboy_sound_hardware 25. Game Boy Sound Emulation | NightShade's Blog, https://nightshade256.github.io/2021/03/27/gb-sound-emulation.html 26. Getting started with Web Audio API | Articles - web.dev, https://web.dev/articles/webaudio-intro 27. Make "Game Boy" music in JavaScript - Adrien Joly, https://adrienjoly.com/slides-webaudio-gameboy-music/ 28. Wasm Audio Worklets API — Emscripten 4.0.9-git (dev) documentation, https://emscripten.org/docs/api_reference/wasm_audio_worklets.html 29. LFSR Noise Generator - Questions - scsynth, https://scsynth.org/t/lfsr-noise-generator/5553 30. WebAudio Experiments - AudioWorklet Noise - SoftSynth, https://www.softsynth.com/webaudio/custom_noise.php 31. Synthesis with the Web Audio API - Envelopes - Chris Lowis, https://chrislowis.co.uk/2013/06/17/synthesis-web-audio-api-envelopes 32. Web Audio API scheduling to build a sequencer. I don't get it - Stack Overflow, https://stackoverflow.com/questions/22926572/web-audio-api-scheduling-to-build-a-sequencer-i-dont-get-it 33. Variable duty cycle square waves with the Web Audio API | Hacker News, https://news.ycombinator.com/item?id=43613180 34. ITotalJustice/gb_apu: implementation of the Gameboy apu in ~2k lines of code. - GitHub, https://github.com/ITotalJustice/gb_apu 35. How gameboy's joypad works? - Reddit, https://www.reddit.com/r/Gameboy/comments/13m92f2/how_gameboys_joypad_works/ 36. GameBoy Emulation in JavaScript: Input - Imran Nazar, https://imrannazar.com/series/gameboy-emulation-in-javascript/input 37. Playing with the Gamepad API - Alvaro Montoro, https://alvaromontoro.com/blog/68044/playing-with-the-gamepad-api 38. How to use JavaScript Gamepad API to Build a Simple Game - YouTube, https://m.youtube.com/watch?v=GOjMP6WY8CU&pp=ygUII2dhbWVhcGk%3D 39. The header - GB ASM Tutorial - gbdev.io, https://gbdev.io/gb-asm-tutorial/part1/header.html 40. gameboy.html - HTML - OneCompiler, https://onecompiler.com/html/424butjrm 41. What Are Typescript Interfaces and How Do They Work? - Strapi, https://strapi.io/blog/typescript-interfaces 42. Writing a Game Boy Emulator in OCaml - The Linoscope Machine, https://linoscope.github.io/writing-a-game-boy-emulator-in-ocaml/ 43. 1. Why your terrible at retro games on your emulator and how you can play better without practice. And 2. Authentic hardware solutions if you have no retro consoles : r/retrogaming - Reddit, https://www.reddit.com/r/retrogaming/comments/1kjpr07/1_why_your_terrible_at_retro_games_on_your/ 44. Why does retro video game emulation not feel good? Latency analysis - SingleLunch, https://singlelunch.com/2018/06/12/can-game-emulation-feel-good-latency-analysis/ 45. Retro Boy: simple Game Boy emulator written in Rust, can be played on the web - Brian Lovin, https://brianlovin.com/hn/43429417 46. 60 fps gameboy emulation - Playdate Developer Forum, https://devforum.play.date/t/60-fps-gameboy-emulation/22865 47. Low-Level JavaScript Performance Best Practices (Crash Course) - YouTube, https://www.youtube.com/watch?v=koky8mDdtAk