AI VOID PROJECT LAB

Build a Game Boy Emulator with F#

Embark on a deep dive into system emulation by building a Game Boy emulator in F#. This guide covers CPU, memory, graphics, input, and sound from scratch, focusing on low-level design.

AI VOID

2026-05-05 19:20 IST

Contents

01	Setting Up Your Emulator Development Environment	3
02	The CPU Core: Registers, Flags, and Basic Instructions	16
03	Memory Management Unit (MMU) and Basic Memory Access	34
04	Loading ROMs and Initial Boot Sequence	49
05	CPU Control Flow: Jumps, Calls, and Conditional Logic	63
06	Interrupts and the Main CPU Execution Loop	81
07	Picture Processing Unit (PPU) Part 1: VRAM and Background Rendering	101
08	Picture Processing Unit (PPU) Part 2: Sprites, Scrolling, and LCD Control	123
09	Input Handling: Connecting Keyboard to Game Boy Buttons	147
10	Advanced MMU: Memory Bank Controllers (MBCs)	163
11	Audio Processing Unit (APU) Basics: Square Wave Channels	180
12	Synchronization, Debugging, and Verifying with Test ROMs	207
13	Building a Game Boy Emulator with F#	226

CHAPTER 01

Setting Up Your Emulator Development Environment

Building a Game Boy emulator from scratch is a deeply rewarding project that takes you into the heart of computer architecture and low-level system design. This journey begins by establishing a robust and efficient development environment. In this chapter, we'll set up everything you need: the F# language, the .NET SDK, and a powerful cross-platform graphics library to bring your emulator to life.

By the end of this chapter, you'll have a fully configured F# project, ready to accept the intricate logic of Game Boy hardware. You'll also confirm that your graphics setup is functional, providing the visual canvas for the pixels your Picture Processing Unit (PPU) will eventually render. This foundational step is critical; a well-prepared environment ensures you can focus on the complex emulation logic without fighting your tools.

Project Overview: Emulating a Game Boy

Our goal is to create a functional Game Boy emulator in F# capable of loading and running simple ROMs. This project isn't just about recreating a classic console; it's about understanding the intricate dance between a CPU, memory, and graphics hardware at a fundamental level. We'll dissect the Game Boy's architecture and rebuild it in software, component by component.

This project will build a desktop application, prioritizing cross-platform compatibility where feasible, primarily for Windows, macOS, and Linux. Performance will be a continuous consideration, especially for the CPU and PPU, as emulators demand precise timing and efficient resource management to run games smoothly.

Tech Stack Decisions

Choosing the right tools is paramount for a project of this complexity. Our primary technology stack is designed for type safety, performance, and cross-platform reach:

- **F# Language:** We'll leverage F#'s strong type system, immutability-first approach, and functional programming paradigms. This helps manage

complexity, reduce bugs, and write concise, expressive code, which is invaluable when dealing with low-level hardware specifications.

- **.NET SDK:** Provides the runtime, compilers, and tooling for F#. The .NET platform offers excellent performance and cross-platform capabilities, allowing our emulator to run on various operating systems.
- **Vortice.SDL2:** This is a .NET binding for SDL (Simple DirectMedia Layer), a widely adopted, cross-platform development library designed to provide low-level access to audio, keyboard, mouse, joystick, and graphics hardware. It's ideal for emulators because it offers direct pixel manipulation and window management without the overhead of higher-level GUI frameworks.

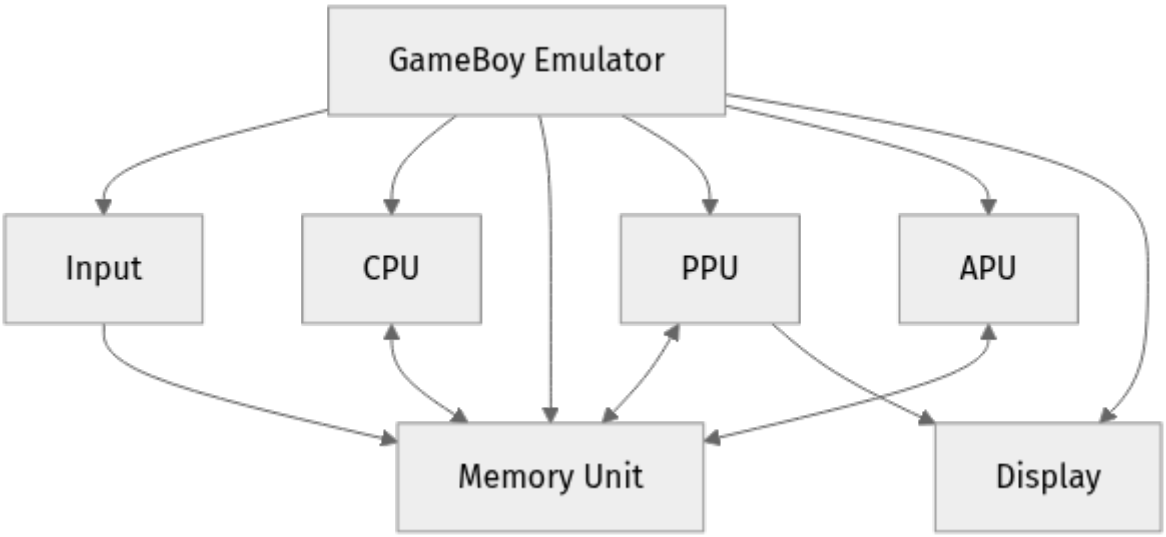
📌 `Key Idea:` F# and .NET provide a powerful, type-safe, and performant environment for systems programming, while `Vortice.SDL2` delivers the necessary low-level graphics control for an emulator.

Architecture and Build Plan

Our emulator's architecture will mirror the Game Boy's hardware components, promoting modularity and testability. Each major component will be a distinct F# module or set of modules.

High-Level Architecture

The core of our emulator will involve several interconnected modules:



- **CPU (SM83):** Interprets and executes Game Boy instructions.

- **MMU (Memory Management Unit):** Manages all memory addresses, including cartridge ROM/RAM, internal RAM, VRAM, and I/O registers. This is the central hub for data access.
- **PPU (Picture Processing Unit):** Responsible for rendering the Game Boy's display based on VRAM data and LCD control registers.
- **APU (Audio Processing Unit):** Handles sound generation.
- **Input Handler:** Reads user input (keyboard) and translates it into Game Boy button presses.
- **Display Output:** Our `Vortice.SDL2` integration will provide the window and render the PPU's output.

Incremental Milestones for the Project

This project guide breaks down the complex task of emulator development into manageable milestones. Here's a glimpse of the initial chapters:

1. **Setting Up Your Emulator Development Environment (This Chapter):** Configure F#, .NET, and SDL.
2. **Basic CPU Emulation (Registers, Flags, Simple Opcodes):** Begin implementing the CPU's core state and instruction execution.
3. **Memory Map & Basic MMU (RAM, ROM Loading):** Model the Game Boy's memory layout and load game ROMs.
4. **Advanced CPU (Interrupts, Stack, More Opcodes):** Flesh out the CPU with interrupt handling and stack operations.
5. **PPU Part 1 (VRAM, Tiles, Background Rendering):** Start rendering basic graphics, focusing on tiles and background.

`⚡ Quick Note:` Emulator development is highly iterative. We'll build a small piece, verify it, and then expand. This approach helps isolate bugs and maintain sanity.

Step-by-Step Environment Setup

Let's get our hands dirty and set up the development environment.

1. Install the .NET SDK

The .NET SDK is the foundation, providing the F# compiler, runtime, and essential development tools.

Current Version (as of 2026-05-05): We will target **.NET 9.0**. This is the latest stable release at this time, offering performance improvements and new language features.

1. **Download:** Navigate to the official .NET website: https://dotnet.microsoft.com/download
2. **Install:** Download and execute the installer appropriate for your operating system (Windows, macOS, or Linux). Follow the on-screen instructions.

`⚡ Quick Note:` If you have a newer version of .NET installed, that's generally fine. The .NET SDK supports side-by-side installations, meaning multiple versions can coexist on your system.

2. Choose and Configure Your Code Editor

A capable code editor with F# support significantly boosts productivity.

- **Visual Studio Code (Recommended for Cross-Platform):**
    1. Download and install VS Code: https://code.visualstudio.com/download
    2. Install the **Ionide-fsharp** extension. This extension provides robust F# tooling, including IntelliSense, syntax highlighting, type inference information, and debugging capabilities.
        - Open VS Code, access the Extensions view (typically `Ctrl+Shift+X` or `Cmd+Shift+X`).
        - Search for "Ionide-fsharp" and select "Install".
- **Visual Studio (Windows/macOS):**
    1. Download and install Visual Studio (the Community Edition is free for individual developers): https://visualstudio.microsoft.com/downloads/
    2. During the installation process, ensure you select the ".NET desktop development" workload. This workload includes comprehensive F# support.

3. Create Your F# Project

Now, let's establish the project directory and initial F# console application.

1. **Open your terminal or command prompt.**
2. **Create a new console application:**
   `bash dotnet new console -lang F# -n GameBoyEmulator` This command initializes a new F# console project named `GameBoyEmulator`

within a new directory of the same name. The `-lang F#` explicitly specifies F# as the language.

3. **Navigate into the newly created project directory:** `bash cd GameBoyEmulator`
4. **Open the project in your chosen editor:** `bash code .` (If using VS Code. For Visual Studio, open the `GameBoyEmulator.fsproj` file.)

You should now see a file named `Program.fs` containing a basic "Hello, World!" program.

4. Set Up SDL for Graphics

This step integrates `Vortice.SDL2`, our chosen library for window management and pixel rendering.

a. Install Native SDL2 Libraries

🧠 `Important:` `Vortice.SDL2` is a .NET binding (wrapper) around the native SDL2 library. It does not include the native binaries. Therefore, you **must** install the native SDL2 runtime libraries on your system for `Vortice.SDL2` to function correctly.

- **Windows:**
    1. Visit the official SDL website: https://wiki.libsdl.org/SDL2/Download
    2. Download the "Development Libraries" for Visual C++ (e.g., `SDL2-devel-2.x.x-vc.zip`).
    3. Extract the downloaded archive. Inside, you'll find a `bin` folder (e.g., `SDL2-2.x.x\x64\bin`).
    4. Copy `SDL2.dll` from this `bin` folder into your F# project's output directory. For development, this is typically `GameBoyEmulator\bin\Debug\net9.0`. For a production deployment, this DLL would be included in your application's distribution package.
- **macOS:**
    1. Install using Homebrew, a popular package manager for macOS: `bash brew install sdl2`
    2. Homebrew places `libSDL2.dylib` in standard library search paths like `/usr/local/lib` or `/opt/homebrew/lib`, making it discoverable by your application.

- **Linux (Ubuntu/Debian-based):**
    1. Update your package lists and install the development libraries for SDL2: `bash sudo apt update sudo apt install libsdl2-dev`
    2. This command installs `libSDL2.so` into system paths, ensuring it's available for your .NET application.

`⚡ Real-world insight:` Managing native dependencies like SDL2 for cross-platform deployment is a common challenge. In production, you'd typically use `dotnet publish` with runtime identifiers (RIDs) to create platform-specific executables, and potentially custom build scripts or packaging tools to bundle the correct native binaries for each target OS. This ensures users don't have to manually install SDL2.

b. Add `Vortice.SDL2` NuGet Package

Now, add the F# binding for SDL2 to your project.

1. **In your project directory, open your terminal and run:** `bash dotnet add package Vortice.SDL2` **Current Version (as of 2026-05-05):** The `dotnet add package` command will automatically fetch the latest stable version of `Vortice.SDL2`. If a specific version were required for compatibility, you could append `--version X.Y.Z`. As of this date, using the latest stable version is the best practice.

5. Basic SDL.NET Test Program

Let's write a minimal F# program to open a window, confirming SDL.NET is correctly configured and can interact with the native SDL2 libraries.

1. **Open `Program.fs`** in your `GameBoyEmulator` project.
2. **Replace its entire content** with the following F# code:

    ```fsharp module Program

    open System open System.Runtime.InteropServices // For Marshal.PtrToStringUTF8 open Vortice.SDL2

    /// Entry point for the application [] let main argv = printfn "Initializing SDL..."
    ```

```
// Initialize SDL with the video subsystem. SDL_INIT_VIDEO is essential
for window creation.
let initResult = SDL.SDL_Init(SDL.SDL_INIT_VIDEO)
if initResult < 0 then
    // If initialization fails, retrieve and print the SDL error message.
    let errorMsg = SDL.SDL_GetError() |> Marshal.PtrToStringUTF8
    fprintfn stderr "ERROR: SDL_Init failed: %s" errorMsg
    exit 1 // Exit with an error code

printfn "Creating window..."

// Define window properties
let windowTitle = "Game Boy Emulator Setup Test"
let windowWidth = 640
let windowHeight = 480
let windowFlags = SDL.SDL_WindowFlags.Resizable // Allow the window to be
resized

// Create an SDL window. SDL_WINDOWPOS_CENTERED places it in the middle
of the screen.
let window = SDL.SDL_CreateWindow(windowTitle,
                                  SDL.SDL_WINDOWPOS_CENTERED,
                                  SDL.SDL_WINDOWPOS_CENTERED,
                                  windowWidth,
                                  windowHeight,
                                  windowFlags)

if window = IntPtr.Zero then
    // If window creation fails, retrieve and print the SDL error
message.
    let errorMsg = SDL.SDL_GetError() |> Marshal.PtrToStringUTF8
    fprintfn stderr "ERROR: SDL_CreateWindow failed: %s" errorMsg
    SDL.SDL_Quit() // Clean up SDL resources before exiting
    exit 1

printfn "Window created successfully. Press any key in the console to
close it."

// Keep the window open until the user presses a key in the console.
Console.ReadKey() |> ignore

printfn "Destroying window and quitting SDL..."
SDL.SDL_DestroyWindow(window) // Release window resources
SDL.SDL_Quit() // Shut down all initialized SDL subsystems

0 // Return 0 to indicate successful execution
```

```

- `open Vortice.SDL2`: Imports the necessary SDL functions and types from the `Vortice.SDL2` NuGet package.
- `open System.Runtime.InteropServices`: Provides `Marshal.PtrToStringUTF8` for converting unmanaged C-style strings (returned by `SDL_GetError`) into managed F# strings.
```

- **`SDL.SDL_Init(SDL.SDL_INIT_VIDEO)`**: This crucial call initializes SDL's video subsystem. Without it, you cannot create windows or render graphics.
- **`SDL.SDL_CreateWindow(...)`**: Creates the actual operating system window. We specify its title, position (centered), dimensions, and `Resizable` flag.
- **Error Handling (`if initResult < 0` / `if window = IntPtr.Zero`):** Proper error checking is vital. SDL functions return negative values or `IntPtr.Zero` on failure. `SDL.SDL_GetError()` retrieves a human-readable error message.
- **`Console.ReadKey() |> ignore`**: This line pauses the console application, effectively keeping the SDL window open until a key press is detected. The `|> ignore` discards the return value of `ReadKey`.
- **`SDL.SDL_DestroyWindow(window)` and `SDL.SDL_Quit()`**: These functions are essential for releasing resources held by SDL and the operating system. Always clean up after yourself!

Testing & Verification

It's time to confirm everything is working as expected before moving on.

1. **Verify .NET SDK and F# Installation:**
    - Open your terminal or command prompt.
    - Run `dotnet --version`. You should see `9.0.x` (or a similar version of .NET 9.0).
    - Run `dotnet fsi`. This launches the F# interactive environment, confirming the F# compiler and runtime are correctly installed. Type `#q` to exit.
2. **Run the SDL Test Program:**
    - Navigate to your `GameBoyEmulator` project directory (where `GameBoyEmulator.fsproj` is located).
    - Execute the program using the .NET CLI: `bash dotnet run`

- **Expected Behavior:** * A new window titled "Game Boy Emulator Setup Test" should appear on your screen, centered. * The console output should display: "Initializing SDL...", "Creating window...", and "Window created successfully. Press any key in the console to close it."

- **To close the window:** Press any key in the console where `dotnet run` is executing. The window should close, and the console should then show "Destroying window and quitting SDL..." before the program exits.

If the window appears and closes cleanly, your F# development environment with SDL graphics is correctly set up!

Production Considerations

Even at this foundational stage, it's valuable to consider how these choices impact a production-ready system.

- **Cross-Platform Deployment:** The need to manually copy `SDL2.dll` (Windows) or rely on system-installed `libSDL2.dylib`/`libSDL2.so` (macOS/Linux) highlights a common challenge with native dependencies. For a shippable product, you'd integrate these native libraries directly into your application's distribution package. This often involves using `dotnet publish` with runtime identifiers (RIDs) (e.g., `dotnet publish -r win-x64 -c Release`) and potentially custom build scripts to bundle the correct native binaries for each target platform.
- **Dependency Management:** Using NuGet (`Vortice.SDL2`) is a robust and standard practice for managing external libraries. It ensures consistent versions across development environments and simplifies updates.
- **Performance Baseline:** While F# and .NET offer excellent performance, the abstraction layer of `Vortice.SDL2` introduces some minimal overhead compared to writing direct C calls. For an emulator, this overhead is typically negligible compared to the performance demands of the CPU and PPU emulation loops, which will be the primary bottlenecks we'll optimize later.

`🔥 Optimization / Pro tip:` For maximum control and to avoid runtime dependency issues, consider bundling platform-specific native SDL2 binaries directly into your project's output or using a tool like Costura.Fody (for Windows DLLs) to embed them into your executable.

Common Issues & Solutions

Here are a few pitfalls you might encounter during this setup phase and how to resolve them:

- **`ERROR: SDL_Init failed: No available video device` (or similar SDL errors):**

- **Issue:** The native SDL2 library (`SDL2.dll`, `libSDL2.dylib`, `libSDL2.so`) is either not found by your application or is incompatible (e.g., wrong architecture, corrupted file).
- **Solution:**
- **Windows:** Ensure `SDL2.dll` is correctly copied into your project's output directory (e.g., `GameBoyEmulator\bin\Debug\net9.0`). Double-check that its architecture (x64 vs. x86) matches your project's target architecture.
- **macOS/Linux:** Verify that SDL2 is correctly installed via Homebrew or your system's package manager and that its libraries (`libSDL2.dylib`/`libSDL2.so`) are in system-searchable paths. A reboot or re-login might be needed after installation on some Linux systems.
    - ⚠️ `What can go wrong:` Forgetting to install the native SDL2 library is the most common mistake. `Vortice.SDL2` is just a wrapper; it can't function without the underlying native code.
- **`error FS0078: This construct is not permitted in code that is compiled to a DLL.`**
- **Issue:** You've likely tried to place the `[<EntryPoint>]` function (which denotes the main execution point of an application) in a library project, not a console application.
- **Solution:** Verify that your `GameBoyEmulator.fsproj` file contains `<OutputType>Exe</OutputType>` within its `<PropertyGroup>`. The `dotnet new console` command should configure this correctly by default. If you created a different project type, you might need to manually edit the `.fsproj` file.
- **`dotnet` command not found:**
- **Issue:** The .NET SDK was not installed correctly, or its installation path is not included in your system's `PATH` environment variable.
- **Solution:** Re-run the .NET SDK installer. On some Linux distributions or after manual installation, you might need to restart your terminal or re-log in for `PATH` changes to take effect.

Summary & Next Step

Congratulations! You've successfully established your F# development environment and confirmed basic graphics functionality using SDL. You now have a robust foundation:

- The **.NET 9.0 SDK** and **F# 9.0** tooling are installed and verified.
- A **new F# console project** (`GameBoyEmulator`) is ready.
- The **`Vortice.SDL2` NuGet package** has been added to your project.
- The **native SDL2 libraries** are correctly installed and accessible on your system.
- A **working test program** opens an SDL window, confirming your graphics setup.

This solid environment prepares us for the real work ahead. In the next chapter, we will begin implementing the core of our emulator: the Game Boy's CPU. We'll dive into understanding its registers, flags, and the execution of its most fundamental opcodes.

🧠 Check Your Understanding

- What is the primary purpose of the `Vortice.SDL2` NuGet package, and why do you still need to install native SDL2 libraries separately?
- Why is a modular project structure, reflecting the Game Boy's hardware components, beneficial for this project?
- If you encounter an `SDL_Init failed` error, what are the first two things you would check, specifically for a Windows development environment?

⚡ Mini Task

- Modify the `Program.fs` test code to change the window's background color. To do this, you'll need to create an `SDL_Renderer`, use `SDL_SetRenderDrawColor` to set a color, and `SDL_RenderClear` to fill the window. Remember to `SDL_RenderPresent` to show the changes. This will deepen your understanding of basic SDL rendering.

🚀 Scenario

You've deployed your Game Boy emulator to a friend who uses a different operating system (e.g., you developed on Windows, they use Linux). They report that the emulator launches but crashes immediately when it tries to open a window, with an error message indicating a missing `libSDL2.so`. What are the most likely causes, and how would you guide them to troubleshoot this issue, considering the setup steps we just completed?

📌 TL;DR

- Set up F# development with .NET 9.0 SDK and your preferred editor (VS Code with Ionide recommended).
- Create a new F# console project (`dotnet new console -lang F#`).
- Install native SDL2 libraries specific to your OS (e.g., `SDL2.dll` on Windows, `brew install sdl2` on macOS, `libsdl2-dev` on Linux).
- Add the `Vortice.SDL2` NuGet package to your project (`dotnet add package Vortice.SDL2`).
- Verify the setup by running a simple F# program that opens and closes an SDL window.

🧠 Core Flow

1. Install .NET SDK (includes F# tooling).
2. Create new F# console project for the emulator.
3. Install native SDL2 libraries for the target operating system.
4. Add `Vortice.SDL2` NuGet package to the F# project.
5. Write and execute a basic SDL window test program to confirm functionality.

🚀 Key Takeaway

Establishing a stable, cross-platform development environment with appropriate tools and a clear modular structure is the non-negotiable first step for any complex system, especially one involving low-level hardware emulation.

References

- .NET Download: https://dotnet.microsoft.com/download
- F# Language Reference: https://learn.microsoft.com/en-us/dotnet/fsharp/
- SDL Documentation: https://wiki.libsdl.org/
- Vortice.SDL2 NuGet Package: https://www.nuget.org/packages/Vortice.SDL2
- Pan Docs (Game Boy Technical Reference): https://gbdev.io/pandocs/

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 02

The CPU Core: Registers, Flags, and Basic Instructions

Introduction

Welcome to the foundational stage of our Game Boy emulator! In this chapter, we're going to construct the very brain of our system: the Central Processing Unit (CPU). The Game Boy uses a custom 8-bit CPU, often referred to as the "SM83," which is a hybrid between a Zilog Z80 and Intel 8080. Understanding and accurately emulating its behavior is paramount to running any Game Boy software.

This milestone is critical because every single action within a Game Boy game—from moving a character to calculating damage—is ultimately a sequence of CPU instructions. Building the CPU core correctly is non-negotiable for a functional emulator. By the end of this chapter, you will have a functional, albeit minimal, CPU core capable of storing its state (registers and flags) and executing a few fundamental instructions. This forms the bedrock upon which we'll build memory access, graphics, and more complex logic.

Project Overview: Building a Game Boy Emulator

The overarching goal of this project is to build a functional Game Boy emulator from first principles using F#. This involves recreating the behavior of the original hardware components: the CPU, Memory Management Unit (MMU), Picture Processing Unit (PPU), Audio Processing Unit (APU), and input system. We'll start with the CPU, as it dictates the flow of execution for all other components.

Tech Stack Deep Dive

For this project, we're leveraging the following technologies:

- **F# (Version 8.0)**: Our primary programming language. F# is a functional-first language on the .NET platform, offering strong typing, immutability by default, and powerful abstractions that are well-suited for modeling complex hardware components and their interactions.

- **.NET SDK (Version 8.0)**: The runtime and development tools for F#. As of 2026-05-05, .NET 8.0 is the latest stable Long-Term Support (LTS) release, providing excellent performance and cross-platform capabilities.
- **SDL Bindings (e.g., Silk.NET.SDL, Veldrid)**: While not directly used in this chapter, a cross-platform graphics library will be essential for rendering the Game Boy's display later. We will explore modern .NET bindings for SDL (like `Silk.NET.SDL`) or other low-level graphics libraries when we reach the PPU implementation.

Milestones and Build Plan

This chapter focuses on getting the CPU's fundamental state management and instruction execution working. Here's our plan:

1. **Model CPU Registers and Flags**: Define F# records to represent the CPU's internal state.
2. **Implement Flag Management Helpers**: Create functions to precisely set, clear, and check individual flags within the `F` register. This is critical for accurate arithmetic and logic operations.
3. **Simulate Memory Access**: Introduce a basic `IMemory` interface and a `MockMemory` implementation. This allows us to test CPU instructions without needing a full Memory Management Unit yet.
4. **Implement Basic Instructions**: Code the `fetch-decode-execute` cycle for a handful of simple, representative Game Boy opcodes.
5. **Verify Execution**: Run a small test program to observe register and flag changes, confirming our CPU's initial correctness.

By the end of this chapter, you'll have a working CPU core that can interpret and execute basic instructions, laying the groundwork for more complex Game Boy emulation.

Planning & Design: Modeling the CPU State

Before we write any code, let's outline how we'll represent the CPU. The SM83 CPU has a set of registers, which are small, fast storage locations used for calculations and addressing memory. It also has a set of flags that indicate the results of operations (e.g., if a result was zero, or if an overflow occurred).

CPU Registers

The Game Boy CPU has the following primary registers:

- **General-Purpose 8-bit Registers:** `A`, `B`, `C`, `D`, `E`, `H`, `L`. These can also be paired for 16-bit operations: `BC`, `DE`, `HL`.
- **Accumulator (`A`):** The primary register for arithmetic and logic operations.
- **Flags Register (`F`):** An 8-bit register where individual bits represent specific flags.
- **Stack Pointer (`SP`):** A 16-bit register pointing to the current top of the stack in memory.
- **Program Counter (`PC`):** A 16-bit register pointing to the memory address of the next instruction to be executed.

CPU Flags (within the F register)

The `F` register is special. Only the upper four bits are used, representing four important flags. The lower four bits (0-3) are always zero.

- **Z (Zero Flag - Bit 7):** Set if the result of an operation is zero.
- **N (Subtract Flag - Bit 6):** Set if the last instruction was a subtraction.
- **H (Half Carry Flag - Bit 5):** Set if there was a carry from bit 3 to bit 4 (useful for BCD arithmetic).
- **C (Carry Flag - Bit 4):** Set if there was a carry from bit 7 or a borrow.

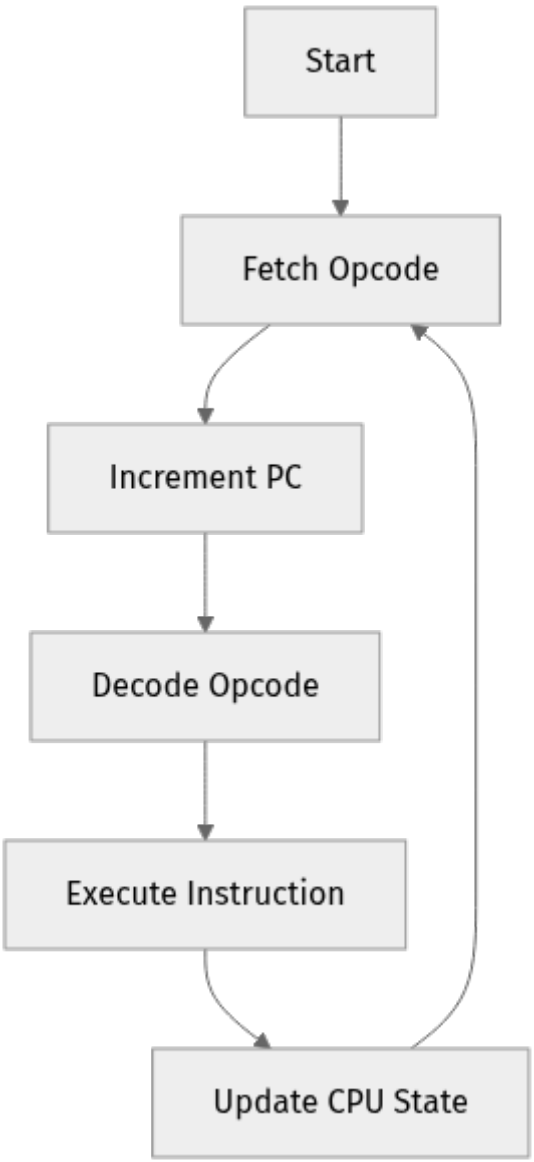
F# Representation

In F#, we'll model the CPU's state using immutable records. This aligns perfectly with functional programming principles, ensuring that state changes are explicit and predictable.

For registers, we'll create a `Registers` record. For the overall CPU state, we'll have a `CpuState` record that includes `Registers` and a placeholder for memory.

CPU Core Data Flow

The basic operation of our CPU will follow a classic fetch-decode-execute cycle:



The CPU constantly repeats this cycle, processing one instruction at a time. This continuous loop is the heart of any emulator.

File Structure

We'll start with a single `Cpu.fs` file to encapsulate all CPU-related logic for now. As the project grows, we might split it further (e.g., `Registers.fs`, `Opcodes.fs`).

```
GameBoyEmulator/
├── GameBoyEmulator.fsproj
├── Program.fs
└── Cpu.fs
```

Step-by-Step Implementation

Let's start by defining our CPU's core data structures.

1. Define Registers and CPU State

Open your `GameBoyEmulator.fsproj` file and ensure it targets .NET 8.0.

```
<!-- GameBoyEmulator.fsproj -->
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <WarnOn>FS3391;FS3392</WarnOn>
  </PropertyGroup>

  <ItemGroup>
    <Compile Include="Cpu.fs" />
    <Compile Include="Program.fs" />
  </ItemGroup>

</Project>
```

Explanation: This standard `.fsproj` setup targets .NET 8.0, which is the latest stable long-term support (LTS) release as of 2026-05-05. The `<WarnOn>` element helps catch common F# code quality issues, promoting better F# coding practices.

Next, create a new file named `Cpu.fs` in your project root.

```
// Cpu.fs

module GameBoyEmulator.Cpu

/// Represents the 8-bit general-purpose registers and the 16-bit program
counter (PC)
/// and stack pointer (SP) of the Game Boy's SM83 CPU.
/// The 'F' register holds the CPU flags.
type Registers = {
    A : byte
    F : byte // Flags register (Z N H C 0 0 0 0)
    B : byte
    C : byte
    D : byte
    E : byte
    H : byte
    L : byte
    SP : uint16 // Stack Pointer
    PC : uint16 // Program Counter
}

/// Represents the full state of the Game Boy CPU.
/// This will be expanded to include memory and other components later.
type CpuState = {
    Registers : Registers
    // Cycles : int64 // We'll add cycle counting later
    // Memory : byte array // Placeholder for memory, will be replaced by MMU
}

/// Initializes the CPU registers to their power-up state.
/// This is based on common Game Boy boot ROM behavior.
let createInitialRegisters () : Registers =
    {
        A = 0x01uy // Often 0x01 for DMG, or 0x11 for CGB
        F =
0xA0uy // ZNHC flags: Z=1, N=0, H=1, C=0 (0b10100000). Lower 4 bits are always
0.
        B = 0x00uy
        C = 0x13uy
        D = 0x00uy
        E = 0xD8uy
        H = 0x01uy
        L = 0x4Duy
        SP = 0xFFFEus // Stack pointer typically starts high
        PC = 0x0100us // PC usually starts after the boot ROM
    }

/// Initializes the CPU state.
let createInitialCpuState () : CpuState =
    {
        Registers = createInitialRegisters ()
    }
```

Explanation:

- `module GameBoyEmulator.Cpu`: Defines a module to organize our CPU-related code. Modules help prevent name collisions and improve code organization within larger projects.

- `type Registers = { ... }`: This F# record defines the structure for our CPU registers. Each field is immutable by default, which is a core F# principle. `byte` is used for 8-bit registers, and `uint16` for 16-bit ones. Using `uint16` for `PC` and `SP` is important as Game Boy addresses are 16-bit.
- `type CpuState = { ... }`: This record will hold the entire CPU state. For now, it just contains `Registers`. We've added comments for `Cycles` and `Memory` as a reminder of future additions, demonstrating forward-thinking design.
- `let createInitialRegisters ()`: This function provides the initial power-up values for the Game Boy's CPU registers. These are specific values that the hardware starts with, crucial for compatibility with the boot ROM. We use `uy` and `us` suffixes to denote unsigned byte and unsigned short literals, respectively. The `F` register is set to `0xA0uy` (10100000 in binary), which means the Zero (Z) flag is set, Half-Carry (H) is set, and Subtract (N) and Carry (C) flags are cleared. The lower 4 bits are always 0.
- `let createInitialCpuState ()`: A simple function to create an initial `CpuState` by calling `createInitialRegisters`.

📌 **Key Idea:** Using immutable records for `Registers` and `CpuState` means that every operation that changes the CPU's state will return a new `CpuState` instance, rather than modifying an existing one. This makes state transitions explicit and easier to reason about.

2. Flag Management

Manipulating individual bits within the `F` register requires bitwise operations. Let's define helper functions for setting, clearing, and checking flags. By encapsulating these operations, we make our code cleaner and less error-prone.

Add the following functions to `Cpu.fs` after `createInitialCpuState`.

```
// Cpu.fs (continued)

// Flag masks for the F register
let FlagZ = 0x80uy // Zero Flag (Bit 7)
let FlagN = 0x40uy // Subtract Flag (Bit 6)
let FlagH = 0x20uy // Half Carry Flag (Bit 5)
let FlagC = 0x10uy // Carry Flag (Bit 4)

/// Sets a specific flag in the F register.
let setFlag flag (registers: Registers) : Registers =
    { registers with F = registers.F ||| flag }

/// Clears a specific flag in the F register.
let clearFlag flag (registers: Registers) : Registers =
    { registers with F = registers.F &&& (~~~flag) }

/// Checks if a specific flag is set in the F register.
let isFlagSet flag (registers: Registers) : bool =
    (registers.F &&& flag) = flag

/// Updates the Z flag based on a value.
let updateZFlag value (registers: Registers) : Registers =
    if value = 0x00uy then setFlag FlagZ registers
    else clearFlag FlagZ registers

/// Updates the N flag based on whether the last operation was a subtraction.
let updateNFlag isSubtraction (registers: Registers) : Registers =
    if isSubtraction then setFlag FlagN registers
    else clearFlag FlagN registers

/// Updates the H flag based on an 8-bit addition or subtraction.
/// For addition, checks if bit 3 carried to bit 4.
/// For subtraction, checks if bit 3 borrowed from bit 4.
let updateHFlag8bit isSubtraction val1 val2 (registers: Registers) : Registers
=
    let halfCarry =
        if isSubtraction then
            // Half borrow check: if lower nibble of val1 is less than lower
nibble of val2
            ((val1 &&& 0x0Fu) < (val2 &&& 0x0Fu))
        else
            // Half carry check: if sum of lower nibbles overflows into bit 4
            (((val1 &&& 0x0Fu) + (val2 &&& 0x0Fu)) &&& 0x10u) = 0x10u
    if halfCarry then setFlag FlagH registers
    else clearFlag FlagH registers

/// Updates the C flag based on an 8-bit addition or subtraction.
/// For addition, checks if bit 7 carried out.
/// For subtraction, checks if there was a borrow.
let updateCFlag8bit isSubtraction val1 val2 (registers: Registers) : Registers
=
    let carry =
        if isSubtraction then
            // Borrow check: if val1 is less than val2
            (val1 < val2)
        else
            // Carry check: if adding val2 to val1 results in overflow
            (uint16 val1 + uint16 val2) > 0xFFu
    if carry then setFlag FlagC registers
    else clearFlag FlagC registers
```

Explanation:

- **Flag Masks**: We define `FlagZ`, `FlagN`, `FlagH`, `FlagC` as `byte` constants. These are bitmasks to isolate or manipulate the specific flag bits within the `F` register.
- **`setFlag` / `clearFlag`**: These functions use bitwise OR (`|||`) to set a bit and bitwise AND (`&&&`) with the bitwise NOT (`~~~`) of the flag to clear a bit. They return a new `Registers` record with the updated `F` field, preserving immutability.
- **`isFlagSet`**: Checks if a flag is currently set.
- **`updateZFlag`, `updateNFlag`, `updateHFlag8bit`, `updateCFlag8bit`**: These are crucial for correctly updating flags after arithmetic operations. The logic for half-carry and carry flags in 8-bit operations is specific to the SM83 and needs careful implementation based on documentation.
    - `isSubtraction` parameter is vital as flag logic often differs between addition and subtraction.
    - `updateHFlag8bit` checks for carry/borrow between the lower and upper nibble (bits 3 and 4).
    - `updateCFlag8bit` checks for overflow/borrow for the full 8-bit value.

🧠 **Important:** The exact logic for flag updates can be tricky and is a common source of bugs in emulators. Always consult detailed SM83 documentation (like Pan Docs) for precise behavior. The Game Boy CPU's flag behavior is not always intuitive and differs from other CPUs like the Z80 in subtle ways. Incorrect flag emulation can lead to subtle bugs in game logic that are extremely hard to debug later.

⚡ **Real-world insight:** The `|||` (bitwise OR), `&&&` (bitwise AND), and `~~~` (bitwise NOT) operators are fundamental for low-level system programming. Understanding how to manipulate individual bits efficiently is a core skill for emulator development.

3. Basic Instruction Execution

Now, let's implement the `execute` function for a few simple instructions. For this chapter, we'll hardcode a few opcodes directly. Later, we'll read them from memory via a proper Memory Management Unit (MMU).

First, we need a way to simulate reading from memory for our CPU. For now, we'll use a very basic mock.

Add the following type and function to `Cpu.fs` (after the flag functions).

```
// Cpu.fs (continued)

/// A simple type to represent our memory interface.
/// In Chapter 3, this will be replaced by a full Memory Management Unit (MMU).
type IMemory =
    abstract member ReadByte : uint16 -> byte
    abstract member WriteByte : uint16 -> byte -> unit

/// A mock memory implementation for initial CPU testing.
/// It's just a byte array for now, representing 64KB of addressable space.
type MockMemory (size : int) =
    let data = Array.zeroCreate<byte> size

    interface IMemory with
        member _.ReadByte addr =
            if addr < uint16 data.Length then
                data[int addr]
            else
                0xFFuy // Default for unmapped memory reads on Game Boy is 0xFF

        member _.WriteByte addr value =
            if addr < uint16 data.Length then
                data[int addr] <- value
            // else ignore write to unmapped memory for now, a real MMU would
handle this

/// The main CPU execution function.
/// It takes the current CPU state and a memory interface,
/// fetches an opcode, decodes it, executes it, and returns the new state.
let executeCycle (cpuState: CpuState) (memory: IMemory) : CpuState =
    let currentRegisters = cpuState.Registers
    let pc = currentRegisters.PC
    let opcode = memory.ReadByte pc // Fetch the opcode at the current Program
Counter

    // Increment PC for the next instruction. Most instructions are 1 byte.
    // Multi-byte instructions will increment PC further during their
execution.
    let registersAfterFetch = { currentRegisters with PC = pc + 1us }

    // Decode and Execute
    match opcode with
    | 0x00uy -> // NOP: No Operation
        // Pan Docs: "No operation. Does nothing."
        // Cycles: 4
        // Flags: None affected
        { cpuState with Registers = registersAfterFetch }

    | 0x06uy -> // LD B, n: Load 8-bit immediate into B
        // Pan Docs: "Load 8-bit immediate value n into register B."
        // Cycles: 8
        // Flags: None affected
        let value = memory.ReadByte (registersAfterFetch.PC) // Read the
immediate value
        let newRegisters = { registersAfterFetch with B = value; PC = registers
AfterFetch.PC + 1us } // Increment PC past the immediate
        { cpuState with Registers = newRegisters }

    | 0x0Euy -> // LD C, n: Load 8-bit immediate into C
        // Pan Docs: "Load 8-bit immediate value n into register C."
        // Cycles: 8
```

```
        // Flags: None affected
        let value = memory.ReadByte (registersAfterFetch.PC)
        let newRegisters = { registersAfterFetch with C = value; PC = registers
AfterFetch.PC + 1us }
        { cpuState with Registers = newRegisters }

    | 0x04uy -> // INC B: Increment register B
        // Pan Docs: "Increment register B. Flags: Z 0 H -"
        // Cycles: 4
        let oldValue = registersAfterFetch.B
        let newValue = oldValue + 1uy
        let newRegisters =
            registersAfterFetch
            |> (updateZFlag newValue) // Update Z flag based on newValue
            |> (updateNFlag false)    // N flag is always cleared for INC
            |> (updateHFlag8bit false oldValue 1uy) // Check half carry for
addition (increment by 1)
            |> (fun r -> { r with B = newValue }) // Update B register
        { cpuState with Registers = newRegisters }

    | 0x05uy -> // DEC B: Decrement register B
        // Pan Docs: "Decrement register B. Flags: Z 1 H -"
        // Cycles: 4
        let oldValue = registersAfterFetch.B
        let newValue = oldValue - 1uy
        let newRegisters =
            registersAfterFetch
            |> (updateZFlag newValue) // Update Z flag based on newValue
            |> (updateNFlag true)     // N flag is always set for DEC
            |> (updateHFlag8bit true oldValue 1uy) // Check half carry for
subtraction (decrement by 1)
            |> (fun r -> { r with B = newValue }) // Update B register
        { cpuState with Registers = newRegisters }

    // Default case for unimplemented opcodes
    | _ ->
        // For now, we'll just throw an exception for unknown opcodes.
        // Later, we'll implement a full opcode map or a more robust error
handling mechanism.
        failwithf "Unimplemented opcode: 0x%02X at PC: 0x%04X" opcode pc
```

Explanation:

- **`IMemory` and `MockMemory`**: We introduce a simple `IMemory` interface and a `MockMemory` implementation. This allows us to pass a memory abstraction to our CPU, decoupling it from the actual memory management unit (which we'll build in the next chapter). For now, `MockMemory` is just a byte array representing the Game Boy's 64KB address space. `ReadByte` returns `0xFF` for out-of-bounds reads, which is common Game Boy behavior. This is a classic example of the Dependency Inversion Principle, making our code more modular and testable.

- **`executeCycle` function**: This is the core of our CPU.
  - It takes `cpuState` and `memory` as input and returns a new `CpuState`. This functional approach ensures state changes are explicit.
- **Fetch**: `memory.ReadByte pc` fetches the opcode at the current `PC`.
- **Increment PC**: `registersAfterFetch` updates the `PC` to point to the next byte. Most instructions are 1 byte, but `LD r, n` (load immediate) is 2 bytes, so `PC` needs an additional increment within the `match` arm.
- **Decode & Execute**: The `match opcode with` statement handles different opcodes.
- **`NOP` (0x00)**: Does nothing except advance `PC`.
- **`LD B, n` (0x06) / `LD C, n` (0x0E)**: These are 2-byte instructions. The first byte is the opcode, the second is the immediate value `n`. We read `n` from `PC + 1` and then increment `PC` again to move past `n`.
- **`INC B` (0x04)**: Increments register `B`. Crucially, it updates the `Z`, `N`, and `H` flags. The `C` flag is not affected by 8-bit `INC`/`DEC` operations, which is an important detail for the SM83. We use the pipe-forward operator `|>` to chain flag updates, making the sequence of operations clear and readable.
- **`DEC B` (0x05)**: Decrements register `B`. Similar to `INC`, it updates `Z`, `N`, and `H` flags, but `N` is set for decrement.
- **Return New State**: Each branch of the `match` returns a new `CpuState` record, reflecting the changes made by the instruction.
- **`failwithf`**: For any unimplemented opcode, we'll throw an exception. This helps us identify missing instructions during development.

⚡ **Quick Note:** The cycle counts (e.g., `Cycles: 4`) mentioned in the comments are for the Game Boy's internal clock (M-cycles). We'll use these later for proper timing and synchronization with the PPU and APU. For now, they are informational but critical for future performance modeling.

⚠️ **What can go wrong:** Forgetting to increment `PC` correctly for multi-byte instructions is a very common bug. If `PC` isn't advanced past the immediate value in `LD r, n`, the CPU will try to interpret the data byte as an opcode, leading to immediate crashes or incorrect execution.

4. Integrate into Program.fs

Let's modify our `Program.fs` to create a CPU and execute a few cycles using our `MockMemory`.

```
// Program.fs

open System
open GameBoyEmulator.Cpu

[<EntryPoint>]
let main argv =
    printfn "Starting Game Boy Emulator CPU Test..."

    // Create a mock memory of 64KB (typical Game Boy memory space)
    let memory = MockMemory(0x10000) :> IMemory

    // Load a simple test program into mock memory
    // This sequence means:
    // 0x0100: NOP
    // 0x0101: LD B, 0x0A (Load 0x0A into B)
    // 0x0103: INC B (B becomes 0x0B)
    // 0x0104: DEC B (B becomes 0x0A)
    // 0x0105: NOP
    let program = [| 0x00uy; 0x06uy; 0x0Auy; 0x04uy; 0x05uy; 0x00uy |]
    for i = 0 to program.Length - 1 do
        (memory :?> MockMemory).data[0x0100 + i] <- program[i] // Directly
access internal array for mock memory

    // Initialize CPU state
    let mutable cpuState = createInitialCpuState ()
    printfn "Initial PC: 0x%04X, B: 0x%02X, F: 0x%02X" cpuState.Registers.PC cp
uState.Registers.B cpuState.Registers.F

    // Execute a few cycles
    for i = 1 to 5 do // Execute 5 instructions
        printfn "\n--- Cycle %d ---" i
        let oldPc = cpuState.Registers.PC
        let opcode = memory.ReadByte oldPc
        printfn "Executing opcode 0x%02X at PC 0x%04X" opcode oldPc

        cpuState <- executeCycle cpuState memory
        printfn "New PC: 0x%04X, B: 0x%02X, F: 0x%02X" cpuState.Registers.PC cp
uState.Registers.B cpuState.Registers.F

        // Check flags after relevant operations
        if opcode = 0x04uy || opcode = 0x05uy then
            printfn "  Flags: Z=%b N=%b H=%b C=%b"
                (isFlagSet FlagZ cpuState.Registers)
                (isFlagSet FlagN cpuState.Registers)
                (isFlagSet FlagH cpuState.Registers)
                (isFlagSet FlagC cpuState.Registers)

    0 // Return an exit code
```

Explanation:

- `open GameBoyEmulator.Cpu`: Imports our CPU module, making its functions and types available.
- `MockMemory`: We instantiate our `MockMemory` with 64KB, which is the full 16-bit addressable range of the Game Boy.

- `program`: A simple array of bytes representing a short sequence of Game Boy opcodes. We manually load this into our `MockMemory` starting at address `0x0100`, which is where the CPU's `PC` typically starts after the boot ROM. This simulates a very basic game program.
- `mutable cpuState`: We declare `cpuState` as `mutable` because our `executeCycle` function returns a new state, and we need to reassign it in the loop. This is a common pattern when combining functional updates with iterative processes in F#.
- **Execution Loop**: We loop a few times, calling `executeCycle` and printing the CPU state (`PC`, `B`, `F` registers, and flags) after each instruction. This allows us to observe the CPU's behavior and verify its correctness.

Testing & Verification

Now, let's run our program and verify the CPU's behavior.

1. **Build and Run**: Open your terminal in the `GameBoyEmulator` directory and run: `bash dotnet run`
2. **Expected Output and Verification**: You should see output similar to this:

   ```text Starting Game Boy Emulator CPU Test... Initial PC: 0x0100, B: 0x00, F: 0xA0

   --- Cycle 1 --- Executing opcode 0x00 at PC 0x0100 New PC: 0x0101, B: 0x00, F: 0xA0

   --- Cycle 2 --- Executing opcode 0x06 at PC 0x0101 New PC: 0x0103, B: 0x0A, F: 0xA0

   --- Cycle 3 --- Executing opcode 0x04 at PC 0x0103 New PC: 0x0104, B: 0x0B, F: 0x00 Flags: Z=False N=False H=False C=False

   --- Cycle 4 --- Executing opcode 0x05 at PC 0x0104 New PC: 0x0105, B: 0x0A, F: 0x40 Flags: Z=False N=True H=False C=False

   --- Cycle 5 --- Executing opcode 0x00 at PC 0x0105 New PC: 0x0106, B: 0x0A, F: 0x40 ```

   **Key points to verify:**

- **PC Increment**: Does `PC` correctly increment after each instruction (by 1 for `NOP`, by 2 for `LD r, n`)?
- `LD B, n`: Does `B` register correctly load `0x0A`?
   ```

- **`INC B` (0x04)**: `B` increments from `0x0A` to `0x0B`. * `Z` flag: `0x0B` is not zero, so `Z` is cleared (from initial `1`). * `N` flag: `INC` clears `N`. * `H` flag: `0x0A + 1` (0b1010 + 0b0001) does not cause a half-carry (no carry from bit 3 to bit 4), so `H` is cleared (from initial `1`). * `C` flag: `INC` does not affect `C`, so it remains cleared (from initial `0`). * Resulting `F` should be `0b00000000` = `0x00`.
- **`DEC B` (0x05)**: `B` decrements from `0x0B` to `0x0A`. * `Z` flag: `0x0A` is not zero, so `Z` is cleared. * `N` flag: `DEC` sets `N`. * `H` flag: `0x0B - 1` (0b1011 - 0b0001) does not cause a half-borrow (no borrow from bit 4 to bit 3), so `H` is cleared. * `C` flag: `DEC` does not affect `C`, so it remains cleared. * Resulting `F` should be `0b01000000` = `0x40`.

  This verification step is crucial. If your output differs, carefully re-check your `executeCycle` implementation and flag logic against the Game Boy's CPU documentation.

Operations & Performance Considerations

- **Performance**: The `executeCycle` function will be called millions of times per second (Game Boy CPU runs at ~4.19 MHz). While F# records are immutable, creating new records on every state change can introduce overhead. For now, this is acceptable for clarity and correctness. Later, in performance-critical sections (like the main emulation loop), we might explore using `mutable` fields or `ref` cells judiciously, or optimizing specific instruction paths using techniques like `inline` or highly optimized loops. The goal is accurate emulation first, then optimization.
- **Maintainability**: By encapsulating CPU logic in `Cpu.fs` and using a clear `Registers` record, we ensure a modular and understandable codebase. The `IMemory` interface is a great example of dependency inversion, making our CPU testable without a full MMU. This separation of concerns is a key principle in building complex systems.
- **Testing**: Unit tests are absolutely crucial for CPU emulation. Every opcode, every flag update needs to be verified. The small `program` array we used is a manual form of an integration test. Later, we'll use dedicated test ROMs like Blargg's CPU instruction tests, which are specifically designed to validate CPU behavior against known correct hardware behavior.

🔥 **Optimization / Pro tip:** For very performance-sensitive parts of an emulator, you might consider using mutable arrays or `ref` cells for memory and registers directly, even in F#. The key is to isolate these mutable parts and manage them

carefully, perhaps within a single `Emulator` record that wraps all mutable components. This balances functional purity with raw speed.

Common Issues & Solutions

- **Incorrect Flag Updates**: This is by far the most common pitfall in emulator development. The `H` (Half Carry) flag logic, in particular, can be tricky as it depends on operations on the lower nibble and differs for addition and subtraction.
- **Solution**: Create dedicated unit tests for flag calculations. Isolate flag logic into pure functions as we did. Consult authoritative documentation (like Pan Docs) rigorously.
- **PC Increment Errors**: For multi-byte instructions, forgetting to increment `PC` by the correct amount will lead to reading incorrect opcodes or data, causing the emulator to quickly desynchronize.
- **Solution**: Carefully read the instruction's byte length from documentation. For each `match` branch, ensure `PC` is advanced correctly, accounting for the opcode itself and any immediate operands.
- **Misinterpreting Documentation**: The SM83 is not a standard CPU, and its documentation is often reverse-engineered. Minor details can have significant impacts.
- **Solution**: Cross-reference multiple sources (Pan Docs, other emulator implementations) and use test ROMs as the ultimate source of truth for behavior. Emulator development is an iterative process of reading docs, implementing, testing, and debugging.

🧠 Check Your Understanding

- What is the purpose of the `F` register, and why are only its upper four bits significant?
- Explain the concept of immutability in F# records and how it applies to our `CpuState` updates.
- Why did we introduce an `IMemory` interface even though we're using a simple `MockMemory` for now?

⚡ Mini Task

- Implement the `LD B, A` (opcode `0x47`) instruction in `executeCycle`. This instruction copies the value from register `A` to register `B`. It does not affect any flags and is a 1-byte instruction.

🚀 Scenario

You've implemented the `ADD A, n` (opcode `0xC6`) instruction, which takes an 8-bit immediate `n` and adds it to register `A`. This instruction affects Z, N, H, and C flags. Describe how you would update `executeCycle` for this, specifically focusing on how you would use the `updateZFlag`, `updateNFlag` (which should be cleared), `updateHFlag8bit`, and `updateCFlag8bit` functions. Consider the number of bytes this instruction consumes.

📌 TL;DR

- The Game Boy's SM83 CPU uses 8-bit registers (A, B, C, D, E, H, L) and 16-bit registers (SP, PC).
- The `F` register stores four critical flags: Zero (Z), Subtract (N), Half Carry (H), and Carry (C). The lower 4 bits are unused.
- We model CPU state using immutable F# records for clarity and functional purity, returning new state with each instruction.
- CPU operation follows a fetch-decode-execute cycle, updating the `CpuState` with new register values.
- Precise flag updates and correct `PC` increments are crucial for accurate emulation.

🧠 Core Flow

1. Define `Registers` and `CpuState` records in `Cpu.fs` to model CPU state.
2. Implement helper functions for managing individual CPU flags using bitwise operations.
3. Create an `IMemory` interface and a `MockMemory` for basic, decoupled memory access.

4. Implement `executeCycle` with a `match` statement for basic opcodes (`NOP`, `LD r, n`, `INC r`, `DEC r`), ensuring correct flag updates and `PC` advancement.
5. Integrate and test the CPU in `Program.fs` by loading a small program into `MockMemory` and stepping through execution, observing register and flag changes.

🚀 Key Takeaway

Building an emulator starts with a meticulously crafted CPU core. Every instruction's effect on registers, program counter, and especially flags, must be precisely emulated. F#'s strong typing and immutability help manage this complex state with confidence, but requires careful attention to detail and thorough verification against hardware documentation.

References

- F# Language Reference: https://learn.microsoft.com/en-us/dotnet/fsharp/
- .NET Documentation: https://learn.microsoft.com/en-us/dotnet/
- Pan Docs (Game Boy Technical Reference): https://gbdev.io/pandocs/
- Game Boy CPU Instruction Set (gbdev.io): https://gbdev.io/pandocs/CPU_Instruction_Set.html

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 03

Memory Management Unit (MMU) and Basic Memory Access

The CPU you started building in the last chapter is blind without memory. It can execute instructions, but it can't load programs, store data, or interact with any of the Game Boy's peripherals like the screen or sound chip. This is where the Memory Management Unit (MMU) comes in.

This chapter guides you through creating the Game Boy's core memory system, the **Memory Management Unit (MMU)**. You'll learn about the Game Boy's memory map, how to model different memory regions, and implement the fundamental `readByte` and `writeByte` operations crucial for any emulator. By the end, your emulator will be able to load a Game Boy ROM into its virtual memory, a significant step towards running actual games.

Introduction

In any computer system, the CPU doesn't directly access every component. Instead, it interacts with a unified address space, and a component called the MMU translates those addresses to the correct physical memory location or hardware register. For the Game Boy, this system is relatively simple but absolutely critical.

This milestone focuses on building a functional `Memory` module that encapsulates the Game Boy's 64KB (0x0000-0xFFFF) address space. You'll define the different memory regions, implement logic to handle reads and writes to these regions, and crucially, enable loading a cartridge ROM into the system.

By the end of this chapter, your emulator will: * Have a structured representation of the Game Boy's memory. * Be able to load a Game Boy ROM file into the designated memory region. * Support basic `readByte` and `writeByte` operations across various memory areas, allowing the CPU to fetch instructions and interact with RAM.

Planning & Design

The Game Boy's CPU (a custom Sharp SM83) has a 16-bit address bus, meaning it can address 2^16 = 65,536 bytes (64KB) of memory. This 64KB is divided into distinct regions, each serving a specific purpose:

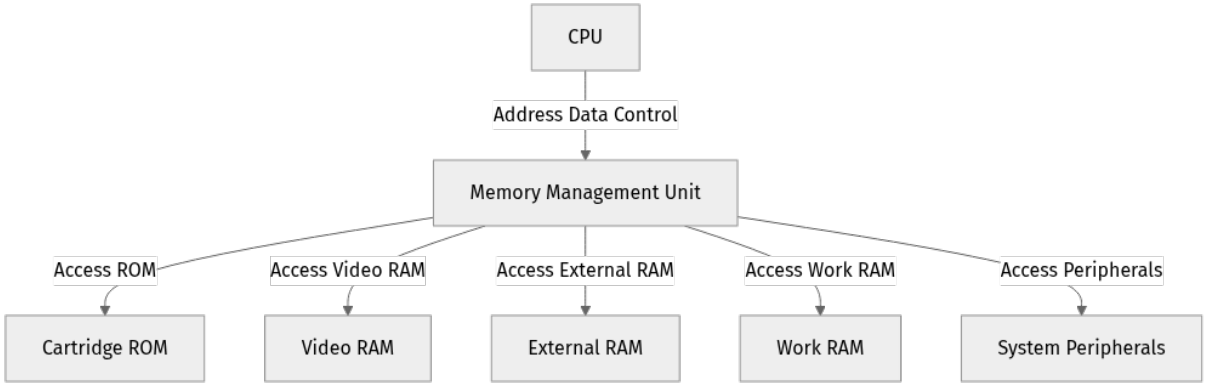
Game Boy Memory Map (Simplified Overview):

Address Range	Size	Description	Access
`0x0000` - `0x7FFF`	32KB	**Cartridge ROM** (Bank 0 and Switchable Banks)	Read Only
`0x8000` - `0x9FFF`	8KB	**VRAM** (Video RAM)	Read/ Write
`0xA000` - `0xBFFF`	8KB	**External RAM** (Cartridge RAM, if present)	Read/ Write
`0xC000` - `0xDFFF`	8KB	**WRAM** (Work RAM)	Read/ Write
`0xE000` - `0xFDFF`	7.5KB	**Echo RAM** (Mirror of `0xC000`-`0xDFFF`)	Read/ Write
`0xFE00` - `0xFE9F`	160 bytes	**OAM** (Object Attribute Memory, for sprites)	Read/ Write
`0xFEA0` - `0xFEFF`	96 bytes	**Unusable/Forbidden**	No Access
`0xFF00` - `0xFF7F`	128 bytes	**I/O Registers** (LCD, Joypad, Sound, Timers)	Read/ Write
`0xFF80` - `0xFFFE`	127 bytes	**HRAM** (High RAM)	Read/ Write
`0xFFFF`	1 byte	**Interrupt Enable Register**	Read/ Write

📌 **Key Idea:** The MMU's job is to act as a switchboard. When the CPU asks for data at address `0x1000`, the MMU knows to fetch it from the cartridge ROM. If it asks for `0xD000`, it goes to WRAM.

Designing the `Memory` Module

We'll model the MMU as an F# record type holding byte arrays for each distinct, addressable memory region. This approach offers clear separation and allows us to use F#'s powerful pattern matching for efficient address decoding.



File Structure

We'll create a new file, `Mmu.fs`, to house our memory logic. We might also create a `MemoryMap.fs` for constants, though for now, we can keep constants within `Mmu.fs` for simplicity.

```
GameboyEmulator/
├── src/
│   ├── GameboyEmulator/
│   │   ├── Cpu.fs
│   │   ├── Mmu.fs          <-- New file for Memory Management Unit
│   │   ├── Program.fs
│   │   └── GameboyEmulator.fsproj
```

Step-by-Step Implementation

1. Define Memory Constants

First, let's define the memory region boundaries as constants. These will be crucial for dispatching read/write operations.

Create a new file `src/GameboyEmulator/Mmu.fs` and add the following:

```
// src/GameboyEmulator/Mmu.fs

module Mmu

/// Defines the Game Boy's memory map boundaries and sizes.
module MemoryMap =
    let RomStart      = 0x0000u
    let RomEnd        = 0x7FFFu // 32KB
    let VramStart     = 0x8000u
    let VramEnd       = 0x9FFFu // 8KB
    let ExtRamStart   = 0xA000u
    let ExtRamEnd     = 0xBFFFu // 8KB
    let WramStart     = 0xC000u
    let WramEnd       = 0xDFFFu // 8KB
    let EchoRamStart  = 0xE000u
    let EchoRamEnd    = 0xFDFFu // 7.5KB - mirror of WRAM
    let OamStart      = 0xFE00u
    let OamEnd        = 0xFE9Fu // 160 bytes
    let UnusableStart = 0xFEA0u
    let UnusableEnd   = 0xFEFFu // 96 bytes - should not be accessed
    let IoStart       = 0xFF00u
    let IoEnd         = 0xFF7Fu // 128 bytes
    let HramStart     = 0xFF80u
    let HramEnd       = 0xFFFEu // 127 bytes
    let IeRegister    = 0xFFFFu // 1 byte

    let WramSize      = (WramEnd - WramStart + 1u) |> int
    let VramSize      = (VramEnd - VramStart + 1u) |> int
    let OamSize       = (OamEnd - OamStart + 1u) |> int
    let IoSize        = (IoEnd - IoStart + 1u) |> int
    let HramSize      = (HramEnd - HramStart + 1u) |> int
    // Note: ROM size varies based on cartridge. We'll handle a default 32KB
for now.
    // External RAM also varies. We'll reserve 8KB for now.
    let ExtRamSize    = (ExtRamEnd - ExtRamStart + 1u) |> int
```

Explanation: * `module Mmu` and `module MemoryMap`: We're using nested modules to organize our code, keeping memory-related constants within `MemoryMap` for clarity. * `u` suffix: Denotes an unsigned 16-bit integer (`uint16`), which is appropriate for Game Boy addresses. * `int` conversion: Used when calculating array sizes, as F# arrays expect `int` for their length.

2. Define the `Memory` Record

Now, let's define the `Memory` record type. This record will hold the byte arrays representing our different memory regions.

Append this to `src/GameboyEmulator/Mmu.fs`:

```
// src/GameboyEmulator/Mmu.fs (continued)

/// Represents the Game Boy's entire memory space.
type Memory = {
    rom       : byte array // Cartridge ROM (first 32KB, more with MBCs)
    vram      : byte array // Video RAM
    extRam    : byte array // External RAM (on cartridge)
    wram      : byte array // Work RAM
    oam       : byte array // Object Attribute Memory
    io        : byte array // I/O Registers
    hram      : byte array // High RAM
    ie        : byte       // Interrupt Enable Register
    // Add other components as they are implemented
}
```

Explanation: * `type Memory = { ... }`: This F# record defines the state of our MMU. Each field is a `byte array` for memory regions or a single `byte` for specific registers. * `rom`: This will initially hold the first 32KB of the cartridge. Later, we'll introduce Memory Bank Controllers (MBCs) to swap in larger ROMs. * `ie`: This is a single byte register, so it's directly represented as `byte`.

3. Initialize Memory

We need a way to create an initial `Memory` state, typically filled with default values (like `0xFF` for uninitialized memory, or `0x00`).

Append this to `src/GameboyEmulator/Mmu.fs`:

```
// src/GameboyEmulator/Mmu.fs (continued)

/// Creates an initial Memory state, typically with all bytes set to 0xFF.
let init () : Memory =
    {
        rom       = Array.create (MemoryMap.RomEnd - MemoryMap.RomStart + 1u |
> int) 0x00uy // Default to 0x00 for ROM, will be overwritten
        vram      = Array.create MemoryMap.VramSize 0xFFuy
        extRam    = Array.create MemoryMap.ExtRamSize 0xFFuy
        wram      = Array.create MemoryMap.WramSize 0xFFuy
        oam       = Array.create MemoryMap.OamSize 0xFFuy
        io        = Array.create MemoryMap.IoSize 0xFFuy
        hram      = Array.create MemoryMap.HramSize 0xFFuy
        ie        = 0x00uy // Interrupt Enable Register typically starts at 0
    }
```

Explanation: * `let init () : Memory`: A function that returns a new `Memory` record. * `Array.create size value`: F# function to create a byte array of a given `size`, initialized with `value`. * `0xFFuy`: The `uy` suffix denotes an unsigned byte. Game Boy memory is typically initialized to `0xFF` unless specified otherwise, representing an "open bus" state. We initialize ROM to `0x00` as it will be immediately overwritten.

4. Load Cartridge ROM

The core purpose of the MMU is to load the game itself. This function takes the raw bytes of a Game Boy ROM file and places them into the `rom` region of our `Memory` state.

Append this to `src/GameboyEmulator/Mmu.fs`:

```
// src/GameboyEmulator/Mmu.fs (continued)

/// Loads a Game Boy ROM byte array into the memory's ROM region.
let loadRom (romData: byte array) (memory: Memory) : Memory =
    let romSize = min romData.Length memory.rom.Length // Don't write beyond
allocated ROM
    Array.blit romData 0 memory.rom 0 romSize
    { memory with rom = memory.rom } // Return new memory state with updated
ROM
```

Explanation: * `loadRom (romData: byte array) (memory: Memory) : Memory`: This function takes the ROM data and the current `Memory` state, returning a new `Memory` state with the ROM loaded. This adheres to functional programming principles of immutability. * `Array.blit`: Efficiently copies a section of one array to another. * `romData`: Source array. * `0`: Source start index. * `memory.rom`: Destination array. * `0`: Destination start index. * `romSize`: Number of elements to copy. * `{ memory with rom = memory.rom }`: This is F#'s record update syntax. It creates a new `Memory` record, copying all fields from the original `memory` but explicitly setting `rom` to the (now modified) `memory.rom` array. While the array itself is mutable, the `Memory` record reference is immutable, which is a common F# pattern for performance-critical mutable data structures within immutable records.

5. Implement readByte

This is the central function for fetching data from memory. Given an address, it determines which memory region that address falls into and returns the byte from that region.

Append this to `src/GameboyEmulator/Mmu.fs`:

```
// src/GameboyEmulator/Mmu.fs (continued)

/// Reads a single byte from the specified memory address.
let readByte (addr: uint16) (memory: Memory) : byte =
    match addr with
    | a when a >= MemoryMap.RomStart && a <= MemoryMap.RomEnd ->
        memory.rom.[int (a - MemoryMap.RomStart)]
    | a when a >= MemoryMap.VramStart && a <= MemoryMap.VramEnd ->
        memory.vram.[int (a - MemoryMap.VramStart)]
    | a when a >= MemoryMap.ExtRamStart && a <= MemoryMap.ExtRamEnd ->
        memory.extRam.[int (a - MemoryMap.ExtRamStart)]
    | a when a >= MemoryMap.WramStart && a <= MemoryMap.WramEnd ->
        memory.wram.[int (a - MemoryMap.WramStart)]
    | a when a >= MemoryMap.EchoRamStart && a <= MemoryMap.EchoRamEnd ->
        // Echo RAM is a mirror of WRAM (0xC000-0xDFFF)
        let wramAddr = a - MemoryMap.EchoRamStart + MemoryMap.WramStart
        memory.wram.[int (wramAddr - MemoryMap.WramStart)]
    | a when a >= MemoryMap.OamStart && a <= MemoryMap.OamEnd ->
        memory.oam.[int (a - MemoryMap.OamStart)]
    | a when a >= MemoryMap.UnusableStart && a <= MemoryMap.UnusableEnd ->
        // Reading from unusable area returns 0xFF
        0xFFuy
    | a when a >= MemoryMap.IoStart && a <= MemoryMap.IoEnd ->
        memory.io.[int (a - MemoryMap.IoStart)]
    | a when a >= MemoryMap.HramStart && a <= MemoryMap.HramEnd ->
        memory.hram.[int (a - MemoryMap.HramStart)]
    | a when a = MemoryMap.IeRegister ->
        memory.ie
    | _ ->
        // Unknown or unhandled address, typically returns 0xFF (open bus)
        // In a real emulator, this might log a warning or be more specific.
        0xFFuy // Default for unmapped memory
```

Explanation: * `match addr with | a when ... ->`: This is F#'s powerful pattern matching. We match the `addr` against ranges defined in `MemoryMap`. * `a - MemoryMap.RomStart`: We subtract the base address of the region to get the correct offset within that region's dedicated byte array. * `int (...)`: Converts the `uint16` offset to an `int` for array indexing. * `Echo RAM`: This region (`0xE000-0xFDFF`) is a direct mirror of WRAM (`0xC000-0xDFFF`). Reads/writes to Echo RAM directly affect WRAM. Our `readByte` logic correctly redirects the access. * `Unusable/Unknown`: For regions that shouldn't be accessed or are not yet implemented, we return `0xFFuy`, which is common behavior for "open bus" or uninitialized memory in hardware.

6. Implement `writeByte`

Similar to `readByte`, this function handles writing data to the correct memory region based on the address.

Append this to `src/GameboyEmulator/Mmu.fs`:

```
// src/GameboyEmulator/Mmu.fs (continued)

/// Writes a single byte to the specified memory address.
let writeByte (addr: uint16) (value: byte) (memory: Memory) : Memory =
    match addr with
    | a when a >= MemoryMap.RomStart && a <= MemoryMap.RomEnd ->
        // ROM is read-only, writes are typically ignored or modify MBC
registers
        // For now, we'll just ignore it.
        memory // Return unchanged memory
    | a when a >= MemoryMap.VramStart && a <= MemoryMap.VramEnd ->
        memory.vram.[int (a - MemoryMap.VramStart)] <- value
        memory // Return unchanged memory (array modified in place)
    | a when a >= MemoryMap.ExtRamStart && a <= MemoryMap.ExtRamEnd ->
        memory.extRam.[int (a - MemoryMap.ExtRamStart)] <- value
        memory
    | a when a >= MemoryMap.WramStart && a <= MemoryMap.WramEnd ->
        memory.wram.[int (a - MemoryMap.WramStart)] <- value
        memory
    | a when a >= MemoryMap.EchoRamStart && a <= MemoryMap.EchoRamEnd ->
        // Echo RAM is a mirror of WRAM
        let wramAddr = a - MemoryMap.EchoRamStart + MemoryMap.WramStart
        memory.wram.[int (wramAddr - MemoryMap.WramStart)] <- value
        memory
    | a when a >= MemoryMap.OamStart && a <= MemoryMap.OamEnd ->
        memory.oam.[int (a - MemoryMap.OamStart)] <- value
        memory
    | a when a >= MemoryMap.UnusableStart && a <= MemoryMap.UnusableEnd ->
        // Writes to unusable area are ignored
        memory
    | a when a >= MemoryMap.IoStart && a <= MemoryMap.IoEnd ->
        memory.io.[int (a - MemoryMap.IoStart)] <- value
        memory
    | a when a >= MemoryMap.HramStart && a <= MemoryMap.HramEnd ->
        memory.hram.[int (a - MemoryMap.HramStart)] <- value
        memory
    | a when a = MemoryMap.IeRegister ->
        { memory with ie = value } // Return new memory state with updated IE
register
    | _ ->
        // Unknown or unhandled address, ignore write for now
        memory
```

Explanation: * `writeByte (addr: uint16) (value: byte) (memory: Memory) : Memory`: Takes the address, value to write, and current `Memory` state, returning a new `Memory` state. * `<-`: This is the F# mutable assignment operator. Since `byte array` is a mutable reference type, we can modify its contents directly. * `{ memory with ie = value }`: For the `ie` (Interrupt Enable) register, which is a single `byte` field in the `Memory` record, we use the record update

syntax to create a new `Memory` record with the updated `ie` value. This maintains immutability for the record itself.

- **ROM Write:** Writes to ROM (`0x0000-0x7FFF`) are generally ignored by the hardware, or they control Memory Bank Controllers (MBCs) which we'll implement later. For now, we simply return the original `memory` unchanged.
- **Unusable/Unknown:** Writes to these areas are also ignored.

Testing & Verification

Now that we have our `Memory` module, let's verify its basic functionality. We'll add some simple tests to `Program.fs` to ensure ROM loading and memory R/W operations work as expected.

Open `src/GameboyEmulator/Program.fs` and modify it:

```
// src/GameboyEmulator/Program.fs

open System
open System.IO
open Mmu // Import our new Mmu module
open Cpu // Assuming you have Cpu.fs from the previous chapter

// Represents the overall state of the Game Boy system
type GameBoy = {
    cpu    : Cpu.State
    memory : Mmu.Memory
    // Other components will be added here (PPU, APU, etc.)
}

[<EntryPoint>]
let main argv =
    printfn "Starting Game Boy Emulator..."

    // --- MMU Testing ---
    let initialMemory = Mmu.init ()
    printfn "Memory initialized."

    // Create a dummy ROM for testing
    let dummyRom = [| 0xC3uy; 0x00uy; 0x01uy; 0xDEuy; 0xADuy; 0xBEuy; 0xEFuy
|] // Simple JR 0x01, then some data
    let memoryWithRom = Mmu.loadRom dummyRom initialMemory
    printfn "Dummy ROM loaded."

    // Verify ROM read
    let romByte0 = Mmu.readByte 0x0000u memoryWithRom
    let romByte1 = Mmu.readByte 0x0001u memoryWithRom
    let romByte2 = Mmu.readByte 0x0002u memoryWithRom
    printfn "Read from ROM: 0x%02X, 0x%02X, 0x%02X (Expected: 0xC3, 0x00,
0x01)" romByte0 romByte1 romByte2
    // Expected: 0xC3, 0x00, 0x01

    // Verify WRAM write/read
    let addressWram = Mmu.MemoryMap.WramStart + 0x10u // Example address in
WRAM
    let memoryAfterWramWrite = Mmu.writeByte addressWram 0xAAuy memoryWithRom
    let readWram = Mmu.readByte addressWram memoryAfterWramWrite
    printfn "Wrote 0xAA to WRAM at 0x%04X, Read: 0x%02X (Expected: 0xAA)" addre
ssWram readWram
    // Expected: 0xAA

    // Verify Echo RAM write/read (should affect WRAM)
    let addressEchoRam = Mmu.MemoryMap.EchoRamStart + 0x10u // Echo of WRAM
0xC010
    let memoryAfterEchoWrite = Mmu.writeByte addressEchoRam 0xBBuy memoryAfterW
ramWrite
    let readEchoRam = Mmu.readByte addressEchoRam memoryAfterEchoWrite
    let readWramViaEcho = Mmu.readByte addressWram
memoryAfterEchoWrite // Read WRAM directly
    printfn "Wrote 0xBB to Echo RAM at 0x%04X, Read: 0x%02X (Expected: 0xBB)" a
ddressEchoRam readEchoRam
    printfn "WRAM at 0x%04X now contains: 0x%02X (Expected: 0xBB)" addressWram
readWramViaEcho
    // Expected: 0xBB for both

    // Verify ROM write attempt (should be ignored)
    let memoryAfterRomWriteAttempt = Mmu.writeByte 0x0000u 0xFFuy memoryAfterEc
```

```
hoWrite
    let romByte0AfterWrite = Mmu.readByte 0x0000u memoryAfterRomWriteAttempt
    printfn "Attempted to write 0xFF to ROM at 0x0000. Read back: 0x%02X
(Expected: 0xC3)" romByte0AfterWrite
    // Expected: 0xC3 (original value)

    // Verify IE register write/read
    let memoryAfterIeWrite = Mmu.writeByte Mmu.MemoryMap.IeRegister 0x01uy memo
ryAfterRomWriteAttempt
    let ieValue = Mmu.readByte Mmu.MemoryMap.IeRegister memoryAfterIeWrite
    printfn "Wrote 0x01 to IE register, Read: 0x%02X (Expected: 0x01)" ieValue
    // Expected: 0x01

    // Instantiate the GameBoy state
    let gameBoyState = {
        cpu = Cpu.init ()
        memory = memoryAfterIeWrite // Use the memory state after all tests
    }

    // You can now access gameBoyState.cpu and gameBoyState.memory
    printfn "GameBoy system state initialized."

    0 // return an integer exit code
```

Verification Steps: 1. Save all files (`Cpu.fs`, `Mmu.fs`, `Program.fs`). 2. Open your terminal in the `src/GameboyEmulator/` directory. 3. Run `dotnet run`.

Expected Output (console):

```
Starting Game Boy Emulator...
Memory initialized.
Dummy ROM loaded.
Read from ROM: 0xC3, 0x00, 0x01 (Expected: 0xC3, 0x00, 0x01)
Wrote 0xAA to WRAM at 0xC010, Read: 0xAA (Expected: 0xAA)
Wrote 0xBB to Echo RAM at 0xE010, Read: 0xBB (Expected: 0xBB)
WRAM at 0xC010 now contains: 0xBB (Expected: 0xBB)
Attempted to write 0xFF to ROM at 0x0000. Read back: 0xC3 (Expected: 0xC3)
Wrote 0x01 to IE register, Read: 0x01 (Expected: 0x01)
GameBoy system state initialized.
```

If your output matches, congratulations! Your basic MMU is working. The CPU can now fetch instructions and interact with memory.

Production Considerations

Performance

`readByte` and `writeByte` are arguably the most frequently called functions in an emulator's core loop. Every CPU instruction fetch and data access goes through them.

- **F# Pattern Matching:** While expressive, using `match` statements with many ranges can introduce a slight overhead compared to direct `if/elif` chains or lookup tables. For now, this is perfectly acceptable and readable. If profiling shows this as a bottleneck later, we can optimize.
- **Array Access:** F# arrays (`byte array`) are efficient for raw byte storage.
- **Immutability vs. Mutability:** We've used a hybrid approach. The `Memory` record itself is treated immutably (functions return new records), but the underlying `byte array` instances are mutable. This is a pragmatic choice in F# when performance is critical for large, stateful data structures. Creating new byte arrays on every write would be prohibitively slow.

Maintainability

- **Clear Memory Map:** Defining constants in `Mmu.MemoryMap` makes the code readable and easy to update if specific address ranges need adjustment.
- **Modular Design:** Keeping MMU logic in `Mmu.fs` separates concerns, making it easier to reason about and extend (e.g., adding MBCs).

Future Enhancements: Memory Bank Controllers (MBCs)

- **Larger ROMs:** Many Game Boy games are larger than 32KB. They use MBCs, which are specialized chips on the cartridge that swap different "banks" of ROM (and sometimes external RAM) into the `0x0000-0x7FFF` and `0xA000-0xBFFF` regions. Our current `rom` array only holds 32KB. Implementing MBCs will be a significant future step, requiring `writeByte` to specific addresses in the ROM region to trigger bank switching.
- **Boot ROM:** The Game Boy actually has a small 256-byte internal boot ROM that runs on startup. This ROM is mapped to `0x0000-0x00FF` initially, then unmapped after execution. We'll address this when we set up the system's boot process.

Common Issues & Solutions

1. **Off-by-one Errors in Address Ranges:**

- **Issue:** Incorrectly calculating array offsets (e.g., `a - MemoryMap.RomStart - 1u`). This can lead to `IndexOutOfRangeException`.
- **Solution:** Double-check your subtraction logic. Remember that `RomStart` is the base, so `a - RomStart` gives the 0-indexed offset. The `+ 1u` in size calculations ensures the end address is included. Use explicit `uint16` types to prevent implicit integer conversions that might hide issues.

1. **Incorrect Handling of Read-Only Regions:**

- **Issue:** Accidentally allowing writes to ROM or unusable areas, which can corrupt the game's code or lead to unexpected behavior.
- **Solution:** Ensure your `writeByte` function explicitly handles read-only regions by doing nothing or logging a warning. Our current implementation ignores writes to ROM, which is generally correct for the hardware's behavior.

1. **Performance Bottlenecks with `readByte`/`writeByte`:**

- **Issue:** If your emulator is slow, and profiling points to these functions, the current `match` statement might be too slow for extremely tight loops.
- **Solution:** For now, optimize only if profiling indicates it's a problem. Future optimizations might include: * A single `byte array` representing the entire 64KB, with direct indexing (though this makes managing distinct regions harder). * Pre-calculating a lookup table for faster address-to-region mapping. * Using `inline` for these functions in F# to reduce call overhead (though the compiler often does this automatically).

Summary & Next Step

You've successfully built the foundation for the Game Boy's memory system! * You've mapped out the Game Boy's 64KB address space. * You've implemented a `Memory` record in F# to hold different memory regions. * Crucially, you've created `readByte` and `writeByte` functions that correctly dispatch memory accesses to the appropriate regions. * Your emulator can now load a basic ROM and interact with its internal RAM.

This MMU is the glue that connects the CPU to the rest of the Game Boy's hardware. With this in place, the CPU can now fetch instructions and data, bringing us closer to executing real Game Boy code.

The next step will be to integrate this `Memory` module with our `Cpu` module. We'll modify the CPU to use `Mmu.readByte` to fetch opcodes and operands, and `Mmu.writeByte` to store results, finally enabling our CPU to execute a small program.

🧠 Check Your Understanding

- Why is an MMU necessary in a system like the Game Boy, rather than the CPU directly addressing components?
- Explain the difference between WRAM and Echo RAM in the Game Boy's memory map. Why are they implemented this way?
- In our F# `Memory` record, why do we return a new `Memory` record when updating `ie`, but return the same `memory` record when modifying `memory.wram.[...]`?

⚡ Mini Task

Modify the `writeByte` function to print a console warning (`printfn "Warning: Write to unhandled address 0x%04X" addr`) if an address falls into the `_ ->` (catch-all) case. This can be useful for debugging later.

🚀 Scenario

You are debugging a Game Boy ROM that occasionally crashes with an `IndexOutOfRangeException` when trying to access memory. You suspect it's related to an incorrect address calculation in your `readByte` function. The crash occurs when the CPU tries to read from `0x8000`. * Which memory region is `0x8000` supposed to be in? * What specific line of code in `readByte` should you investigate for potential off-by-one errors or incorrect range checks for that region? * How would you add a temporary logging statement to pinpoint the exact value of `(a - MemoryMap.VramStart)` right before the array access?

📌 TL;DR

- The Game Boy MMU maps the CPU's 64KB address space to various hardware components.
- Key memory regions include ROM, VRAM, WRAM, OAM, I/O, and HRAM.
- `readByte` and `writeByte` functions are critical for all CPU memory interactions.

🧠 Core Flow

1. Define memory map constants for address ranges.
2. Create an F# `Memory` record holding byte arrays for each region.
3. Implement `init` to set up initial memory state.
4. Implement `loadRom` to copy cartridge data into the ROM region.
5. Implement `readByte` and `writeByte` using pattern matching over address ranges to dispatch to correct memory regions.

🚀 Key Takeaway

A well-designed MMU is the foundational abstraction layer for any emulator. It manages the complex interplay between the CPU and diverse memory-mapped hardware, providing a consistent interface for the CPU while abstracting away the underlying physical layout.

References

- Pan Docs - The Ultimate Game Boy Technical Manual
- F# Language Reference - Records
- F# Language Reference - Pattern Matching

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 04

Loading ROMs and Initial Boot Sequence

In this chapter, we transition from a theoretical CPU to a system capable of loading and preparing a Game Boy game for execution. This is the pivotal moment where your emulator begins to take on a tangible form, moving from abstract concepts to processing actual game data. We'll implement the crucial functionality of loading a Game Boy ROM file into our Memory Management Unit (MMU) and setting up the initial state of the CPU, mirroring what happens after the Game Boy's internal boot ROM completes.

By the end of this milestone, your emulator will be capable of reading a `.gb` file, storing its contents in memory, and initializing the CPU's registers to the correct post-boot state. While you won't see a game running yet, this is the foundational step that enables all subsequent instruction execution and brings us closer to a fully functional Game Boy.

Project Overview

Our overarching goal is to build a functional Game Boy emulator in F#. This involves accurately replicating the behavior of the Game Boy's CPU (SM83), Memory Management Unit (MMU), Picture Processing Unit (PPU), and other components. Each chapter builds incrementally, adding critical pieces to achieve a working system.

This chapter specifically focuses on the initial system bootstrap. We're tackling how the emulator first ingests game data (the ROM) and sets up the CPU's internal state to mimic the power-on sequence of a real Game Boy, but with a practical shortcut to speed up development.

Tech Stack

For this project, we are leveraging the following core technologies:

- **F# (F# 8.0):** The primary programming language, chosen for its strong type system, functional paradigms, and excellent interoperability with .NET.
- **`.NET SDK (8.0.x)`:** The runtime and development platform providing core libraries and tools. This includes the F# compiler and runtime.

- **SDL2 (via .NET bindings like `SDL2-CS` or `Veldrid`):** While not directly used in this chapter, this will be our chosen cross-platform graphics and input library for later chapters. We'll set up the bindings when we reach the PPU implementation.

📌 **Key Idea:** Using a modern .NET SDK ensures we benefit from performance improvements and up-to-date tooling, while F# provides the expressive power for modeling complex hardware states immutably.

Planning & Design

To get a Game Boy game running, we need to handle two primary concerns in this chapter:

1. **ROM Loading:** Reading the `.gb` file from the disk and placing its data into the MMU's address space. Game Boy ROMs are essentially just byte arrays that contain the game's executable code and data.
2. **Initial CPU State:** The real Game Boy has a small, internal 256-byte boot ROM that runs when the device powers on. This boot ROM initializes various hardware components (like sound, video, and memory) and then transfers control to the cartridge ROM at address `0x100`. For simplicity in development, and because we're not yet emulating the PPU or APU, we will skip emulating the boot ROM and instead directly initialize the CPU's registers and certain MMU registers to the state they would be in after the boot ROM has finished. This allows us to jump straight into game code.

Why skip the boot ROM? Emulating the boot ROM requires accurate PPU and APU emulation to even display the Nintendo logo and play its sound. By directly setting the post-boot state, we can focus on CPU and MMU logic first, getting to a runnable state faster. This is a common practice in emulator development for iterative progress.

Game Boy Memory Map & ROM Integration

Let's quickly revisit how the ROM fits into the Game Boy's memory map, which we discussed in Chapter 3:

- **0x0000 - 0x3FFF:** ROM Bank 0 (fixed, always accessible). This is where the initial game code resides.
- **0x4000 - 0x7FFF:** Switchable ROM Banks (controlled by Memory Bank Controllers, or MBCs, which we'll cover later).

For now, we'll load the entire ROM into an internal `byte array` within our `MMU` and ensure that at least ROM Bank 0 is copied into the `MMUState.Memory` array for immediate access.

Architecture Changes

We'll primarily extend our existing `MMU` module to include ROM loading capabilities and modify the `CPU` module to allow for initial state setup. The main application will orchestrate these steps.

This diagram illustrates the data flow: the user provides a ROM path, the `Main Application` loads it via the `File System`, the ROM data initializes the `MMU`, and finally, the `CPU` is initialized with its post-boot register values. The `MMU` and `CPU` are the core `Emulator Components`.

Step-by-Step Implementation

We'll start by enhancing our `MMU` to handle ROM data and then configure our `CPU`'s initial state.

1. Update `MMU` to Load ROMs

First, let's modify `src/GameBoyEmulator/MMU.fs` to include a way to load a byte array representing the cartridge ROM.

File: `src/GameBoyEmulator/MMU.fs`

```
// src/GameBoyEmulator/MMU.fs
module MMU

open System
open System.IO

/// Represents the Game Boy's memory.
type MMUState = {
    /// 64KB main memory array
    Memory : byte array
    /// Cartridge ROM data (full ROM, potentially multiple banks)
    Rom : byte array
}

/// Creates a new MMU state with an empty memory map and loads a ROM.
/// It initializes the memory to a post-boot ROM state.
let create (romPath: string) : MMUState =
    let memory = Array.zeroCreate 0x10000 // 64KB (0x0000 to 0xFFFF)

    let romData =
        if File.Exists(romPath) then
            File.ReadAllBytes(romPath)
        else
            // ⚠️ What can go wrong: Incorrect path or missing file.
            // This failwithf will stop execution and provide a clear error.
            failwithf "ROM file not found at %s" romPath

    // Copy ROM Bank 0 (0x0000-0x3FFF) into main memory
    // For now, we only copy the first 16KB. Memory Bank Controllers (MBCs)
    // will handle larger ROMs and bank switching in a future chapter.
    let romBank0Size = min romData.Length 0x4000 // Max 16KB for Bank 0
    Array.blit romData 0 memory 0x0000 romBank0Size

    // 🧠 Important: These values are critical for skipping the boot ROM.
    // They represent the state of various I/O and hardware registers
    // after the official Game Boy boot ROM has completed its work.
    // These are derived from Game Boy technical documentation (e.g., Pan
Docs).
    memory.[0xFF05] <- 0x00uy // TIMA - Timer counter
    memory.[0xFF06] <- 0x00uy // TMA - Timer modulo
    memory.[0xFF07] <- 0x00uy // TAC - Timer control
    memory.[0xFF10] <- 0x80uy // NR10 - Channel 1 sweep
    memory.[0xFF11] <- 0xBFuy // NR11 - Channel 1 length/duty
    memory.[0xFF12] <- 0xF3uy // NR12 - Channel 1 volume/envelope
    memory.[0xFF14] <- 0xBFuy // NR14 - Channel 1 frequency hi
    memory.[0xFF16] <- 0x3Fuy // NR21 - Channel 2 length/duty
    memory.[0xFF17] <- 0x00uy // NR22 - Channel 2 volume/envelope
    memory.[0xFF19] <- 0xBFuy // NR24 - Channel 2 frequency hi
    memory.[0xFF1A] <- 0x7Fuy // NR30 - Channel 3 DAC enable
    memory.[0xFF1B] <- 0xFFuy // NR31 - Channel 3 length
    memory.[0xFF1C] <- 0x9Fuy // NR32 - Channel 3 volume
    memory.[0xFF1E] <- 0xBFuy // NR33 - Channel 3 frequency hi
    memory.[0xFF20] <- 0xFFuy // NR41 - Channel 4 length
    memory.[0xFF21] <- 0x00uy // NR42 - Channel 4 volume/envelope
    memory.[0xFF22] <- 0x00uy // NR43 - Channel 4 polynomial counter
    memory.[0xFF23] <- 0xBFuy // NR44 - Channel 4 frequency hi
    memory.[0xFF24] <- 0x77uy // NR50 - Channel control (volume & VIN)
    memory.[0xFF25] <- 0xF3uy // NR51 - Selection of sound output terminal
    memory.[0xFF26] <- 0xF1uy // NR52 - Sound on/off (DMG specific value)
    memory.[0xFF40] <- 0x91uy // LCDC - LCD Control (important for PPU)
    memory.[0xFF42] <- 0x00uy // SCY - Scroll Y
```

```
    memory.[0xFF43] <- 0x00uy // SCX - Scroll X
    memory.[0xFF45] <- 0x00uy // LYC - LY Compare
    memory.[0xFF47] <- 0xFCuy // BGP - BG Palette Data
    memory.[0xFF48] <- 0xFFuy // OBP0 - Obj Palette 0 Data
    memory.[0xFF49] <- 0xFFuy // OBP1 - Obj Palette 1 Data
    memory.[0xFF4A] <- 0x00uy // WY - Window Y
    memory.[0xFF4B] <- 0x00uy // WX - Window X
    memory.[0xFFFF] <- 0x00uy // IE - Interrupt Enable Register

    { Memory = memory; Rom = romData }

/// Reads a byte from memory at the given address.
let readByte (mmu: MMUState) (address: uint16) : byte =
    // ⚡ Quick Note: Direct array access is highly performant.
    // In later chapters, this will involve more complex logic for I/O
registers
    // and memory bank controllers, but the fundamental access pattern remains.
    mmu.Memory.[int address]

/// Writes a byte to memory at the given address.
let writeByte (mmu: MMUState) (address: uint16) (value: byte) : MMUState =
    // ⚡ Real-world insight: While F# encourages immutability,
    // emulators often deal with large, mutable state (like memory).
    // Modifying a byte array in-place and returning the same record
    // is a pragmatic compromise for performance in such scenarios.
    mmu.Memory.[int address] <- value
    mmu // Return the same MMUState record (as only its internal mutable array
changed)
```

Explanation of Changes:

- `MMUState` **record:** We added a `Rom : byte array` field to `MMUState`. This stores the entire cartridge ROM data, which can be larger than 64KB for games with multiple memory banks.
- `create` **function:**
    - Now takes `romPath: string` as an argument to specify the ROM file to load.
    - It reads all bytes from the specified ROM file using `File.ReadAllBytes`.
    - Includes a `failwithf` call for robust error handling if the file isn't found, preventing unexpected crashes.
    - `Array.blit` is used to copy the first 16KB (ROM Bank 0, from `0x0000` to `0x3FFF`) of the loaded ROM directly into the `Memory` array. This makes the initial game code immediately accessible to the CPU.
- **Post-Boot ROM Initialization:** A series of `memory.[address] <- value` lines are added. These set specific I/O registers (like those for sound and LCD control) to their default values after the Game Boy's internal boot ROM has run. This is crucial for skipping the boot ROM and jumping directly into

game execution. These values are meticulously derived from Game Boy technical documentation like the Pan Docs.

- **`readByte` and `writeByte`:** These functions now operate on the `MMUState` record, accessing its `Memory` array. Their core logic remains direct array access, which is essential for performance.

2. Set Initial CPU State

Next, we need to initialize our CPU's registers to the state expected after the boot ROM.

File: `src/GameBoyEmulator/CPU.fs`

```
// src/GameBoyEmulator/CPU.fs
module CPU

open System

/// Represents the CPU's 8-bit registers (A, F, B, C, D, E, H, L).
type Registers = {
    A : byte
    F : byte // Flag register
    B : byte
    C : byte
    D : byte
    E : byte
    H : byte
    L : byte
}

/// Represents the CPU's 16-bit registers (SP, PC).
type SpecialRegisters = {
    SP : uint16 // Stack Pointer
    PC : uint16 // Program Counter
}

/// Represents the CPU's current state.
type CPUState = {
    Registers : Registers
    SpecialRegisters : SpecialRegisters
    // Add more fields as needed, e.g., interrupt enable flags, clock cycles
}

/// Creates a new CPU state, initialized to the post-boot ROM values.
let create () : CPUState =
    // 🧠 Important: These are the CPU register values for the original Game
Boy (DMG)
    // after its internal boot ROM has finished execution.
    // Setting these correctly is vital for games to start properly without
    // emulating the boot ROM itself.
    {
        Registers = {
            A = 0x01uy // Accumulator
            F = 0xB0uy // Flags (Z=1, N=0, H=1, C=1 - post-boot ROM state)
            B = 0x00uy
            C = 0x13uy
            D = 0x00uy
            E = 0xD8uy
            H = 0x01uy
            L = 0x4Buy
        }
        SpecialRegisters = {
            SP = 0xFFFEus // Stack Pointer: Initialized to the top of the
stack.
            PC = 0x0100us // Program Counter: Critical! This is the entry point
                          // for Game Boy cartridges after the boot ROM.
        }
    }

/// Updates the Z flag (Zero flag) based on the result.
/// The Z flag is set if the result of an operation is zero.
let private updateZFlag (flags: byte) (result: byte) =
    if result = 0x00uy then flags ||| 0x80uy // Set Z flag (bit 7)
    else flags &&& (~~0x80uy) // Clear Z flag
```

```
/// Updates the N flag (Subtraction flag).
/// The N flag is set if the last instruction was a subtraction.
let private updateNFlag (flags: byte) (isSub: bool) =
    if isSub then flags ||| 0x40uy // Set N flag (bit 6)
    else flags &&& (~~0x40uy) // Clear N flag

/// Updates the H flag (Half Carry flag).
/// The H flag is set if there's a carry from bit 3 to bit 4. Useful for BCD
operations.
let private updateHFlag (flags: byte) (carry: bool) =
    if carry then flags ||| 0x20uy // Set H flag (bit 5)
    else flags &&& (~~0x20uy) // Clear H flag

/// Updates the C flag (Carry flag).
/// The C flag is set if there's a carry from bit 7 (or bit 15 for 16-bit ops).
let private updateCFlag (flags: byte) (carry: bool) =
    if carry then flags ||| 0x10uy // Set C flag (bit 4)
    else flags &&& (~~0x10uy) // Clear C flag

// Placeholder for future instruction execution
let step (cpu: CPUState) (mmu: MMU.MMUState) : CPUState * MMU.MMUState =
    // In future chapters, this will fetch an opcode from MMU at PC,
    // decode it, execute it, update CPU state and MMU state, and increment PC.
    printfn "CPU PC: 0x%04x" cpu.SpecialRegisters.PC
    cpu, mmu // Placeholder: return current state for now
```

Explanation of Changes:

- **`create` function:** This function now initializes the `CPUState` with specific values for all registers:
    - `A = 0x01`, `F = 0xB0`, `B = 0x00`, `C = 0x13`, `D = 0x00`, `E = 0xD8`, `H = 0x01`, `L = 0x4B`. These are the standard values for the DMG (original Game Boy) CPU registers after the boot ROM has finished executing.
    - `SP = 0xFFFE`: The stack pointer is initialized to the top of the stack in High RAM.
    - `PC = 0x0100`: The program counter is set to `0x0100`. This is the entry point for Game Boy cartridges, where game code execution begins. This is a critical value for game startup.
- **Flag Helper Functions:** The `updateZFlag`, `updateNFlag`, `updateHFlag`, `updateCFlag` functions are included as a reference. While not used in this chapter, they illustrate the pattern for how flags will be managed later through bitwise operations on the `F` register.
- **`step` function:** Still a placeholder, but now it prints the current `PC`, which will be useful for verifying our initial setup.

3. Integrate into the Main Application

Finally, let's update our main program to use these new initialization functions. This `Program.fs` will serve as our emulator's entry point.

File: `src/GameBoyEmulator/Program.fs`

```
// src/GameBoyEmulator/Program.fs
open System
open System.IO
open GameBoyEmulator

[<EntryPoint>]
let main argv =
    if argv.Length < 1 then
        printfn "Usage: GameBoyEmulator <path_to_rom.gb>"
        exit 1

    let romPath = argv.[0]
    printfn "Loading ROM: %s" romPath

    try
        // 1. Initialize MMU with the loaded ROM
        let mmu = MMU.create romPath
        printfn "MMU initialized with ROM."

        // 2. Initialize CPU to its post-boot ROM state
        let cpu = CPU.create ()
        printfn "CPU initialized to post-boot state."

        // Verification output for critical registers
        printfn "Initial PC: 0x%04x" cpu.SpecialRegisters.PC
        printfn "Initial SP: 0x%04x" cpu.SpecialRegisters.SP
        printfn "Initial A: 0x%02x, F: 0x%02x" cpu.Registers.A cpu.Registers.F
        printfn "Initial B: 0x%02x, C: 0x%02x" cpu.Registers.B cpu.Registers.C
        printfn "Initial D: 0x%02x, E: 0x%02x" cpu.Registers.D cpu.Registers.E
        printfn "Initial H: 0x%02x, L: 0x%02x" cpu.Registers.H cpu.Registers.L

        // For now, we'll just step once to show the PC.
        // In later chapters, this will become a continuous emulation loop.
        let (nextCpu, nextMmu) = CPU.step cpu mmu

        0 // Exit code indicating success
    with
    | :? System.IO.FileNotFoundException as ex ->
        printfn "Error: ROM file not found. Please check the path. %s" ex.Messa
ge
        1 // Exit code indicating error
    | ex ->
        printfn "An unexpected error occurred: %s" ex.Message
        1 // Exit code indicating error
```

Explanation of Changes:

- **Command Line Argument:** The program now expects the path to a ROM file as a command-line argument. This makes it practical and flexible to load different games.

- `MMU.create romPath`: This line initializes the MMU, which now handles reading the ROM file from disk and setting up the initial memory state.
- `CPU.create ()`: This initializes the CPU with the correct post-boot register values, ready to execute game code.
- **Verification Output:** We print out the initial values of all CPU registers (`PC`, `SP`, `A`, `F`, `B`, `C`, `D`, `E`, `H`, `L`) to confirm they are set correctly. This is a crucial debugging step.
- **Error Handling:** Added a `try...with` block to gracefully handle `FileNotFoundException` if the ROM path is incorrect, and a general exception handler for any other unforeseen issues. This improves the robustness of our application.

Testing & Verification

Now, let's verify that our emulator correctly loads the ROM and sets the initial CPU state.

1. **Get a Test ROM:** You'll need an actual Game Boy ROM file. A good starting point is a simple, publicly available "hello world" ROM or one of Blargg's CPU instruction test ROMs. For instance, you can download `cpu_instrs/individual/01-special.gb` from Blargg's test ROMs. Create a `roms` folder in your project root (`GameBoyEmulator/roms/`) and place the `.gb` file there.

- **Blargg's CPU Test ROMs:** https://github.com/retroworks/gb-test-roms/tree/main/cpu_instrs

1. **Build the Project:** Open your terminal in the `GameBoyEmulator` project root and run: `bash dotnet build`
2. **Run the Emulator:** From your project root, execute the program, providing the path to your test ROM: `bash dotnet run --project src/GameBoyEmulator/GameBoyEmulator.fsproj -- roms/01-special.gb` (Adjust `roms/01-special.gb` to the actual path if you placed your ROM elsewhere.)

Expected Output:

`Loading ROM: roms/01-special.gb MMU initialized with ROM. CPU initialized to post-boot state. Initial PC: 0x0100 Initial SP: 0xFFFE Initial A: 0x01, F: 0xB0 Initial B: 0x00, C: 0x13 Initial D: 0x00, E: 0xD8 Initial H: 0x01, L: 0x4B CPU PC: 0x0100`- **`MMU`**

`initialized with ROM.`: Confirms the MMU's `create` function ran successfully, implying the ROM file was read and initial memory values were set.

- `Initial PC: 0x0100`: This is the most critical check. It confirms your CPU is starting at the correct entry point for Game Boy games, bypassing the boot ROM.
- `Initial SP: 0xFFFE`: Confirms the stack pointer is correctly set to the top of the stack.
- `Initial A: 0x01, F: 0xB0` (and other registers): Confirms other key registers are in their expected post-boot state.

If you see these values, your emulator has successfully loaded a ROM and is ready to begin executing instructions!

Production Considerations

- **Robust ROM Loading:** While `File.ReadAllBytes` is sufficient for now, a production emulator would perform more extensive validation on the ROM header (e.g., checking the Nintendo logo checksum, global ROM checksum, cartridge type, ROM size, RAM size). This prevents loading corrupted or unsupported ROMs.
- **Performance of Memory Access:** `readByte` and `writeByte` operations are in the emulator's critical path, called millions of times per second. Our current implementation using direct `byte array` access (`mmu.Memory.[int address]`) is highly efficient in .NET. Avoiding unnecessary abstractions or bounds checks in inner loops (where F# array access is already optimized) is key.
- **Maintainability and Clarity:** Keeping the ROM loading logic within the `MMU` module maintains a clean separation of concerns. The `MMU` is explicitly responsible for all memory-related operations, including how ROM data is integrated and accessed.
- **Cross-platform Compatibility:** Using `System.IO` and standard .NET types (`byte array`, `uint16`) ensures our core logic remains cross-platform. The only platform-specific concerns will arise later when integrating a graphics library.

🔥 **Optimization / Pro tip:** For extremely performance-sensitive scenarios, especially if you were to target WebAssembly or low-level native targets, you might consider `Span<byte>` or `Memory<byte>` for memory access. However, for a

desktop .NET emulator, direct `byte[]` access is typically fast enough due to JIT optimizations.

Common Issues & Solutions

1. **"ROM file not found" Error:**

- **Issue:** The `failwithf "ROM file not found..."` is triggered, or the `FileNotFoundException` is caught.
- **Solution:**
- **Verify Path:** Double-check the exact path to your `.gb` file. Ensure it's relative to where you're running the `dotnet run` command or an absolute path. File paths can be tricky across different OSes (e.g., `/` vs `\`).
- **File Existence:** Make sure the file actually exists at that location and you have read permissions.
- **Working Directory:** When running from an IDE, the working directory might be different. Running directly with `dotnet run --project ...` helps standardize the execution context. 2. **Incorrect Initial Register Values (e.g., PC not 0x0100):**
- **Issue:** Your output for `Initial PC` or other registers doesn't match the expected `0x0100`, `0xFFFE`, etc.
- **Solution:** Review your `CPU.fs` and `MMU.fs` `create` functions carefully. Ensure you've copied all the literal `byte` and `uint16` values exactly as specified in the implementation steps. A single typo can lead to unexpected behavior. These values are magic numbers derived from hardware specifications, so precision is key. 3. **No Output or Program Crashes Immediately:**
- **Issue:** The program exits without printing the expected initialization messages, or crashes with a generic error.
- **Solution:**
- **Compilation Errors:** First, ensure `dotnet build` completes without errors.
- **Unhandled Exception:** If it compiles but crashes, it's likely an unhandled exception before our `try...with` block. Add more `printfn` statements around different parts of your `main` function to pinpoint where the crash occurs. For example, print `argv.[0]` before using `romPath` to confirm the argument is received correctly.

- **Memory Issues:** While unlikely at this stage, ensure your `Array.zeroCreate` for memory is large enough (`0x10000`).

🧠 Check Your Understanding

- Why do we explicitly initialize CPU registers and specific MMU addresses in the `create` functions, rather than letting the CPU run from `0x0000`?
- What is the significance of setting the CPU's `PC` register to `0x0100`?
- How does the `MMU` handle the ROM data, and what part of the ROM is immediately accessible after loading?

⚡ Mini Task

Modify the `Program.fs` to also print the byte value at `MMU` address `0x0100` after initialization. This should be the first byte of the game's actual executable code (the first opcode).

🚀 Scenario

Imagine you're trying to load a very large Game Boy ROM (e.g., 2MB), but your `MMU.create` function only copies the first 16KB into the main `Memory` array. What problem might this cause when the CPU tries to access data beyond `0x3FFF`? How would you design the MMU to handle larger ROMs with multiple banks more effectively, without copying the entire 2MB into the 64KB addressable space?

📌 TL;DR

- **ROM Loading:** The `MMU` now handles loading a `.gb` file from disk into a `byte array` and copies the initial 16KB (ROM Bank 0) into the main addressable memory.
- **Initial State:** CPU registers (`PC=0x0100`, `SP=0xFFFE`, etc.) and key MMU I/O registers are explicitly set to their post-boot ROM values.
- **Skipping Boot ROM:** We bypass emulating the Game Boy's internal boot ROM by directly setting the system to its state after the boot sequence, allowing us to jump straight into game code.

🧠 Core Flow

1. The `Main Application` receives the ROM file path as a command-line argument.
2. `MMU.create` reads the ROM file from disk, stores the full ROM data, copies the first 16KB into the main 64KB memory map, and initializes critical I/O registers to post-boot values.
3. `CPU.create` sets the CPU's general-purpose and special registers (`PC`, `SP`, `A`, `F`, etc.) to their specific post-boot ROM values.
4. The `Main Application` verifies these initial states by printing register values and performs a single `CPU.step` (currently a placeholder).

🚀 Key Takeaway

Successfully loading a ROM and initializing the system state to a known, post-boot configuration is the critical first step in emulator development. It enables the CPU to begin executing actual game code from the correct starting point, forming the bedrock for all subsequent emulation logic.

References

- Pan Docs: https://gbdev.io/pandocs/
- Microsoft F# Language Reference: https://learn.microsoft.com/en-us/dotnet/fsharp/
- Microsoft .NET Documentation: https://learn.microsoft.com/en-us/dotnet/
- Blargg's Game Boy CPU Test ROMs: https://github.com/retroworks/gb-test-roms/tree/main/cpu_instrs

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 05

CPU Control Flow: Jumps, Calls, and Conditional Logic

In this chapter, we're going to give our Game Boy CPU the ability to make decisions and reuse code. We'll implement the crucial control flow instructions: `JP` (Jump), `JR` (Jump Relative), `CALL`, and `RET` (Return), along with their conditional variants. These instructions are fundamental to how programs execute, allowing them to branch, loop, and call subroutines.

By the end of this milestone, your emulator will be able to follow more complex program paths, enabling it to execute actual Game Boy program logic beyond simple linear instruction sequences. This is a significant step towards running real Game Boy ROMs, as it unlocks the ability for programs to react to different states and organize their code efficiently.

Project Overview

Our overarching goal is to build a functional Game Boy emulator in F# from first principles. This involves meticulously replicating the behavior of the Game Boy's hardware components, including its custom 8-bit CPU (SM83), Memory Management Unit (MMU), Picture Processing Unit (PPU), and more. Each chapter incrementally adds a critical piece of this complex system.

Tech Stack

For this project, we are leveraging:

- **F# (latest stable release via .NET SDK):** A functional-first language on the .NET platform, chosen for its strong type system, conciseness, and excellent tooling. Its immutability-first approach helps manage complex state in an emulator.
- **.NET SDK (latest stable release):** Provides the runtime, libraries, and build tools for F# applications. As of 2026-05-05, this would typically be .NET 9 or .NET 10.
- **SDL.NET (or similar):** While not directly implemented in this chapter, a cross-platform graphics library like SDL.NET will eventually be used for rendering the Game Boy's display output.

Milestones and Build Plan

This chapter focuses on enabling the CPU to alter its execution path. Our plan includes:

1. **Update CPU State:** Introduce the Stack Pointer (SP) register to our `CpuState` record.
2. **Implement Stack Operations:** Create `pushWord` and `popWord` functions in the MMU to handle 16-bit data on the stack, respecting Game Boy's little-endian architecture.
3. **Integrate Control Flow Opcodes:** Extend the `executeInstruction` function with `match` cases for `JP`, `JR`, `CALL`, and `RET` instructions, including their conditional variants.
4. **Verify Execution:** Test stack operations with unit tests and validate CPU control flow by observing CPU state during the execution of simple test ROMs.

Architecture

Implementing control flow requires a deeper interaction between the CPU and memory, specifically concerning the stack. The stack is a region of memory used by the CPU to temporarily store return addresses for `CALL` instructions and other local data.

The Stack and Stack Pointer

The Game Boy's CPU (a custom Z80 variant, often called SM83) has a 16-bit `Stack Pointer` (SP) register. The stack grows downwards from higher memory addresses to lower ones. When a value is "pushed" onto the stack, the `SP` is decremented, and the value is written to the new `SP` address. When a value is "popped," the value is read from `SP`, and `SP` is incremented. This Last-In, First-Out (LIFO) structure is crucial for managing subroutine calls.

Stack Behavior Summary: * `PUSH value`: `SP` decrements by 2 (for a 16-bit value), then `value` is written to `[SP]`. Due to little-endian ordering, the low byte goes to `SP`, and the high byte to `SP+1`. * `POP value`: `value` is read from `[SP]` (low byte at `SP`, high byte at `SP+1`), then `SP` increments by 2.

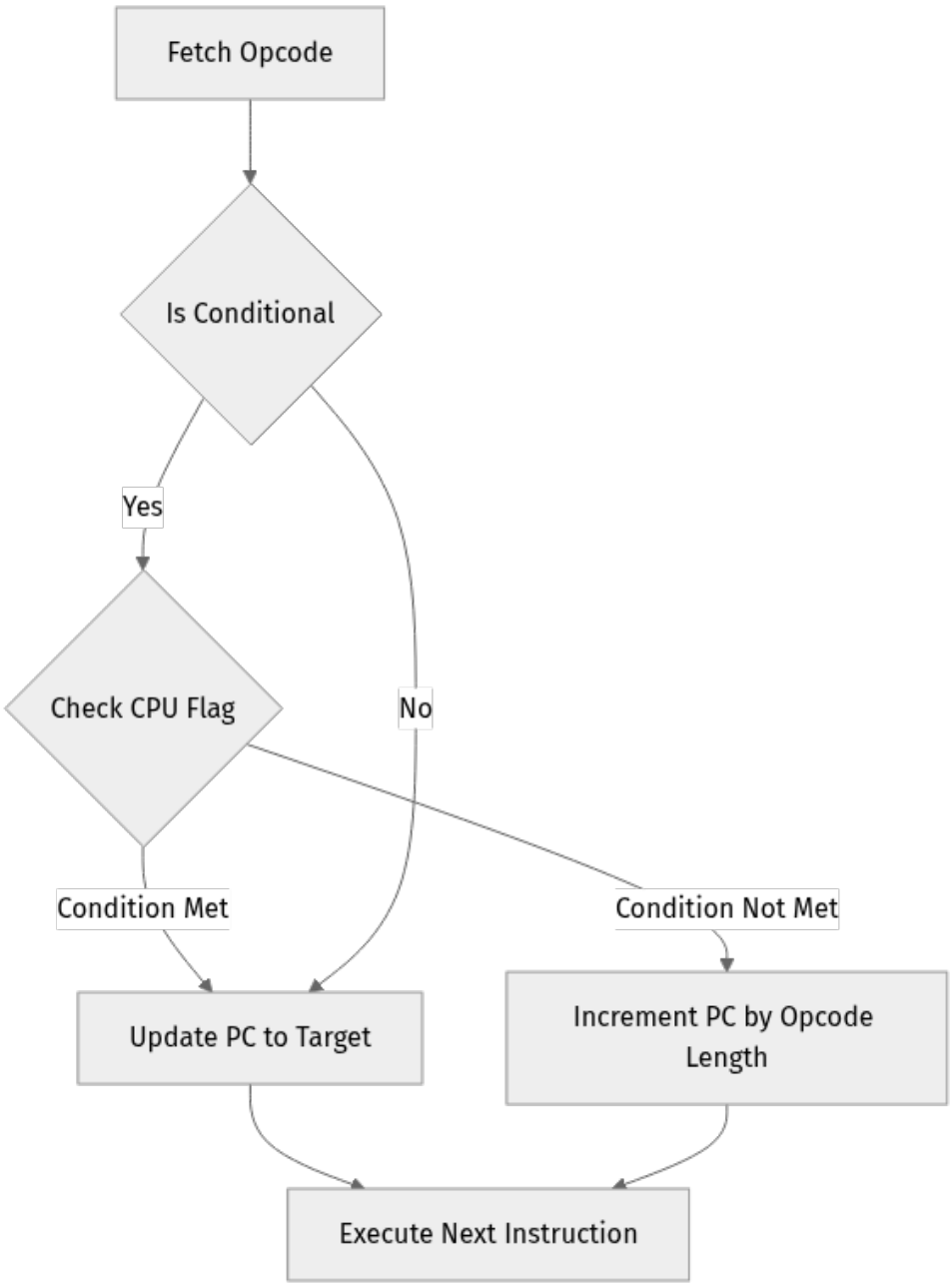
Control Flow Instruction Types

We're implementing three primary categories of control flow instructions:

1. **Jumps (`JP`, `JR`):** These instructions alter the `Program Counter` (PC) to a new address, either unconditionally or based on CPU flags.
    - `JP nn`: Jump to an absolute 16-bit address `nn`.
    - `JR e`: Jump relative by a signed 8-bit offset `e` from the current PC.
    - Conditional Jumps: `JP Z, nn` (jump if Zero flag is set), `JP NZ, nn` (jump if Zero flag is not set), `JP C, nn` (jump if Carry flag is set), `JP NC, nn` (jump if Carry flag is not set). Similar conditional variants exist for `JR`.
2. **Calls (`CALL`):** These instructions are used to call subroutines. They save the current execution context (the address of the next instruction) before jumping.
    - `CALL nn`: Push the address of the next instruction (after the `CALL` itself) onto the stack, then jump to absolute 16-bit address `nn`.
    - Conditional Calls: `CALL Z, nn`, `CALL NZ, nn`, `CALL C, nn`, `CALL NC, nn`.
3. **Returns (`RET`):** These instructions return from subroutines, restoring the program flow to where the `CALL` originated.
    - `RET`: Pop a 16-bit address from the stack, and jump to that address.
    - Conditional Returns: `RET Z`, `RET NZ`, `RET C`, `RET NC`.

CPU Decision Flow for Conditional Jumps

The CPU's logic for handling conditional control flow is straightforward: fetch, check condition, then act.



This diagram illustrates how the CPU evaluates a conditional jump or call. If the condition is met, the `PC` is updated to the target address. Otherwise, the `PC` simply advances past the instruction.

Step-by-Step Implementation

We'll start by updating our `CpuState` to include the `SP`, then implement the necessary stack operations in `Mmu.fs`, and finally integrate the new opcodes into `Cpu.fs`.

1. Update CpuState

First, open `Cpu.fs` and add the `SP` register to our `CpuState` record. We'll also update the `initialCpuState` to reflect a typical Game Boy boot-up value for `SP`.

```
// src/GbSharp.Core/Cpu.fs

module GbSharp.Core.Cpu

// ... (existing types)

type CpuState =
    {
        A: byte
        F: byte
        B: byte
        C: byte
        D: byte
        E: byte
        H: byte
        L: byte
        PC: uint16
        SP: uint16 // Add the Stack Pointer here
        IME: bool // Interrupt Master Enable
        Halted: bool
        Stopped: bool
        Flags: CpuFlags
    }
    member this.BC with get() = (uint16 this.B <<< 8) ||| (uint16 this.C) and s
et(value) = { this with B = byte (value >>> 8); C = byte (value &&& 0xFFu) }
    member this.DE with get() = (uint16 this.D <<< 8) ||| (uint16 this.E) and s
et(value) = { this with D = byte (value >>> 8); E = byte (value &&& 0xFFu) }
    member this.HL with get() = (uint16 this.H <<< 8) ||| (uint16 this.L) and s
et(value) = { this with H = byte (value >>> 8); L = byte (value &&& 0xFFu) }

// ... (rest of the module)

let initialCpuState =
    {
        A = 0x01uy // Typically 0x01 after boot ROM
        F = 0xB0uy // Typically 0xB0 after boot ROM
        B = 0x00uy
        C = 0x13uy
        D = 0x00uy
        E = 0xD8uy
        H = 0x01uy
        L = 0x4Duy
        PC = 0x0100u // Start after boot ROM, or 0x0000 if no boot ROM
        SP = 0xFFFEu // Initial stack pointer
        IME = false
        Halted = false
        Stopped = false
        Flags = { Z = true; N = false; H = true; C =
true } // Corresponds to F = 0xB0
    }
```

We initialize `SP` to `0xFFFE`. This is the typical starting point for the stack pointer after the Game Boy's boot ROM executes. This address is within the high RAM area, which is typically used for the stack.

2. Implement Stack Operations in `Mmu.fs`

The Game Boy's CPU is little-endian, meaning the least significant byte (LSB) of a 16-bit value is stored at the lower memory address, and the most significant byte (MSB) at the higher address. We need to account for this when pushing and popping 16-bit values to/from memory.

Open `Mmu.fs` and add the following functions:

```
// src/GbSharp.Core/Mmu.fs

module GbSharp.Core.Mmu

open GbSharp.Core.Cpu
// ... (existing types and functions)

/// Writes a 16-bit word to memory, little-endian.
/// The low byte (LSB) is written to 'addr', and the high byte (MSB) to 'addr +
1'.
let writeWord (mmu: MmuState) (addr: uint16) (value: uint16) : MmuState =
    mmu
    |> writeByte addr (byte (value &&& 0xFFu)) // Write low byte to addr
    |> fun newMmu -> writeByte (addr + 1u) (byte (value >>> 8))
newMmu // Write high byte to addr + 1

/// Reads a 16-bit word from memory, little-endian.
/// The low byte (LSB) is read from 'addr', and the high byte (MSB) from 'addr
+ 1'.
let readWord (mmu: MmuState) (addr: uint16) : uint16 =
    let lowByte = readByte mmu addr
    let highByte = readByte mmu (addr + 1u)
    (uint16 highByte <<< 8) ||| (uint16 lowByte)

/// Pushes a 16-bit word onto the stack.
/// SP is decremented by 2, then the value is written to the new SP address.
let pushWord (mmu: MmuState) (sp: uint16) (value: uint16) : MmuState * uint16 =
    let newSp = sp - 2u // Stack grows downwards
    let newMmu = writeWord mmu newSp value
    (newMmu, newSp)

/// Pops a 16-bit word from the stack.
/// The value is read from the current SP address, then SP is incremented by 2.
let popWord (mmu: MmuState) (sp: uint16) : uint16 * uint16 * MmuState =
    let value = readWord mmu sp
    let newSp = sp + 2u // Stack grows upwards after pop
    (value, newSp, mmu)
```

📌 `Key Idea:` The `pushWord` and `popWord` functions are crucial for managing the call stack. They abstract away the little-endian byte order and the `SP`

manipulation, making `CALL` and `RET` logic cleaner and less error-prone in the CPU emulation.

3. Implement Control Flow Opcodes in `Cpu.fs`

Now, we'll extend the `executeInstruction` function in `Cpu.fs` to handle the new opcodes. This will involve reading operands, checking flags, and updating `PC` and `SP`. We'll also add helper functions for conditional checks to keep the `match` cases clean.

```
// src/GbSharp.Core/Cpu.fs

module GbSharp.Core.Cpu

open GbSharp.Core.Mmu
// ... (existing types and functions)

/// Helper to check if Zero flag condition is met (condition = true for Z,
false for NZ)
let checkZ (flags: CpuFlags) (condition: bool) =
    if condition then flags.Z else not flags.Z

/// Helper to check if Carry flag condition is met (condition = true for C,
false for NC)
let checkC (flags: CpuFlags) (condition: bool) =
    if condition then flags.C else not flags.C

let executeInstruction (cpu: CpuState) (mmu: MmuState) : CpuState * MmuState *
int =
    let opcode = Mmu.readByte mmu cpu.PC

    let getOperand8 () = Mmu.readByte mmu (cpu.PC + 1u)
    let getOperand16 () = Mmu.readWord mmu (cpu.PC + 1u)

    let mutable cycles = 0

    let newCpu, newMmu =
        match opcode with
        // ... (existing opcodes)

        // Jumps (JP nn)
        | 0xC3uy -> // JP nn (unconditional)
            let addr = getOperand16 ()
            cycles <- 16
            { cpu with PC = addr }, mmu // PC directly set to new address
        | 0xC2uy -> // JP NZ, nn (Jump if Zero flag NOT set)
            let addr = getOperand16 ()
            cycles <- if checkZ cpu.Flags false then 16 else 12
            if checkZ cpu.Flags false then
                { cpu with PC = addr }, mmu
            else
                { cpu with PC = cpu.PC + 3u }, mmu // Skip operand
        | 0xCAuy -> // JP Z, nn (Jump if Zero flag SET)
            let addr = getOperand16 ()
            cycles <- if checkZ cpu.Flags true then 16 else 12
            if checkZ cpu.Flags true then
                { cpu with PC = addr }, mmu
            else
                { cpu with PC = cpu.PC + 3u }, mmu
        | 0xD2uy -> // JP NC, nn (Jump if Carry flag NOT set)
            let addr = getOperand16 ()
            cycles <- if checkC cpu.Flags false then 16 else 12
            if checkC cpu.Flags false then
                { cpu with PC = addr }, mmu
            else
                { cpu with PC = cpu.PC + 3u }, mmu
        | 0xDAuy -> // JP C, nn (Jump if Carry flag SET)
            let addr = getOperand16 ()
            cycles <- if checkC cpu.Flags true then 16 else 12
            if checkC cpu.Flags true then
                { cpu with PC = addr }, mmu
```

```
            else
                { cpu with PC = cpu.PC + 3u }, mmu
        | 0xE9uy -> // JP (HL) (Jump to address in HL)
            cycles <- 4
            { cpu with PC = cpu.HL }, mmu

        // Relative Jumps (JR e)
        | 0x18uy -> // JR e (unconditional)
            let offset = int8 (getOperand8 ()) // Signed 8-bit offset
            cycles <- 12
            // PC points to opcode. Instruction is 2 bytes (opcode + operand).
            // So, new PC = (current PC + 2) + offset.
            { cpu with PC = uint16 (int cpu.PC + 2 + int offset) }, mmu
        | 0x20uy -> // JR NZ, e
            let offset = int8 (getOperand8 ())
            cycles <- if checkZ cpu.Flags false then 12 else 8
            if checkZ cpu.Flags false then
                { cpu with PC = uint16 (int cpu.PC + 2 + int offset) }, mmu
            else
                { cpu with PC = cpu.PC + 2u }, mmu // Skip operand
        | 0x28uy -> // JR Z, e
            let offset = int8 (getOperand8 ())
            cycles <- if checkZ cpu.Flags true then 12 else 8
            if checkZ cpu.Flags true then
                { cpu with PC = uint16 (int cpu.PC + 2 + int offset) }, mmu
            else
                { cpu with PC = cpu.PC + 2u }, mmu
        | 0x30uy -> // JR NC, e
            let offset = int8 (getOperand8 ())
            cycles <- if checkC cpu.Flags false then 12 else 8
            if checkC cpu.Flags false then
                { cpu with PC = uint16 (int cpu.PC + 2 + int offset) }, mmu
            else
                { cpu with PC = cpu.PC + 2u }, mmu
        | 0x38uy -> // JR C, e
            let offset = int8 (getOperand8 ())
            cycles <- if checkC cpu.Flags true then 12 else 8
            if checkC cpu.Flags true then
                { cpu with PC = uint16 (int cpu.PC + 2 + int offset) }, mmu
            else
                { cpu with PC = cpu.PC + 2u }, mmu

        // Calls (CALL nn)
        | 0xCDuy -> // CALL nn (unconditional)
            let addr = getOperand16 ()
            cycles <- 24
            // Return address is PC of instruction AFTER CALL (opcode + 2
operand bytes = 3 bytes total)
            let returnAddr = cpu.PC + 3u
            let mmuAfterPush, newSp = pushWord mmu cpu.SP returnAddr
            { cpu with PC = addr; SP = newSp }, mmuAfterPush
        | 0xC4uy -> // CALL NZ, nn
            let addr = getOperand16 ()
            cycles <- if checkZ cpu.Flags false then 24 else 12
            if checkZ cpu.Flags false then
                let returnAddr = cpu.PC + 3u
                let mmuAfterPush, newSp = pushWord mmu cpu.SP returnAddr
                { cpu with PC = addr; SP = newSp }, mmuAfterPush
            else
                { cpu with PC = cpu.PC + 3u }, mmu // Skip operand
        | 0xCCuy -> // CALL Z, nn
            let addr = getOperand16 ()
```

```
            cycles <- if checkZ cpu.Flags true then 24 else 12
            if checkZ cpu.Flags true then
                let returnAddr = cpu.PC + 3u
                let mmuAfterPush, newSp = pushWord mmu cpu.SP returnAddr
                { cpu with PC = addr; SP = newSp }, mmuAfterPush
            else
                { cpu with PC = cpu.PC + 3u }, mmu
        | 0xD4uy -> // CALL NC, nn
            let addr = getOperand16 ()
            cycles <- if checkC cpu.Flags false then 24 else 12
            if checkC cpu.Flags false then
                let returnAddr = cpu.PC + 3u
                let mmuAfterPush, newSp = pushWord mmu cpu.SP returnAddr
                { cpu with PC = addr; SP = newSp }, mmuAfterPush
            else
                { cpu with PC = cpu.PC + 3u }, mmu
        | 0xDCuy -> // CALL C, nn
            let addr = getOperand16 ()
            cycles <- if checkC cpu.Flags true then 24 else 12
            if checkC cpu.Flags true then
                let returnAddr = cpu.PC + 3u
                let mmuAfterPush, newSp = pushWord mmu cpu.SP returnAddr
                { cpu with PC = addr; SP = newSp }, mmuAfterPush
            else
                { cpu with PC = cpu.PC + 3u }, mmu

        // Returns (RET)
        | 0xC9uy -> // RET (unconditional)
            cycles <- 16
            let returnAddr, newSp, mmuAfterPop = popWord mmu cpu.SP
            { cpu with PC = returnAddr; SP = newSp }, mmuAfterPop
        | 0xC0uy -> // RET NZ
            cycles <- if checkZ cpu.Flags false then 20 else 8
            if checkZ cpu.Flags false then
                let returnAddr, newSp, mmuAfterPop = popWord mmu cpu.SP
                { cpu with PC = returnAddr; SP = newSp }, mmuAfterPop
            else
                { cpu with PC = cpu.PC + 1u }, mmu // Skip opcode
        | 0xC8uy -> // RET Z
            cycles <- if checkZ cpu.Flags true then 20 else 8
            if checkZ cpu.Flags true then
                let returnAddr, newSp, mmuAfterPop = popWord mmu cpu.SP
                { cpu with PC = returnAddr; SP = newSp }, mmuAfterPop
            else
                { cpu with PC = cpu.PC + 1u }, mmu
        | 0xD0uy -> // RET NC
            cycles <- if checkC cpu.Flags false then 20 else 8
            if checkC cpu.Flags false then
                let returnAddr, newSp, mmuAfterPop = popWord mmu cpu.SP
                { cpu with PC = returnAddr; SP = newSp }, mmuAfterPop
            else
                { cpu with PC = cpu.PC + 1u }, mmu
        | 0xD8uy -> // RET C
            cycles <- if checkC cpu.Flags true then 20 else 8
            if checkC cpu.Flags true then
                let returnAddr, newSp, mmuAfterPop = popWord mmu cpu.SP
                { cpu with PC = returnAddr; SP = newSp }, mmuAfterPop
            else
                { cpu with PC = cpu.PC + 1u }, mmu

        // ... (other opcodes)
        | _ ->
```

```
            // Fallback for unimplemented opcodes
            failwithf "Unimplemented opcode: 0x%02X at PC: 0x%04X" opcode cpu.P
C

    // Advance PC for non-jump/call/return instructions if not already handled
    let finalCpu =
        if not (cpu.Halted || cpu.Stopped) && newCpu.PC = cpu.PC then // Only
advance if PC wasn't explicitly set by a control flow instruction
            match opcode with
            // Instructions that take 2 bytes (opcode + 8-bit operand)
            | 0x18uy | 0x20uy | 0x28uy | 0x30uy | 0x38uy -> { newCpu with PC =
newCpu.PC + 2u }
            // Instructions that take 3 bytes (opcode + 16-bit operand)
            | 0xC2uy | 0xC3uy | 0xCAuy | 0xD2uy | 0xDAuy
            | 0xC4uy | 0xCCuy | 0xD4uy | 0xDCuy -> { newCpu with PC =
newCpu.PC + 3u }
            // Instructions that take 1 byte (all others)
            | _ -> { newCpu with PC = newCpu.PC + 1u }
        else
            newCpu

    (finalCpu, newMmu, cycles)
```

🧠 `Important:` The `PC` calculation for relative jumps (`JR e`) is critical. The offset `e` is signed and added to `PC + 2` (the current `PC` plus the length of the `JR e` instruction itself). For `CALL nn`, the `returnAddr` pushed to the stack is `PC + 3` (the current `PC` plus the length of the `CALL nn` instruction). Pay close attention to these offsets; off-by-one errors are a very common source of bugs in emulators.

⚡ `Quick Note:` The cycle counts included are based on common Game Boy CPU documentation (like Pan Docs). These are approximations and will need fine-tuning as we build out more of the emulator's timing system. For now, they provide a reasonable baseline for instruction execution cost.

Testing & Verification

With control flow implemented, we can start verifying that our CPU can correctly alter its execution path. This involves both isolated unit tests for stack operations and observing the CPU's behavior with simple test ROMs.

Unit Tests for Stack Operations

Let's add some basic unit tests for `pushWord` and `popWord` in `MmuTests.fs` to ensure our little-endian handling and `SP` manipulation are correct.

```
// tests/GbSharp.Core.Tests/MmuTests.fs

module GbSharp.Core.Tests.MmuTests

open Xunit
open GbSharp.Core
open GbSharp.Core.Mmu

// ... (existing tests)

[<Fact>]
let ``pushWord and popWord should correctly handle 16-bit values and SP`` () =
    let initialMmu = Mmu.create () // Assuming Mmu.create initializes memory
    let initialSp = 0xFFFEu
    let testValue = 0x1234u

    // Push word
    let mmuAfterPush, spAfterPush = Mmu.pushWord initialMmu initialSp testValue
    Assert.Equal(0xFFFCu, spAfterPush) // SP should decrement by 2 (0xFFFE ->
0xFFFC)
    // Verify little-endian storage: LSB at SP, MSB at SP+1
    Assert.Equal(0x34uy, Mmu.readByte mmuAfterPush 0xFFFCu) // Low byte (0x34)
at 0xFFFC
    Assert.Equal(0x12uy, Mmu.readByte mmuAfterPush
0xFFFDu) // High byte (0x12) at 0xFFFD

    // Pop word
    let poppedValue, spAfterPop, _ = Mmu.popWord mmuAfterPush spAfterPush
    Assert.Equal(testValue, poppedValue) // Verify the correct value was popped
    Assert.Equal(initialSp, spAfterPop) // SP should return to its original
value (0xFFFC -> 0xFFFE)
```

Verification with Simple Test ROMs

While full Game Boy ROMs will require more components (especially the PPU), we can use small, custom-written ROMs (or early stages of well-known test suites like Blargg's CPU instruction tests) to verify these opcodes.

Consider a simple Game Boy assembly program:

```
; Example Assembly (RGBDS syntax)
; Filename: control_flow_test.asm
    SECTION "Test Code",ROM0[$100] ; Start at 0x100, past typical boot ROM area

    LD SP, $FFFE    ; Initialize stack pointer
    LD HL, $C050    ; Load an arbitrary value into HL for JP (HL) test

Start:
    CALL Subroutine1 ; Call a subroutine
    LD A, $AA       ; This instruction should execute after Subroutine1 returns
    JR NZ, SkipJump ; Conditional relative jump (if Z flag NOT set)
    JP $0000        ; Should not be reached if NZ is true

SkipJump:
    LD B, $BB       ; Should execute
    JP $0100        ; Jump back to Start for a simple loop (or halt here for a
real test)

Subroutine1:
    LD C, $CC       ; Set C register
    JR Z, SkipRet   ; Conditional relative jump (if Z flag SET, should NOT jump
if Z is false)
    RET             ; Return from subroutine
SkipRet:
    LD D, $DD       ; Should not be reached in normal flow
    RET
```

To test this: 1. **Assemble the ROM:** Use an assembler like `RGBDS` (RGBASM) to compile this `.asm` file into a `.gb` ROM. `bash rgbasm -o control_flow_test.o control_flow_test.asm rgblink -o control_flow_test.gb control_flow_test.o` 2. **Load into Emulator:** Modify your `Program.fs` (or main emulation loop) to load `control_flow_test.gb`. 3. **Log CPU State:** Implement detailed logging of `cpu.PC`, `cpu.SP`, and relevant registers (`A`, `B`, `C`, `D`, `E`, `H`, `L`, `F`) at each instruction step. 4. **Trace Execution:** Manually trace the `PC` and `SP` values in your logs: * Observe `SP` initializing to `0xFFFE`. * `CALL Subroutine1` should push `0x103` (address of `LD A, $AA`) onto the stack, and `PC` should jump to `Subroutine1`'s address. `SP` should decrement to `0xFFFC`. * Inside `Subroutine1`, `LD C, $CC` executes. * `JR Z, SkipRet` should not jump if the Z flag is false (which it typically is after `LD C, $CC` unless `CC` was 0). * `RET` should pop `0x103` from the stack, restoring `PC` to `0x103`. `SP` should increment back to `0xFFFE`. * `LD A, $AA` should then execute. * `JR NZ, SkipJump` should jump to `SkipJump` (if Z flag is still false). * `LD B, $BB` should execute. * `JP $0100` should jump `PC` back to `0x0100` (Start).

This manual inspection, possibly augmented with a debugger or detailed logging, is invaluable for understanding the subtle interactions of hardware components.

⚡ `Real-world insight:` Emulator development heavily relies on detailed logging of CPU state (PC, SP, registers, flags) and memory dumps. This "printf

debugging" approach, especially in early stages, is often more effective than traditional debuggers for understanding the subtle, cycle-accurate behavior of emulated hardware.

Production Considerations

The correctness and performance of control flow instructions are paramount. Any error here will quickly lead to a crashed or misbehaving emulator, making the system unstable and unpredictable.

- **Correctness is King:** A single byte off in a `PC` calculation, an incorrect signed offset for `JR`, or an endianness error in stack operations will cause the CPU to execute garbage data or jump to incorrect locations. This leads to immediate crashes or subtle, hard-to-debug issues later in development. Thoroughly test `PC` updates and `pushWord`/`popWord` against official documentation.
- **Performance:** Jumps, calls, and returns are frequent operations in any program. Our current F# implementation, while functional and clear, might involve some overhead due to immutable state updates and pattern matching. For a Game Boy, which runs at approximately 4.19 MHz, the CPU emulation loop needs to be extremely fast. We'll revisit performance optimizations later, but for now, focus on absolute correctness.
- **Stack Overflow/Underflow:** While Game Boy software typically manages the stack within its limits, a robust emulator might include checks to detect `SP` going out of the expected RAM range (e.g., `0xC000-0xDFFF` for WRAM, or `0xFFFE` downwards). This could indicate a bug in the emulated program or, more critically, in the emulator itself. Detecting this early can prevent memory corruption.

`🔥 Optimization / Pro tip:` For extreme performance-critical sections, especially the main CPU loop, some functional emulators might resort to highly optimized, potentially more imperative, internal structures or leverage F#'s mutable features carefully. However, prioritize correctness and clarity first; optimize only when profiling reveals a bottleneck.

Common Issues & Solutions

1. **Incorrect `PC` after Jump/Call:**

- **Issue:** The CPU jumps to the wrong address, or returns to an incorrect instruction, leading to unexpected program flow or crashes.
- **Cause:** Miscalculation of operand addresses (`PC + 1u`, `PC + 2u`), or incorrect handling of signed offsets for `JR`. For `CALL`, pushing `PC` instead of `PC + instruction_length` (the address after the `CALL` instruction).
- **Solution:** Double-check the Game Boy CPU documentation (e.g., Pan Docs, GBDEV Wiki) for each instruction's operand length and how `PC` is modified. Log `PC` before and after each instruction to trace the flow. ⚠️ `What can go wrong:` For `JR e`, the `e` is a signed 8-bit value. Using `uint8` directly without casting to `int8` before arithmetic operations will lead to incorrect signed extension, causing jumps to wildly wrong addresses. Always cast `byte` from `readByte` to `int8` before performing arithmetic with it for relative offsets.

1. **Stack Corruption:**

- **Issue:** `CALL`s push incorrect return addresses, or `RET`s pop garbage values, leading to program crashes or jumps to random memory locations.
- **Cause:** Incorrect `pushWord`/`popWord` implementation, particularly regarding the little-endian byte order or `SP` increment/decrement logic.
- **Solution:** Thoroughly test `pushWord` and `popWord` in isolation (as shown in unit tests). Print `SP` and the stack memory region (e.g., `0xFFF0` to `0xFFFF`) before and after calls/returns to verify values are pushed and popped correctly. ⚡ `Real-world insight:` The stack is a common vector for security exploits in real systems. While our emulator isn't directly exposed to external threats, a corrupted stack in emulation can mimic such issues, making it a valuable learning experience in debugging memory-related problems.

1. **Conditional Jumps/Calls Not Working:**

- **Issue:** Instructions like `JP Z, nn` always jump, or never jump, regardless of the Zero flag's state, leading to incorrect program logic.
- **Cause:** Incorrectly checking the `CpuFlags` record, or the flags themselves are not being set correctly by previous arithmetic/logic instructions (from previous chapters).

- **Solution:** Ensure the `checkZ` and `checkC` helper functions are correct. Verify that arithmetic and logic instructions are correctly updating the `Z`, `N`, `H`, `C` flags in `CpuState.Flags`. Debug by inspecting the `Flags` state immediately before a conditional instruction.

🧠 Check Your Understanding

- Why is the stack pointer initialized to `0xFFFE` in the Game Boy, and what does this imply about the direction the stack grows?
- Explain the critical difference between `JP nn` and `JR e` instructions, both in terms of addressing and common use cases.
- What is the significance of pushing `PC + 3u` as the return address for a `CALL nn` instruction, rather than just `PC`?

⚡ Mini Task

Modify a simple Game Boy assembly program (or create a new one if you're comfortable with assembly) to include a `CALL` to a subroutine, and then a `RET`. Add an `LD A, $00` instruction before a `JP Z, $XXXX` and another `LD A, $01` before a `JP NZ, $YYYY`. Load this ROM into your emulator and trace the `PC` and `SP` values, along with the Zero flag, to confirm correct conditional execution.

🚀 Scenario

You've implemented all control flow instructions, but when running a test ROM, the emulator consistently crashes with an "Unimplemented opcode" error after a specific `CALL` instruction, even though the opcode itself is implemented. You suspect stack corruption. How would you approach debugging this, focusing on the stack, `PC`, and memory dumps? What specific data points would you log or inspect?

📌 TL;DR

- CPU control flow instructions (`JP`, `JR`, `CALL`, `RET`) enable complex program execution paths.
- The Stack Pointer (`SP`) and memory stack are fundamental for `CALL`/`RET` operations, managing return addresses.

- Correctly handling 16-bit values (little-endian) and `SP` manipulation in `pushWord`/`popWord` is crucial for stack integrity.
- Precise `PC` calculation, especially for relative jumps and call return addresses, is a common source of emulator bugs.

🧠 Core Flow

1. **Extend `CpuState`:** Add the `SP` register to track the stack's current top.
2. **Implement Stack Primitives:** Create `pushWord` and `popWord` in `Mmu.fs` to safely interact with memory, handling little-endian byte order and `SP` updates.
3. **Integrate Opcodes:** Add `match` cases for all `JP`, `JR`, `CALL`, and `RET` variants into the `executeInstruction` loop in `Cpu.fs`.
4. **Verify Logic:** Ensure `PC` and `SP` are updated correctly, paying close attention to instruction lengths and conditional flag checks.
5. **Test and Debug:** Use unit tests for stack operations and detailed logging with simple test ROMs to trace execution flow.

🚀 Key Takeaway

Mastering CPU control flow is the gateway to executing complex program logic. The interplay between the Program Counter, Stack Pointer, and memory stack is a fundamental pattern in computer architecture, and understanding its precise, cycle-accurate implementation is crucial for any low-level system development.

References

1. **Pan Docs:** The Ultimate Game Boy Technical Manual. Essential for Game Boy hardware details, including CPU, MMU, and PPU specifications. https://gbdev.io/pandocs/
2. **F# Language Reference:** Microsoft Learn. Official F# documentation, providing syntax, language features, and best practices. https://learn.microsoft.com/en-us/dotnet/fsharp/
3. **.NET SDK:** Microsoft Learn. Official .NET documentation, covering the runtime, libraries, and tools for F# development. https://learn.microsoft.com/en-us/dotnet/

4. **Game Boy CPU Opcodes:** GBDEV Wiki. Detailed opcode information for the SM83 CPU, including instruction formats, effects on flags, and cycle counts. https://gbdev.io/wiki/index.php?title=CPU_Instruction_Set

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 06

Interrupts and the Main CPU Execution Loop

Introduction

In the previous chapters, we laid the groundwork for our Game Boy emulator by implementing the CPU's core instruction set and a basic Memory Management Unit. However, a real system isn't just a CPU executing instructions sequentially. Hardware components like the display, timer, and input devices need to signal the CPU when an event occurs, and the CPU needs a way to respond. This is where **interrupts** come in.

This chapter will guide you through understanding and implementing the Game Boy's interrupt system. More critically, we'll build the **main CPU execution loop**, the heart of our emulator that orchestrates the CPU, PPU, and other components to run in a synchronized, cycle-accurate manner. By the end of this chapter, your emulator will have a functional, albeit basic, timing mechanism and the ability to respond to hardware events, bringing us closer to a fully interactive system.

Planning & Design

Building a Game Boy emulator requires careful synchronization of several independent "hardware" components. The CPU executes instructions, the Picture Processing Unit (PPU) renders graphics, the Timer keeps track of time, and the Audio Processing Unit (APU) generates sound. All these components operate based on the same underlying clock cycles.

The Game Boy Interrupt System

The Game Boy CPU (a custom Sharp SM83) uses a simple interrupt mechanism. When a hardware event occurs (e.g., the screen finishes drawing a frame, or the timer overflows), the corresponding hardware component "requests" an interrupt. The CPU then checks if that interrupt is enabled and if overall interrupts are master enabled. If both conditions are met, the CPU temporarily pauses its current execution, saves its current program counter (PC) to the stack, jumps to a specific memory address (the interrupt vector), and clears the master interrupt enable flag.

The Game Boy has five primary interrupt sources, each with its own vector address:

- **VBlank (Vertical Blanking Interrupt)**: Triggered when the PPU finishes drawing a frame and is in the vertical blanking period. This is crucial for synchronizing screen updates and game logic. Vector: `0x0040`.
- **LCD Stat (LCD Status Interrupt)**: Triggered by various events related to the PPU's LCD status register, such as specific scanline conditions. Vector: `0x0048`.
- **Timer Interrupt**: Triggered when the internal timer overflows. Vector: `0x0050`.
- **Serial Interrupt**: Triggered when a byte has been transferred via the serial cable. Vector: `0x0058`.
- **Joypad Interrupt**: Triggered when a button state changes (pressed or released). Vector: `0x0060`.

To manage these, the CPU uses two key registers and one internal flag:

- **Interrupt Enable (IE) Register (`0xFFFF`)**: A byte where each bit corresponds to an interrupt source. If a bit is set, that interrupt source is enabled.
- **Interrupt Flag (IF) Register (`0xFF0F`)**: A byte where each bit corresponds to an interrupt source. Hardware components set these bits when an interrupt is requested. The CPU clears them when servicing.
- **Interrupt Master Enable (IME) Flag**: An internal CPU flag (boolean) that globally enables or disables all interrupts. It can be set (`EI` instruction) or cleared (`DI` instruction).

🧠 **Important:** An interrupt is only serviced if its corresponding bit in `IF` is set, its corresponding bit in `IE` is set, AND the `IME` flag is true.

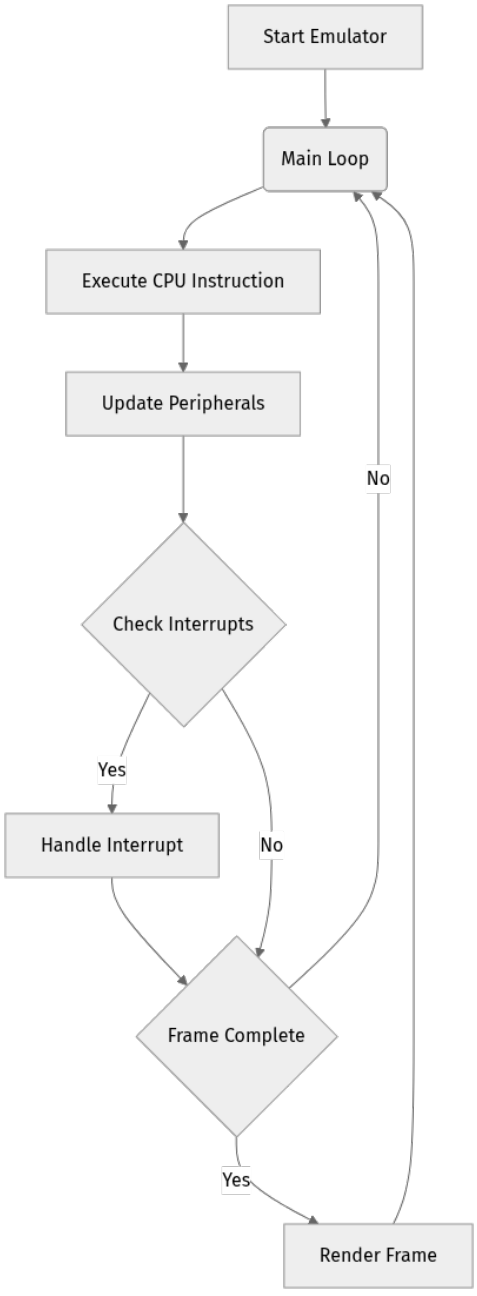
The Main Execution Loop

The Game Boy's CPU runs at approximately 4.19 MHz. The PPU updates the screen at 60 frames per second (FPS). To accurately emulate this, our main loop must:

1. **Execute CPU instructions**: Each instruction takes a specific number of CPU cycles.
2. **Advance other components**: Based on the cycles consumed by the CPU, we must update the PPU, Timer, and APU by the same number of cycles.

3. **Check and handle interrupts**: After each instruction (or at specific points in the loop), we check for pending interrupts.

A typical Game Boy frame consists of exactly 70224 CPU cycles. Our main loop will aim to complete this many cycles within one frame, updating the screen once per frame.



The diagram above illustrates the continuous cycle of our emulator. The "Update PPU" and "Update Timer" steps will consume the same number of CPU cycles that the CPU instruction just took, ensuring all components stay synchronized.

Step-by-Step Implementation

We'll start by enhancing our `CpuState` to manage interrupts, then create a basic `Timer` component, and finally assemble the main execution loop in an `Emulator` module.

1. Update CPU State for Interrupts

First, let's add the necessary fields to our `CpuState` record in `Src/GameBoy/Cpu.fs`.

```
// Src/GameBoy/Cpu.fs

module GameBoy.Cpu

open GameBoy.Mmu

/// Represents the CPU's current state.
type CpuState =
    {
        Registers : Registers
        PC : uint16 // Program Counter
        SP : uint16 // Stack Pointer
        IME : bool // Interrupt Master Enable
        IE : byte // Interrupt Enable Register (0xFFFF)
        IF : byte // Interrupt Flag Register (0xFF0F)
        Halt : bool // CPU is halted
        Stop : bool // CPU is stopped
        CyclesThisInstruction : int // Cycles consumed by the last instruction
    }

// ... existing code ...

/// Initializes a new CPU state.
let init () : CpuState =
    {
        Registers = Registers.init ()
        PC = 0x0100us // Game Boy boots at 0x0100
        SP = 0xFFFEus
        IME = false // Interrupts disabled by default on boot
        IE = 0x00uy // No interrupts enabled initially
        IF = 0x00uy // No interrupts pending initially
        Halt = false
        Stop = false
        CyclesThisInstruction = 0
    }

// ... rest of Cpu.fs ...
```

Explanation: - We added `IME` (Interrupt Master Enable), `IE` (Interrupt Enable Register), and `IF` (Interrupt Flag Register) to `CpuState`. - `IME` is a boolean flag, `IE` and `IF` are bytes. - On initialization, `IME` is `false` (interrupts are globally disabled until an `EI` instruction is executed), and `IE`/`IF` are `0x00`.

2. Implement Interrupt Handling Logic

Now, let's add a function `handleInterrupts` to `Cpu.fs` that will check for and service interrupts. This function will return the updated `CpuState` and the number of cycles taken to handle the interrupt.

```
// Src/GameBoy/Cpu.fs

module GameBoy.Cpu

open GameBoy.Mmu

// ... CpuState and init function ...

/// Represents the status of the CPU after an operation.
type CpuResult = { State: CpuState; Cycles: int }

// ... existing opcode definitions and helper functions ...

/// Handles pending interrupts if IME is enabled and an interrupt is requested
and enabled.
let handleInterrupts (cpu: CpuState) (mmu: MmuState) : CpuResult * MmuState =
    if cpu.IME then
        // Check each interrupt in priority order (VBlank -> LCD Stat -> Timer
-> Serial -> Joypad)
        let pendingInterrupts = cpu.IF &&& cpu.IE

        // VBlank Interrupt (Bit 0)
        if (pendingInterrupts &&& 0x01uy) <> 0x00uy then
            let newMmu = Mmu.writeByte mmu 0xFF0F (cpu.IF &&& (0xFEuy)) //
Clear VBlank bit in IF
            let newMmu' = Mmu.writeByte newMmu (cpu.SP - 1us) (byte (cpu.PC
>>> 8)) // Push PC high byte
            let newMmu'' = Mmu.writeByte newMmu' (cpu.SP - 2us) (byte cpu.PC) /
/ Push PC low byte
            ( { cpu with PC = 0x0040us; SP = cpu.SP - 2us; IME = false; IF = (c
pu.IF &&& (0xFEuy)) }, newMmu'' ) // 5 cycles for interrupt handling
        // LCD Stat Interrupt (Bit 1)
        else if (pendingInterrupts &&& 0x02uy) <> 0x00uy then
            let newMmu = Mmu.writeByte mmu 0xFF0F (cpu.IF &&&
(0xFDu)) // Clear LCD Stat bit in IF
            let newMmu' = Mmu.writeByte newMmu (cpu.SP - 1us) (byte (cpu.PC
>>> 8))
            let newMmu'' = Mmu.writeByte newMmu' (cpu.SP - 2us) (byte cpu.PC)
            ( { cpu with PC = 0x0048us; SP = cpu.SP - 2us; IME = false; IF = (c
pu.IF &&& (0xFDu)) }, newMmu'' )
        // Timer Interrupt (Bit 2)
        else if (pendingInterrupts &&& 0x04uy) <> 0x00uy then
            let newMmu = Mmu.writeByte mmu 0xFF0F (cpu.IF &&& (0xFBuy)) //
Clear Timer bit in IF
            let newMmu' = Mmu.writeByte newMmu (cpu.SP - 1us) (byte (cpu.PC
>>> 8))
            let newMmu'' = Mmu.writeByte newMmu' (cpu.SP - 2us) (byte cpu.PC)
            ( { cpu with PC = 0x0050us; SP = cpu.SP - 2us; IME = false; IF = (c
pu.IF &&& (0xFBuy)) }, newMmu'' )
        // Serial Interrupt (Bit 3)
        else if (pendingInterrupts &&& 0x08uy) <> 0x00uy then
            let newMmu = Mmu.writeByte mmu 0xFF0F (cpu.IF &&& (0xF7uy)) //
Clear Serial bit in IF
            let newMmu' = Mmu.writeByte newMmu (cpu.SP - 1us) (byte (cpu.PC
>>> 8))
            let newMmu'' = Mmu.writeByte newMmu' (cpu.SP - 2us) (byte cpu.PC)
            ( { cpu with PC = 0x0058us; SP = cpu.SP - 2us; IME = false; IF = (c
pu.IF &&& (0xF7uy)) }, newMmu'' )
        // Joypad Interrupt (Bit 4)
        else if (pendingInterrupts &&& 0x10uy) <> 0x00uy then
            let newMmu = Mmu.writeByte mmu 0xFF0F (cpu.IF &&& (0xEFuy)) //
```

```
Clear Joypad bit in IF
            let newMmu' = Mmu.writeByte newMmu (cpu.SP - 1us) (byte (cpu.PC
>>> 8))
            let newMmu'' = Mmu.writeByte newMmu' (cpu.SP - 2us) (byte cpu.PC)
            ( { cpu with PC = 0x0060us; SP = cpu.SP - 2us; IME = false; IF = (c
pu.IF &&& (0xEFuy)) }, newMmu'' )
        else
            // No enabled and pending interrupts
            ( cpu, mmu )
    else
        // IME is disabled, no interrupts serviced
        ( cpu, mmu )
```

Explanation: - The `handleInterrupts` function takes the current `CpuState` and `MmuState`. - It first checks `cpu.IME`. If `false`, no interrupts are serviced. - It calculates `pendingInterrupts` by bitwise ANDing `cpu.IF` and `cpu.IE`. This gives us a byte where only the bits corresponding to both requested and enabled interrupts are set. - It then checks these bits in priority order (VBlank has highest priority). - If an interrupt is found: - The corresponding bit in `IF` is cleared (using a bitwise AND with a mask). - The current `PC` is pushed onto the stack (high byte then low byte). This requires writing to MMU. - The `SP` (Stack Pointer) is decremented by 2. - `IME` is set to `false` to prevent nested interrupts. - The `PC` is set to the interrupt's vector address. - The updated `CpuState` and `MmuState` are returned. - If no enabled and pending interrupt is found, the original `CpuState` and `MmuState` are returned.

⚡ **Quick Note:** The `CpuResult` type we defined previously is still relevant for `Cpu.step`, but `handleInterrupts` directly returns the updated CPU and MMU states because it's tightly coupled with MMU writes for the stack.

3. Basic Timer Implementation

The Game Boy's timer is a simple but important component. Let's create `Src/GameBoy/Timer.fs`.

```
// Src/GameBoy/Timer.fs

module GameBoy.Timer

open GameBoy.Mmu
open GameBoy.Cpu // Needed to set IF bit

/// Represents the state of the Game Boy's internal timer.
type TimerState =
    {
        DivRegister :
uint16 // Divider Register (0xFF04) - increments at 16384 Hz
        TimaRegister : byte // Timer Counter (0xFF05) - increments based on TAC
        TmaRegister : byte // Timer Modulo (0xFF06) - value TIMA is reloaded
with
        TacRegister : byte // Timer Control (0xFF07) - controls timer enable
and clock select
        InternalDivCounter : int // Internal counter for DIV register
        InternalTimaCounter : int // Internal counter for TIMA register
    }

/// Initializes a new Timer state.
let init () : TimerState =
    {
        DivRegister = 0x0000us
        TimaRegister = 0x00uy
        TmaRegister = 0x00uy
        TacRegister = 0x00uy
        InternalDivCounter = 0
        InternalTimaCounter = 0
    }

/// Calculates the clock speed for TIMA based on TAC.
let private getTimaClockSpeed (tac: byte) : int =
    match tac &&& 0x03uy with // Bits 0-1 of TAC
    | 0x00uy -> 1024 // 4096 Hz (CPU clock / 1024)
    | 0x01uy -> 16 // 262144 Hz (CPU clock / 16)
    | 0x02uy -> 64 // 65536 Hz (CPU clock / 64)
    | 0x03uy -> 256 // 16384 Hz (CPU clock / 256)
    | _ -> failwith "Invalid TAC clock select" // Should not happen

/// Updates the timer state based on CPU cycles.
let updateTimer (timer: TimerState) (mmu: MmuState) (cycles: int) : TimerState
* MmuState =
    let mutable currentTimer = timer
    let mutable currentMmu = mmu

    // Update DIV register
    currentTimer <- { currentTimer with InternalDivCounter = currentTimer.Inter
nalDivCounter + cycles }
    while currentTimer.InternalDivCounter >= 256
do // DIV increments every 256 CPU cycles (4.19MHz / 256 = 16384 Hz)
        currentTimer <- { currentTimer with InternalDivCounter = currentTimer.I
nternalDivCounter - 256 }
        currentTimer <- { currentTimer with DivRegister = currentTimer.DivRegis
ter + 1us }

    // Update TIMA register if enabled
    if (currentTimer.TacRegister &&& 0x04uy) <> 0x00uy then // Bit 2 of TAC is
Timer Enable
        let clockSpeed = getTimaClockSpeed currentTimer.TacRegister
```

```
        currentTimer <- { currentTimer with InternalTimaCounter =
currentTimer.InternalTimaCounter + cycles }
        while currentTimer.InternalTimaCounter >= clockSpeed do
            currentTimer <- { currentTimer with InternalTimaCounter = currentTi
mer.InternalTimaCounter - clockSpeed }
            let newTima = currentTimer.TimaRegister + 1uy
            if newTima > 0xFFuy then // Overflow
                currentTimer <- { currentTimer with TimaRegister =
currentTimer.TmaRegister } // Reload from TMA
                // Request Timer Interrupt (Bit 2 of IF)
                currentMmu <- Mmu.writeByte currentMmu 0xFF0F (currentMmu.Memor
y.[0xFF0F] ||| 0x04uy)
            else
                currentTimer <- { currentTimer with TimaRegister = newTima }

    (currentTimer, currentMmu)
```

Explanation: - `TimerState` holds the four timer registers (`DIV`, `TIMA`, `TMA`, `TAC`) and internal counters (`InternalDivCounter`, `InternalTimaCounter`) to track sub-cycle increments. - `init()` sets initial values. - `getTimaClockSpeed` translates the `TAC` register's clock select bits into the number of CPU cycles per TIMA increment. - `updateTimer` is the core logic: - It takes `cycles` (from CPU) and updates `InternalDivCounter` and `InternalTimaCounter`. - `DIV` increments every 256 CPU cycles. - `TIMA` increments based on `TAC`'s clock speed. If `TIMA` overflows, it's reloaded from `TMA`, and the Timer interrupt bit (bit 2) in the `IF` register (`0xFF0F`) is set. - Notice the use of `mutable` for `currentTimer` and `currentMmu` within `updateTimer`. While F# favors immutability, in performance-critical inner loops where state is frequently updated (like counters), using mutable locals can sometimes be a pragmatic choice, especially when the mutable state is confined to a single function. This balances functional purity with practical performance.

⚡ **Real-world insight:** Accurate timer emulation is critical for many games, especially those with precise timing-based mechanics or sound effects. A slight deviation here can cause games to run too fast, too slow, or break entirely.

4. Placeholder PPU Update

For now, our PPU doesn't render much, but it needs to track cycles and request the VBlank interrupt. Let's add a simple `updatePpu` function to `Src/GameBoy/Ppu.fs`.

```
// Src/GameBoy/Ppu.fs

module GameBoy.Ppu

open GameBoy.Mmu
open GameBoy.Cpu // Needed to set IF bit

// ... existing PpuState and init ...

/// Total CPU cycles per frame (approx. 4.19 MHz / 60 FPS)
let private TOTAL_CYCLES_PER_FRAME = 70224

/// Updates the PPU state based on CPU cycles.
let updatePpu (ppu: PpuState) (mmu: MmuState) (cycles: int) : PpuState * MmuSta
te =
    let mutable currentPpu = ppu
    let mutable currentMmu = mmu

    currentPpu <- { currentPpu with CyclesThisFrame = currentPpu.CyclesThisFram
e + cycles }

    // Check for VBlank interrupt
    if currentPpu.CyclesThisFrame >= TOTAL_CYCLES_PER_FRAME then
        // Request VBlank interrupt (Bit 0 of IF)
        currentMmu <- Mmu.writeByte currentMmu 0xFF0F (currentMmu.Memory.[0xFF0
F] ||| 0x01uy)
        currentPpu <- { currentPpu with CyclesThisFrame = currentPpu.CyclesThis
Frame - TOTAL_CYCLES_PER_FRAME } // Reset for next frame
        // For now, we also clear the screen here (in a real PPU, this would be
a more complex rendering step)
        // currentPpu <- { currentPpu with ScreenBuffer = Array.create (160 *
144) 0x00FF0000u } // Example: clear to green

    (currentPpu, currentMmu)
```

Explanation: - `updatePpu` increments `CyclesThisFrame` by the CPU `cycles`. - When `CyclesThisFrame` exceeds `TOTAL_CYCLES_PER_FRAME` (70224 cycles), it means a frame has finished. - At this point, the VBlank interrupt (bit 0) in `IF` (`0xFF0F`) is requested. - `CyclesThisFrame` is then decremented to account for the cycles "overflowing" into the next frame.

5. Integrate Interrupt Handling into `Cpu.step`

Now, we need to modify our `Cpu.step` function to check for interrupts before executing an instruction and after. This is crucial for handling cases where `IME` is enabled or disabled by an instruction.

```
// Src/GameBoy/Cpu.fs

module GameBoy.Cpu

open GameBoy.Mmu
open GameBoy.Registers

// ... CpuState, init, CpuResult, handleInterrupts ...

/// Executes a single CPU instruction.
let step (cpu: CpuState) (mmu: MmuState) : CpuResult * MmuState =
    let mutable currentCpu = cpu
    let mutable currentMmu = mmu
    let mutable cyclesConsumed = 0

    // Check for and handle interrupts if IME is enabled and pending
    // This check is performed before fetching an instruction
    let (cpuAfterInterruptCheck, mmuAfterInterruptCheck) = handleInterrupts cur
rentCpu currentMmu
    if cpuAfterInterruptCheck.PC <> currentCpu.PC then // If PC changed, an
interrupt was serviced
        // An interrupt was handled, 5 cycles are consumed for the ISR entry
        ( { State = cpuAfterInterruptCheck; Cycles = 5 }, mmuAfterInterruptChec
k )
    else
        currentCpu <- cpuAfterInterruptCheck
        currentMmu <- mmuAfterInterruptCheck

        // If CPU is halted, just consume cycles until an interrupt occurs
        if currentCpu.Halt then
            // In halt mode, CPU consumes cycles but doesn't execute
instructions
            // It waits for an interrupt to occur. We'll simply return 4 cycles
for now.
            // A more accurate model would check IF/IE for pending interrupts
every cycle.
            ( { State = currentCpu; Cycles = 4 }, currentMmu ) // Consume
minimal cycles while halted
        else if currentCpu.Stop then
            // In stop mode, CPU and LCD are off. Only external reset or joypad
interrupt can wake it.
            // For now, just consume 4 cycles.
            ( { State = currentCpu; Cycles = 4 }, currentMmu ) // Consume
minimal cycles while stopped
        else
            // Fetch opcode
            let opcode = Mmu.readByte currentMmu currentCpu.PC
            currentCpu <- { currentCpu with PC = currentCpu.PC + 1us }

            // Decode and execute opcode
            let (nextCpu, nextMmu, instructionCycles) =
                match opcode with
                // 0x00 NOP
                | 0x00uy -> ({ currentCpu with CyclesThisInstruction = 4 }, cur
rentMmu, 4)
                // 0x01 LD BC,d16
                | 0x01uy ->
                    let val1 = Mmu.readByte currentMmu currentCpu.PC
                    let val2 = Mmu.readByte currentMmu (currentCpu.PC + 1us)
                    let value = (uint16 val2 <<< 8) ||| (uint16 val1)
                    let newRegisters = Registers.setBC currentCpu.Registers val
```

```
ue
                    ({ currentCpu with Registers = newRegisters; PC = currentCp
u.PC + 2us; CyclesThisInstruction = 12 }, currentMmu, 12)
                // ... (add all other opcodes here) ...

                // 0xFB EI - Enable Interrupts
                | 0xFBuy -> ({ currentCpu with IME = true; CyclesThisInstructio
n = 4 }, currentMmu, 4)
                // 0xF3 DI - Disable Interrupts
                | 0xF3uy -> ({ currentCpu with IME = false; CyclesThisInstructi
on = 4 }, currentMmu, 4)

                // Example RST 0x00 (0xC7) - Call 0x0000
                | 0xC7uy ->
                    let newMmu = Mmu.writeByte currentMmu (currentCpu.SP -
1us) (byte (currentCpu.PC >>> 8))
                    let newMmu' = Mmu.writeByte newMmu (currentCpu.SP - 2us) (b
yte currentCpu.PC)
                    ({ currentCpu with PC = 0x0000us; SP = currentCpu.SP -
2us; CyclesThisInstruction = 16 }, newMmu', 16)

                // Unknown opcode
                | _ ->
                    printfn "Unknown opcode: 0x%02x at 0x%04x" opcode currentCp
u.PC
                    failwithf "Unknown opcode: 0x%02x at 0x%04x" opcode current
Cpu.PC

            cyclesConsumed <- instructionCycles // cyclesConsumed was 0 if no
interrupt was handled
            ( { State = nextCpu; Cycles = cyclesConsumed }, nextMmu )
```

Explanation: - The `Cpu.step` function now includes a check for interrupts before fetching and executing an instruction. This handles cases where an interrupt occurs while the CPU is halted or waiting. - If `handleInterrupts` returns a `CpuState` with a changed `PC`, it means an interrupt was serviced. We then return immediately with the updated state and the 5 cycles overhead for the interrupt. - If no interrupt was serviced, we proceed to normal instruction execution. - We've added placeholder logic for `Halt` and `Stop` states, which consume a minimal number of cycles. - Crucially, we've added the `EI` (0xFB) and `DI` (0xF3) opcodes to set/clear the `IME` flag. These are essential for controlling the interrupt system. - The return type of `step` is `CpuResult * MmuState` to propagate the MMU state changes (e.g., from pushing to stack).

⚠️ **What can go wrong:** The timing of interrupt checks and instruction execution is extremely subtle in real Game Boy hardware. Our current `step` function checks interrupts before an instruction. Some sources suggest checks might happen after an instruction, or even mid-instruction for certain events. If games behave strangely, this is a prime area to investigate for timing inaccuracies. Pan Docs is your friend here.

6. Create the Emulator Loop

Finally, let's create a top-level `Emulator` module that orchestrates all components. Create `Src/GameBoy/Emulator.fs`.

```
// Src/GameBoy/Emulator.fs

module GameBoy.Emulator

open GameBoy.Cpu
open GameBoy.Mmu
open GameBoy.Ppu
open GameBoy.Timer

/// Represents the overall state of the Game Boy emulator.
type EmulatorState =
    {
        Cpu : CpuState
        Mmu : MmuState
        Ppu : PpuState
        Timer : TimerState
        IsRunning : bool
    }

/// Initializes a new Emulator state.
let init (romBytes: byte array) : EmulatorState =
    let initialMmu = Mmu.init romBytes
    {
        Cpu = Cpu.init ()
        Mmu = initialMmu
        Ppu = Ppu.init ()
        Timer = Timer.init ()
        IsRunning = true
    }

/// The main emulator loop, stepping all components.
let run (initialState: EmulatorState) =
    let mutable emulator = initialState

    let rec loop () =
        if emulator.IsRunning then
            // Step CPU and get cycles consumed
            let (cpuResult, newMmuFromCpu) = Cpu.step emulator.Cpu emulator.Mmu
            emulator <- { emulator with Cpu = cpuResult.State; Mmu = newMmuFrom
Cpu }
            let cycles = cpuResult.Cycles

            // Update PPU based on CPU cycles
            let (newPpu, newMmuFromPpu) = Ppu.updatePpu emulator.Ppu emulator.M
mu cycles
            emulator <- { emulator with Ppu = newPpu; Mmu = newMmuFromPpu }

            // Update Timer based on CPU cycles
            let (newTimer, newMmuFromTimer) = Timer.updateTimer emulator.Timer
emulator.Mmu cycles
            emulator <- { emulator with Timer = newTimer; Mmu =
newMmuFromTimer }

            // Potentially add APU update here later

            // Recursive call for the next step
            loop ()
        else
            printfn "Emulator stopped."

    loop ()
```

Explanation: - `EmulatorState` is a record that aggregates the states of all major components. - `init` creates an initial `EmulatorState` by initializing each component and loading the ROM into the MMU. - `run` is a recursive function that forms our main emulator loop: - It calls `Cpu.step` to execute one CPU instruction (or handle an interrupt). - It then uses the `cycles` returned by `Cpu.step` to update `Ppu.updatePpu` and `Timer.updateTimer`. This ensures all components advance by the same amount of time. - The `MmuState` is passed around and updated by each component that modifies memory (like `Cpu` pushing to stack, `Ppu` writing to `IF`, `Timer` writing to `IF`). This reflects how hardware interacts with shared memory. - The loop continues as long as `IsRunning` is true.

🔥 **Optimization / Pro tip:** While a simple recursive loop is good for clarity, in a real production emulator, you might batch CPU instruction execution (e.g., run CPU for a fixed number of cycles, then update PPU/Timer once) or use a cycle-accurate scheduler to avoid deep recursion and overhead. For now, this step-by-step approach is sufficient.

7. Update `Program.fs` to use the Emulator

Finally, let's modify our `Program.fs` to load a ROM and start the emulator.

```
// Program.fs

open System.IO
open GameBoy.Emulator

[<EntryPoint>]
let main argv =
    if argv.Length = 0 then
        printfn "Usage: dotnet run <rom_path>"
        1 // Indicate error
    else
        let romPath = argv.[0]
        if not (File.Exists romPath) then
            printfn "Error: ROM file not found at %s" romPath
            1 // Indicate error
        else
            printfn "Loading ROM: %s" romPath
            let romBytes = File.ReadAllBytes romPath
            printfn "ROM loaded. Size: %d bytes." romBytes.Length

            let emulator = Emulator.init romBytes
            printfn "Starting emulator..."
            Emulator.run emulator
            printfn "Emulator finished."
            0 // Indicate success
```

Explanation: - The `main` function now takes the ROM path as a command-line argument. - It loads the ROM bytes. - It initializes the `EmulatorState` with these bytes. - It then calls `Emulator.run` to start the main loop.

Testing & Verification

With the main loop and basic interrupt system in place, we can start observing the emulator's behavior.

1. **Compile and Run:** Navigate to your project root in the terminal and run: `bash dotnet build dotnet run -- <path_to_your_rom.gb>` For example: `bash dotnet run -- ../roms/tetris.gb` Initially, the emulator will likely just print "Emulator finished." if `IsRunning` is never set to false, or crash on an unimplemented opcode. This is expected. The goal is to observe the internal state.
2. **Debugging Interrupts:**
    - **Set breakpoints**: Place breakpoints in `Cpu.handleInterrupts`, `Ppu.updatePpu` (where VBlank is requested), and `Timer.updateTimer` (where Timer interrupt is requested).
    - **Inspect `IF` and `IE`**: During execution, inspect the values of `emulator.Cpu.IF` and `emulator.Cpu.IE`. You should see `IF` bits being set by `Ppu.updatePpu` and `Timer.updateTimer`.
    - **Verify `PC` jump**: When `Cpu.handleInterrupts` is triggered, check if `cpu.PC` correctly jumps to `0x0040` (for VBlank) or `0x0050` (for Timer) and if `cpu.IME` becomes `false`.
    - **Test ROMs**: Use simple Blargg's test ROMs like `cpu_instrs/individual/01-special.gb` which specifically test `DI`/`EI` instructions and interrupt behavior. You'll need to implement more opcodes for these to run fully, but you can trace the `IME` flag changes.
3. **Debugging Timing:**
    - Add `printfn` statements inside `Emulator.run` to print `cpuResult.Cycles`, `emulator.Ppu.CyclesThisFrame`, and `emulator.Timer.DivRegister` periodically.
    - Observe how these values increment. `Ppu.CyclesThisFrame` should eventually reach `70224` and then reset, triggering a VBlank interrupt.
    - The `Timer.DivRegister` should increment much slower, at 16384 Hz.

Production Considerations

- **Cycle Accuracy**: This is paramount for emulator accuracy. Every component must advance by the exact number of CPU cycles. Small discrepancies will lead to desynchronization, visual glitches, or broken game

logic. Consult Pan Docs thoroughly for cycle counts of each instruction and hardware event.

- **Performance**: The `run` loop is the most performance-critical part of your emulator. Every function called within this loop (`Cpu.step`, `Ppu.updatePpu`, `Timer.updateTimer`) needs to be as efficient as possible. F# record updates (`with` syntax) create new records, which can generate some garbage. For extreme performance, you might consider `mutable` fields or `ref` cells for very hot counters or state, but only if profiling proves it's a bottleneck.
- **Memory Management**: Continuously passing and updating `MmuState` (which contains the entire memory array) can be expensive if not handled carefully. F# records are immutable, so each `Mmu.writeByte` creates a new `MmuState` with a new (or copied) internal array. For better performance with large mutable data structures like memory, consider an `Array<byte>` wrapped in a `ref` cell or a `Memory<byte>`/`Span<byte>` if you want to optimize for .NET's low-level memory access. For now, the current approach is acceptable for correctness.

Common Issues & Solutions

1. **Interrupts Not Firing**:

- **Issue**: The Game Boy screen is blank, or games relying on VBlank don't progress.
- **Cause**: `IME` might be `false`, `IE` might not have the correct bit set, or `IF` bits are never being set by the hardware components (PPU, Timer).
- **Solution**: Debug `Cpu.handleInterrupts`. Check the values of `cpu.IME`, `cpu.IE`, and `cpu.IF` at runtime. Ensure `Ppu.updatePpu` and `Timer.updateTimer` correctly set the `IF` bits. Verify `EI` and `DI` opcodes are correctly implemented and being used by the ROM.

1. **Incorrect Timing / Game Runs Too Fast/Slow**:

- **Issue**: Game music plays too fast/slow, animations are off, or frame rate is inconsistent.
- **Cause**: The `cycles` returned by `Cpu.step` are incorrect for some opcodes, or the `updatePpu`/`updateTimer` functions are not advancing by the correct amount of cycles.
- **Solution**: Double-check all opcode cycle counts against reliable documentation (Pan Docs). Ensure `TOTAL_CYCLES_PER_FRAME` is accurate.

Profile the `run` loop and verify that each component correctly consumes the `cycles` provided.

1. **Stack Overflow (F# `run` loop)**:

- **Issue**: Your emulator crashes with a stack overflow exception after running for some time.
- **Cause**: The `loop ()` recursive call in `Emulator.run` is not tail-recursive. F# can optimize tail-recursive calls into iterative loops, but if any operation happens after the recursive call, it breaks tail recursion.
- **Solution**: Ensure the `loop ()` call is the very last operation in the `loop` function. If you need to do something after, restructure the loop or use an explicit `while` loop (which F# can translate from specific recursive patterns). For `run` as written, it should be tail-recursive if nothing else follows the `loop ()` call. If it's still an issue, you might need to convert it to an iterative `while` loop explicitly.

Summary & Next Step

In this chapter, you've equipped your Game Boy emulator with the crucial ability to handle hardware interrupts, allowing external components to signal the CPU. You've also built the foundational **main execution loop**, synchronizing the CPU, PPU, and Timer components by advancing them based on consumed CPU cycles. This is a significant leap towards a functional emulator, providing the timing backbone for all subsequent features.

While our PPU and Timer implementations are still basic, they now correctly interact with the CPU's interrupt system. The emulator can conceptually "run" through instructions and advance internal hardware states in a synchronized manner.

Next, we'll dive deeper into the **Picture Processing Unit (PPU)**, implementing the logic to actually render the Game Boy's graphics, from tiles and backgrounds to sprites and LCD control.

🧠 Check Your Understanding

- What are the five main interrupt sources on the Game Boy, and what is their priority order?
- Explain the purpose of the `IME`, `IE`, and `IF` registers/flags in the Game Boy's interrupt system.

- Why is cycle accuracy crucial for an emulator's main execution loop, and what happens if it's not maintained?

⚡ Mini Task

- Modify the `Cpu.init` function to start with `IE` and `IF` having specific bits set (e.g., `IE = 0x01uy` for VBlank, `IF = 0x01uy` for a pending VBlank). Then, debug and observe if `handleInterrupts` correctly services the VBlank interrupt on the first CPU step (assuming `IME` is enabled by an `EI` instruction in your ROM).

🚀 Scenario

You're running a Game Boy ROM, and the screen remains completely black, but the CPU appears to be executing instructions. You suspect a problem with the PPU's ability to request VBlank interrupts, or the CPU's ability to service them. Outline your debugging strategy, focusing on the components and registers introduced in this chapter, to pinpoint the issue.

📌 TL;DR

- **Interrupts** allow hardware to signal the CPU, crucial for events like VBlank and Timer.
- The Game Boy uses `IME` (Master Enable), `IE` (Enabled Masks), and `IF` (Flagged Requests) to manage interrupts.
- The **Main Execution Loop** synchronizes CPU, PPU, and Timer by advancing them based on CPU cycles consumed.

🧠 Core Flow

1. CPU executes an instruction, returning cycles consumed.
2. PPU and Timer components advance their internal state by the same number of cycles.
3. If PPU or Timer events occur, they set corresponding bits in the `IF` register.
4. CPU checks `IF` and `IE` (if `IME` is true) to service any pending, enabled interrupts.

🚀 Key Takeaway

Accurate cycle-based synchronization and robust interrupt handling are the backbone of any functional emulator, ensuring faithful reproduction of hardware behavior and enabling games to run as intended.

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

References

- Pan Docs: https://gbdev.io/pandocs/
- Game Boy CPU (SM83) instruction set: https://gbdev.io/pandocs/CPU_Instruction_Set.html
- Game Boy Interrupts: https://gbdev.io/pandocs/Interrupts.html
- Blargg's Game Boy Test ROMs (often found in emulator source repositories, e.g., via a quick search for "Blargg Game Boy test ROMs GitHub")
- F# Language Reference: https://learn.microsoft.com/en-us/dotnet/fsharp/
- .NET SDK: https://dotnet.microsoft.com/download

CHAPTER 07

Picture Processing Unit (PPU) Part 1: VRAM and Background Rendering

Introduction

So far, our Game Boy emulator can execute CPU instructions and manage memory, but it's a silent, black box. This chapter changes that. We're about to bring the Game Boy to life by tackling the Picture Processing Unit (PPU) – the hardware responsible for all the visuals. This is a significant milestone, as seeing actual graphics from a ROM is incredibly rewarding and validates much of our prior work.

In this first part of PPU emulation, we'll focus on understanding the Game Boy's display architecture, specifically how Video RAM (VRAM) stores graphical data, how tiles are defined, and how the PPU renders the background layer. By the end of this chapter, our emulator will be able to display static backgrounds from simple Game Boy ROMs, making the project visually verifiable.

The PPU is one of the most complex components of the Game Boy, demanding precise timing and careful interpretation of hardware specifications. We'll break it down into manageable pieces, starting with the fundamentals of background rendering.

Planning & Design

The Game Boy's PPU is responsible for drawing a 160x144 pixel display. It operates by drawing one horizontal line (a "scanline") at a time, moving from top to bottom. This process involves fetching tile data from VRAM, assembling it into a background layer, and then drawing it to the screen.

Game Boy Display Architecture Overview

The Game Boy's LCD has a fixed resolution of 160 pixels wide by 144 pixels high. The PPU cycles through different modes as it draws the screen: - **Mode 2 (OAM Scan):** The PPU spends about 80 CPU cycles scanning Object Attribute Memory (OAM) to find sprites that appear on the current scanline. - **Mode 3 (Drawing Pixels):** The PPU spends about 172-289 CPU cycles (variable) fetching tile data,

background map data, and sprite data, then drawing the pixels for the current scanline. This is the most computationally intensive phase. - **Mode 0 (HBlank):** After drawing a scanline, the PPU enters a Horizontal Blanking period for about 204 CPU cycles. This is a good time for the CPU to access VRAM or OAM without contention. - **Mode 1 (VBlank):** Once all 144 visible scanlines are drawn, the PPU enters a Vertical Blanking period. This lasts for 10 scanlines (lines 144-153) and takes roughly 4560 CPU cycles in total. This is the ideal time to update the screen buffer.

PPU Memory and Registers

The PPU interacts heavily with specific memory regions and I/O registers:

- **Video RAM (VRAM):** `0x8000-0x9FFF`
  - `0x8000-0x97FF`: Tile Data. This area stores the actual pixel patterns for 8x8 tiles. Each tile uses 16 bytes (2 bytes per row, 8 rows). Each pixel has 2 bits, allowing 4 colors.
  - `0x9800-0x9BFF`: Tile Map 0. A 32x32 grid of tile indices for the background layer.
  - `0x9C00-0x9FFF`: Tile Map 1. Another 32x32 grid of tile indices. The LCDC register determines which map is active.
- **PPU I/O Registers:**
  - `0xFF40` (LCDC): LCD Control. Crucial for enabling/disabling PPU features (LCD, background, sprites, tile map selection, tile data selection).
  - `0xFF41` (STAT): LCD Status. Contains the current PPU mode, LYC compare flag, and interrupt enable bits.
  - `0xFF42` (SCY): Scroll Y. Vertical scroll offset for the background.
  - `0xFF43` (SCX): Scroll X. Horizontal scroll offset for the background.
  - `0xFF44` (LY): LCDC Y-Coordinate. The current scanline being drawn (0-153).
  - `0xFF45` (LYC): LY Compare. An interrupt can be triggered when LY equals LYC.
  - `0xFF47` (BGP): Background Palette. Defines the 4 colors used for the background.
  - `0xFF48` (OBP0), `0xFF49` (OBP1): Object Palettes. Define colors for sprites (covered in Part 2).

Modeling the PPU State

We'll introduce a new `PpuState` record to encapsulate all PPU-related registers and internal state. This state will be updated by the CPU in cycles.

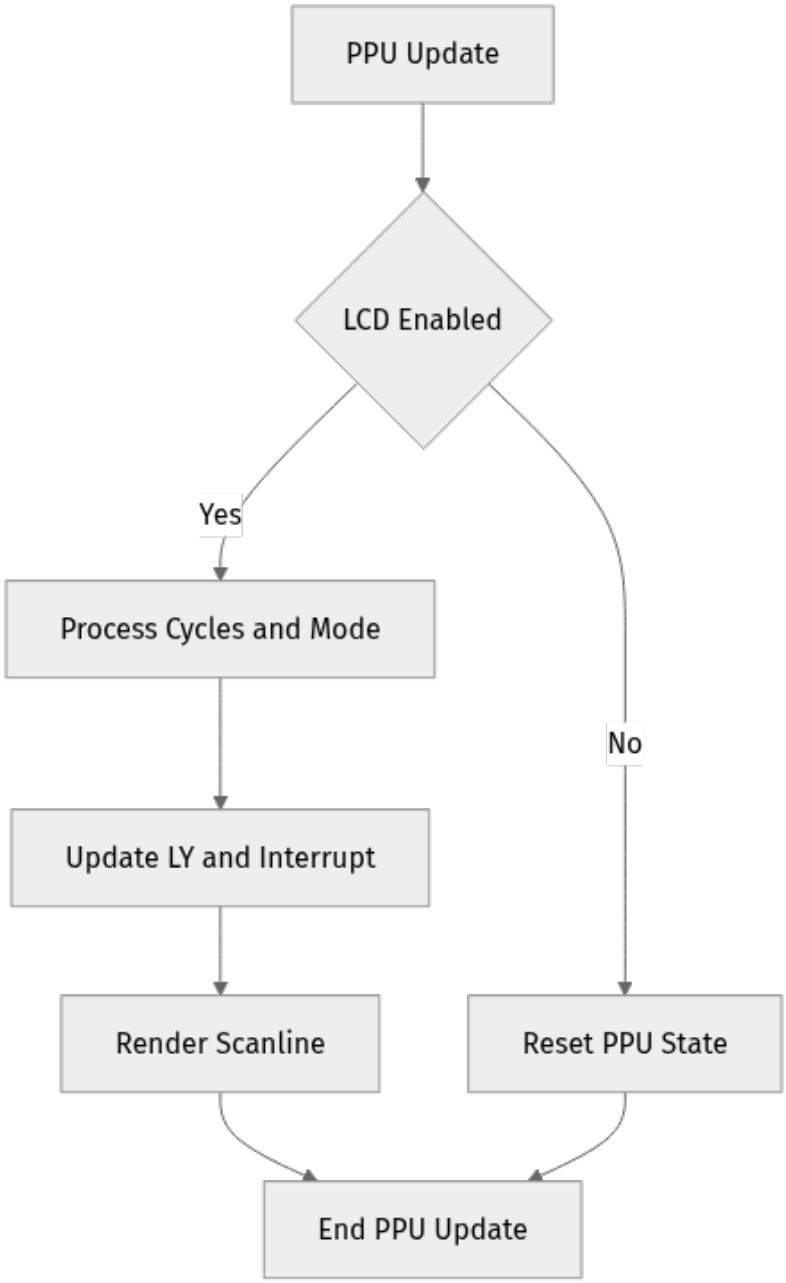
```
// In Ppu.fs
type PpuMode =
    | HBlank = 0
    | VBlank = 1
    | OAMScan = 2
    | DrawingPixels = 3

type PpuState = {
    mutable Lcdc : byte // LCD Control (0xFF40)
    mutable Stat : byte // LCD Status (0xFF41)
    mutable Scy : byte  // Scroll Y (0xFF42)
    mutable Scx : byte  // Scroll X (0xFF43)
    mutable Ly : byte   // LCDC Y-Coordinate (0xFF44)
    mutable Lyc : byte  // LY Compare (0xFF45)
    mutable Bgp : byte  // Background Palette (0xFF47)
    mutable Obp0 : byte // Object Palette 0 (0xFF48)
    mutable Obp1 : byte // Object Palette 1 (0xFF49)

    mutable CyclesThisScanline : int
    mutable CurrentFrameBuffer : byte [] // 160 * 144 * 4 bytes for RGBA
    mutable FrameReady : bool
}
```

PPU Rendering Flow

The core PPU logic will reside in an `updatePpu` function, which will be called by the main emulator loop for every CPU cycle (or a block of cycles).



PPU Update Cycle: 1. **Accumulate Cycles:** The `updatePpu` function receives the number of CPU cycles executed. 2. **Mode Progression:** Based on accumulated cycles, the PPU transitions between modes (OAM Scan, Drawing Pixels, HBlank). 3. **Scanline Increment:** When a scanline completes (after HBlank), the `LY` register is incremented. 4. **VBlank:** When `LY` reaches 144, the PPU enters VBlank, and a VBlank interrupt is triggered. The `FrameReady` flag is set. 5. **Render Scanline (Mode 3):** During the "Drawing Pixels" mode, we'll implement the logic to fetch tile data and draw pixels to our internal frame buffer.

Graphics Library: SDL2-CS

For rendering, we'll use `SDL2-CS`, which is a C# binding for the SDL2 library, fully compatible with F# and .NET. SDL (Simple DirectMedia Layer) is a cross-platform

development library designed to provide low-level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL and Direct3D.

⚡ **Quick Note:** `SDL2-CS` is a NuGet package that wraps the native SDL2 library. You'll need the native SDL2 runtime binaries installed on your system or bundled with your application. For Windows, this typically means `SDL2.dll` in your executable directory. For macOS and Linux, it's usually `libSDL2.dylib` or `libSDL2.so`.

Step-by-Step Implementation

We'll start by defining the PPU state, then integrate it into our `MMU` for register access, and finally implement the core `updatePpu` and `renderScanline` logic.

1. Update `Memory.fs` for PPU Registers

First, we need to ensure our `MMU` can read and write to the PPU's I/O registers. We'll add specific handling for the PPU's memory-mapped registers.

File: `src/GameBoy/Memory.fs`

```
module GameBoy.Memory

open System

// ... (existing MemoryState type)

type MemoryState = {
    // ... (existing fields)
    mutable Vram : byte [] // 0x8000-0x9FFF
    mutable Oam : byte []  // 0xFE00-0xFE9F
    mutable Lcdc : byte // 0xFF40
    mutable Stat : byte // 0xFF41
    mutable Scy : byte  // 0xFF42
    mutable Scx : byte  // 0xFF43
    mutable Ly : byte   // 0xFF44
    mutable Lyc : byte  // 0xFF45
    mutable Bgp : byte  // 0xFF47
    mutable Obp0 : byte // 0xFF48
    mutable Obp1 : byte // 0xFF49
    mutable Dma : byte  // 0xFF46 (DMA Transfer)
    mutable If : byte   // 0xFF0F (Interrupt Flag)
    mutable Ie : byte   // 0xFFFF (Interrupt Enable)
}

// ... (existing `createMemoryState` function)
let createMemoryState (romData: byte []) =
    let state = {
        // ... (existing initializations)
        Vram = Array.zeroCreate (0x9FFF - 0x8000 + 1)
        Oam = Array.zeroCreate (0xFE9F - 0xFE00 + 1)
        Lcdc = 0x91uy // Default boot ROM value
        Stat = 0x00uy
        Scy = 0x00uy
        Scx = 0x00uy
        Ly = 0x00uy
        Lyc = 0x00uy
        Bgp = 0xFCuy // Default palette
        Obp0 = 0xFFuy
        Obp1 = 0xFFuy
        Dma = 0x00uy
        If = 0xE1uy // Default boot ROM value
        Ie = 0x00uy
    }
    // ... (existing ROM loading)
    state

// ... (existing `readByte` function)
let readByte (addr: uint16) (state: MemoryState) =
    match addr with
    // ... (existing ranges)
    | _ when addr >= 0x8000us && addr <= 0x9FFFus -> state.Vram.[int (addr - 0x
8000us)] // VRAM
    | _ when addr >= 0xFE00us && addr <= 0xFE9Fus -> state.Oam.[int (addr - 0xF
E00us)] // OAM
    | 0xFF40us -> state.Lcdc
    | 0xFF41us -> state.Stat
    | 0xFF42us -> state.Scy
    | 0xFF43us -> state.Scx
    | 0xFF44us -> state.Ly
    | 0xFF45us -> state.Lyc
    | 0xFF46us -> state.Dma // DMA register
```

```
    | 0xFF47us -> state.Bgp
    | 0xFF48us -> state.Obp0
    | 0xFF49us -> state.Obp1
    | 0xFF0Fus -> state.If
    | 0xFFFFus -> state.Ie
    | _ ->
        // ... (existing default read)
        state.Hram.[int (addr - 0xFF80us)] // HRAM
```

Explanation: - We've added `Vram` and `Oam` arrays to `MemoryState` to represent these memory regions. - New fields (`Lcdc`, `Stat`, `Scy`, etc.) are added to `MemoryState` to hold the values of the PPU's I/O registers. These are `mutable` because the PPU (and CPU) will modify them. - `createMemoryState` now initializes these PPU registers with typical boot ROM values. - The `readByte` function is updated with `match` cases to correctly return the values from these new fields when their respective addresses are queried.

File: `src/GameBoy/Memory.fs` (continued, `writeByte`)

```
let writeByte (addr: uint16) (value: byte) (state: MemoryState) =
    match addr with
    // ... (existing ranges)
    | _ when addr >= 0x8000us && addr <= 0x9FFFus -> state.Vram.[int (addr - 0x
8000us)] <- value // VRAM
    | _ when addr >= 0xFE00us && addr <= 0xFE9Fus -> state.Oam.[int (addr - 0xF
E00us)] <- value // OAM
    | 0xFF40us -> state.Lcdc <- value
    | 0xFF41us -> state.Stat <- value // STAT register. Only bits 3-6 are
writable by CPU.
    | 0xFF42us -> state.Scy <- value
    | 0xFF43us -> state.Scx <- value
    | 0xFF44us -> state.Ly <- value // LY is read-only for CPU, PPU writes it.
Writing to it resets it to 0.
    | 0xFF45us -> state.Lyc <- value
    | 0xFF46us -> // DMA Transfer
        state.Dma <- value
        // 🧠 Important: DMA transfer is complex. For now, we'll just store
the value.
        // A full DMA implementation would block the CPU for 160 cycles and
copy data from (value * 0x100) to OAM.
    | 0xFF47us -> state.Bgp <- value
    | 0xFF48us -> state.Obp0 <- value
    | 0xFF49us -> state.Obp1 <- value
    | 0xFF0Fus -> state.If <- value // Interrupt Flag
    | 0xFFFFus -> state.Ie <- value // Interrupt Enable
    | _ ->
        // ... (existing default write)
        state.Hram.[int (addr - 0xFF80us)] <- value // HRAM
```

Explanation: - The `writeByte` function is similarly updated to allow the CPU to write to these PPU registers. - **`LY` write:** Writing to `0xFF44` (LY) typically resets it to 0. We'll handle this in the `writeByte` function. - **`STAT` write:** Only specific bits of `STAT` are writable by the CPU. For simplicity, we'll allow full writes for now, but

a production emulator would mask these. - **`DMA`:** Direct Memory Access (DMA) to OAM is a critical feature for sprites. When the CPU writes to `0xFF46`, a block of memory is transferred to OAM. This takes 160 machine cycles and freezes the CPU. We'll implement this more fully in a later chapter.

2. Create `Ppu.fs`

Now, let's create the core PPU logic.

File: `src/GameBoy/Ppu.fs`

```
module GameBoy.Ppu

open GameBoy.Memory
open GameBoy.Cpu // Needed for interrupt flags
open System

// PPU Constants
let SCREEN_WIDTH = 160
let SCREEN_HEIGHT = 144
let VBLANK_SCANLINES = 10 // Scanlines 144-153
let TOTAL_SCANLINES = SCREEN_HEIGHT + VBLANK_SCANLINES // 154
let CPU_CYCLES_PER_SCANLINE = 456 // Roughly 456 CPU cycles per scanline
(including HBlank)

// PPU Modes
type PpuMode =
    | HBlank = 0 // Mode 0
    | VBlank = 1 // Mode 1
    | OAMScan = 2 // Mode 2
    | DrawingPixels = 3 // Mode 3

// PPU State
type PpuState = {
    mutable Lcdc : byte // LCD Control (0xFF40)
    mutable Stat : byte // LCD Status (0xFF41)
    mutable Scy : byte  // Scroll Y (0xFF42)
    mutable Scx : byte  // Scroll X (0xFF43)
    mutable Ly : byte   // LCDC Y-Coordinate (0xFF44)
    mutable Lyc : byte  // LY Compare (0xFF45)
    mutable Bgp : byte  // Background Palette (0xFF47)
    mutable Obp0 : byte // Object Palette 0 (0xFF48)
    mutable Obp1 : byte // Object Palette 1 (0xFF49)

    mutable CyclesThisScanline : int
    mutable CurrentFrameBuffer : byte [] // 160 * 144 * 4 bytes for RGBA
    mutable FrameReady : bool
}

let createPpuState () = {
    Lcdc = 0x91uy
    Stat = 0x00uy
    Ly = 0x00uy
    Lyc = 0x00uy
    Scy = 0x00uy
    Scx = 0x00uy
    Bgp = 0xFCuy // Default palette: 0xFF, 0xAA, 0x55, 0x00
    Obp0 = 0xFFuy
    Obp1 = 0xFFuy
    CyclesThisScanline = 0
    CurrentFrameBuffer = Array.zeroCreate (SCREEN_WIDTH * SCREEN_HEIGHT *
4) // RGBA
    FrameReady = false
}

// Helper to get pixel color from a 2-bit value and palette
let getPaletteColor (paletteRegister: byte) (pixelValue: byte) =
    let colorIndex = (paletteRegister >>> (int pixelValue * 2)) &&& 0x03uy
    match colorIndex with
    | 0x00uy -> (0xFFuy, 0xFFuy, 0xFFuy, 0xFFuy) // White
    | 0x01uy -> (0xAAuy, 0xAAuy, 0xAAuy, 0xFFuy) // Light Gray
    | 0x02uy -> (0x55uy, 0x55uy, 0x55uy, 0xFFuy) // Dark Gray
```

```
    | _       -> (0x00uy, 0x00uy, 0x00uy, 0xFFuy) // Black

// Function to read a tile's pixel data from VRAM
// tileIndex: 0-255
// tileDataAddress: base address for tile data (0x8000 or 0x8800)
let readTilePixel (mmu: MemoryState) (tileIndex: byte) (tileDataAddress:
uint16) (yInTile: int) (xInTile: int) =
    let baseAddr =
        if tileDataAddress = 0x8000us then
            // Unsigned addressing (0-255)
            uint16 (0x8000 + (int tileIndex * 16))
        else
            // Signed addressing (0-127, -128- -1) mapping to 0-255
            // If tileIndex is 0-127, it maps to 0x8800-0x8FFF.
            // If tileIndex is 128-255 (interpreted as -128 to -1), it maps to
0x9000-0x97FF.
            let signedTileIndex = sbyte tileIndex
            if signedTileIndex >= 0s then
                uint16 (0x8800 + (int signedTileIndex * 16))
            else
                uint16 (0x9000 + ((int signedTileIndex + 256) * 16)) //
Equivalent to (signedTileIndex + 128 + 128) * 16

    let tileLineAddr = baseAddr + uint16 (yInTile * 2)
    let byte1 = Memory.readByte tileLineAddr mmu
    let byte2 = Memory.readByte (tileLineAddr + 1us) mmu

    let bit1 = (byte1 >>> (7 - xInTile)) &&& 0x01uy
    let bit2 = (byte2 >>> (7 - xInTile)) &&& 0x01uy
    (bit2 <<< 1) ||| bit1 // Combine into 2-bit pixel value

// Renders a single scanline to the frame buffer
let renderScanline (ppu: PpuState) (mmu: MemoryState) =
    let bgDisplayEnable = (ppu.Lcdc &&& 0x01uy) = 0x01uy // LCDC Bit 0
    if not bgDisplayEnable then
        // If background display is disabled, clear the scanline to white
        for x = 0 to SCREEN_WIDTH - 1 do
            let pixelIndex = ((int ppu.Ly * SCREEN_WIDTH) + x) * 4
            ppu.CurrentFrameBuffer.[pixelIndex] <- 0xFFuy // R
            ppu.CurrentFrameBuffer.[pixelIndex + 1] <- 0xFFuy // G
            ppu.CurrentFrameBuffer.[pixelIndex + 2] <- 0xFFuy // B
            ppu.CurrentFrameBuffer.[pixelIndex + 3] <- 0xFFuy // A
        ()
    else
        let bgTileMapAddress =
            if (ppu.Lcdc &&& 0x08uy) = 0x08uy then 0x9C00us // LCDC Bit 3:
1=9C00-9FFF, 0=9800-9BFF
            else 0x9800us

        let bgTileDataAddress =
            if (ppu.Lcdc &&& 0x10uy) = 0x10uy then 0x8000us // LCDC Bit 4:
1=8000-8FFF, 0=8800-97FF
            else 0x8800us

        let scrollY = int ppu.Scy
        let scrollX = int ppu.Scx
        let currentLy = int ppu.Ly

        let yInMap = (currentLy + scrollY) %
256 // Wrapped Y coordinate in the 256x256 background map
        let yInTile = yInMap % 8 // Y coordinate within the 8x8 tile
```

```
        for x = 0 to SCREEN_WIDTH - 1 do
            let xInMap = (x + scrollX) % 256 // Wrapped X coordinate in the
256x256 background map
            let xInTile = xInMap % 8 // X coordinate within the 8x8 tile

            let tileMapX = xInMap / 8
            let tileMapY = yInMap / 8

            let tileMapOffset = uint16 (tileMapY * 32 + tileMapX)
            let tileIndex = Memory.readByte (bgTileMapAddress + tileMapOffset)
mmu

            let pixelValue = readTilePixel mmu tileIndex bgTileDataAddress yInT
ile xInTile
            let (r, g, b, a) = getPaletteColor ppu.Bgp pixelValue

            let pixelIndex = ((currentLy * SCREEN_WIDTH) + x) * 4
            ppu.CurrentFrameBuffer.[pixelIndex] <- r
            ppu.CurrentFrameBuffer.[pixelIndex + 1] <- g
            ppu.CurrentFrameBuffer.[pixelIndex + 2] <- b
            ppu.CurrentFrameBuffer.[pixelIndex + 3] <- a

// Main PPU update function, called by the CPU
let updatePpu (cycles: int) (ppu: PpuState) (mmu: MemoryState) =
    ppu.CyclesThisScanline <- ppu.CyclesThisScanline + cycles

    // 🧠 Important: Check LCD enable/disable. If LCD is off, PPU state should
be reset.
    let lcdEnable = (ppu.Lcdc &&& 0x80uy) = 0x80uy // LCDC Bit 7
    if not lcdEnable then
        ppu.CyclesThisScanline <- 0
        ppu.Ly <- 0x00uy
        // Reset PPU mode to HBlank if LCD is off
        ppu.Stat <- (ppu.Stat &&& 0xF8uy) ||| byte
PpuMode.HBlank // Clear bits 0-2 (mode) and set to HBlank
        ppu.FrameReady <- false
        ()
    else
        // Get current PPU mode from STAT register
        let currentMode = (ppu.Stat &&& 0x03uy) |> enum<PpuMode>

        match currentMode with
        | OAMScan ->
            if ppu.CyclesThisScanline >= 80 then
                ppu.CyclesThisScanline <- ppu.CyclesThisScanline - 80
                ppu.Stat <- (ppu.Stat &&& 0xF8uy) ||| byte PpuMode.DrawingPixel
s // Change mode to Drawing Pixels
        | DrawingPixels ->
            if ppu.CyclesThisScanline >= 172 then // Minimum cycles for
DrawingPixels
                ppu.CyclesThisScanline <- ppu.CyclesThisScanline - 172 // Use
minimum for simplicity, actual is variable
                renderScanline ppu mmu // Render the completed scanline
                ppu.Stat <- (ppu.Stat &&& 0xF8uy) ||| byte PpuMode.HBlank //
Change mode to HBlank
                // Check and trigger HBlank interrupt if enabled (STAT bit 3)
                if (ppu.Stat &&& 0x08uy) = 0x08uy then
                    mmu.If <- mmu.If ||| byte Cpu.InterruptType.LcdStat
        | HBlank ->
            if ppu.CyclesThisScanline >= CPU_CYCLES_PER_SCANLINE - 80 - 172 the
n // Remaining cycles for HBlank
                ppu.CyclesThisScanline <- 0
```

```
                ppu.Ly <- ppu.Ly + 1uy // Increment scanline

                // Check and trigger LYC=LY interrupt if enabled (STAT bit 6)
                if ppu.Ly = ppu.Lyc && (ppu.Stat &&& 0x40uy) = 0x40uy then
                    mmu.If <- mmu.If ||| byte Cpu.InterruptType.LcdStat

                if ppu.Ly >= byte SCREEN_HEIGHT then
                    // Enter VBlank
                    ppu.Stat <- (ppu.Stat &&& 0xF8uy) ||| byte PpuMode.VBlank
                    mmu.If <- mmu.If ||| byte Cpu.InterruptType.VBlank //
Trigger VBlank interrupt
                    // Check and trigger VBlank interrupt if enabled (STAT bit
4)
                    if (ppu.Stat &&& 0x10uy) = 0x10uy then
                        mmu.If <- mmu.If ||| byte Cpu.InterruptType.LcdStat
                    ppu.FrameReady <- true // A full frame is ready
                else
                    // Back to OAM Scan for next scanline
                    ppu.Stat <- (ppu.Stat &&& 0xF8uy) ||| byte PpuMode.OAMScan
                    // Check and trigger OAM interrupt if enabled (STAT bit 5)
                    if (ppu.Stat &&& 0x20uy) = 0x20uy then
                        mmu.If <- mmu.If ||| byte Cpu.InterruptType.LcdStat
        | VBlank ->
            if ppu.CyclesThisScanline >= CPU_CYCLES_PER_SCANLINE then
                ppu.CyclesThisScanline <- ppu.CyclesThisScanline - CPU_CYCLES_P
ER_SCANLINE
                ppu.Ly <- ppu.Ly + 1uy

                // Check and trigger LYC=LY interrupt if enabled (STAT bit 6)
                if ppu.Ly = ppu.Lyc && (ppu.Stat &&& 0x40uy) = 0x40uy then
                    mmu.If <- mmu.If ||| byte Cpu.InterruptType.LcdStat

                if ppu.Ly >= byte TOTAL_SCANLINES then
                    // End of VBlank, reset to scanline 0, OAM Scan mode
                    ppu.Ly <- 0x00uy
                    ppu.Stat <- (ppu.Stat &&& 0xF8uy) ||| byte PpuMode.OAMScan
                    // Check and trigger OAM interrupt if enabled (STAT bit 5)
                    if (ppu.Stat &&& 0x20uy) = 0x20uy then
                        mmu.If <- mmu.If ||| byte Cpu.InterruptType.LcdStat

    // Update LY and STAT registers in MMU for CPU visibility
    mmu.Ly <- ppu.Ly
    mmu.Stat <- ppu.Stat
```

Explanation: - **`PpuState` and `createPpuState`:** Defines the PPU's internal mutable state and provides an initialization function. We use a `byte array` for `CurrentFrameBuffer` to store RGBA pixel data. - **`getPaletteColor`:** A helper function that takes a palette register value (`BGP` for background) and a 2-bit pixel value, then returns the corresponding RGBA color tuple. This simplifies color mapping. - **`readTilePixel`:** This function is crucial. It takes the `MemoryState` (to access VRAM), a `tileIndex`, the `tileDataAddress` (either `0x8000` or `0x8800` depending on LCDC bit 4), and the `(xInTile, yInTile)` coordinates. It then fetches the two bytes that define the pixel row, extracts the 2-bit pixel value, and returns it. The `tileDataAddress` logic handles the two different tile data addressing modes. - **`renderScanline`:** This is the heart of background rendering.

- It checks `LCDC` bit 0 to see if the background display is enabled. If not, it clears the current scanline to white. - It determines which tile map (`0x9800` or `0x9C00`) and tile data region (`0x8000` or `0x8800`) to use based on `LCDC` bits 3 and 4, respectively. - It calculates the `(xInMap, yInMap)` coordinates, taking `SCX` and `SCY` (scroll registers) into account, and wrapping around the 256x256 background map. - It then determines the `tileMapX` and `tileMapY` (which 8x8 tile in the map) and `xInTile`, `yInTile` (pixel within that tile). - It reads the `tileIndex` from the active tile map. - It calls `readTilePixel` to get the 2-bit pixel value. - It uses `getPaletteColor` to convert the pixel value into an RGBA color. - Finally, it writes the RGBA bytes into the `ppu.CurrentFrameBuffer` at the correct `(x, y)` position. - **`updatePpu`:** This function simulates the PPU's internal clock and mode transitions. - It accumulates CPU cycles. - It checks `LCDC` bit 7 to see if the LCD is enabled. If not, it resets the PPU state. - It progresses through `OAMScan`, `DrawingPixels`, `HBlank`, and `VBlank` modes based on elapsed cycles. - When `DrawingPixels` completes, it calls `renderScanline`. - When `HBlank` completes, it increments `ppu.Ly`. - When `ppu.Ly` reaches `SCREEN_HEIGHT` (144), it enters `VBlank` and sets `ppu.FrameReady` to `true`. - It also handles triggering `VBlank` and `LCDStat` interrupts by setting bits in `mmu.If`. - Crucially, it updates `mmu.Ly` and `mmu.Stat` so the CPU can read the current PPU state.

3. Integrate PPU into `Emulator.fs`

Now we need to wire the PPU into our main emulator loop and set up SDL2-CS for display.

File: `src/GameBoy/Emulator.fs`

```
module GameBoy.Emulator

open GameBoy.Cpu
open GameBoy.Memory
open GameBoy.Ppu
open SDL2
open System

// Define the main emulator state
type EmulatorState = {
    mutable Cpu : CpuState
    mutable Mmu : MemoryState
    mutable Ppu : PpuState
    mutable MasterInterruptEnable : bool
    mutable TotalCycles : int64}

let createEmulatorState (bootRomData: byte []) (gameRomData: byte []) = {
    Cpu = createCpuState()
    Mmu = createMemoryState gameRomData // Pass game ROM data
    Ppu = createPpuState()
    MasterInterruptEnable = false
    TotalCycles = 0
}

// ⚡ Quick Note: For simplicity, we'll load the boot ROM into memory here.
// In a real scenario, the boot ROM might be loaded into a separate ROM chip
region.
let loadBootRom (bootRomData: byte []) (mmu: MemoryState) =
    for i = 0 to bootRomData.Length - 1 do
        mmu.Rom0.[i] <- bootRomData.[i]

// Main emulation loop function
let runEmulator (emulatorState: EmulatorState) (bootRomData: byte []) =
    // Initialize SDL
    if SDL.SDL_Init(SDL.SDL_INIT_VIDEO) < 0 then
        failwithf "Could not initialize SDL: %s" (SDL.SDL_GetError())

    let window = SDL.SDL_CreateWindow("F# Game Boy Emulator",
                                      SDL.SDL_WINDOWPOS_CENTERED, SDL.SDL_WINDO
WPOS_CENTERED,
                                      SCREEN_WIDTH, SCREEN_HEIGHT,
                                      SDL.SDL_WindowFlags.SDL_WINDOW_SHOWN)
    if window = NativePtr.zero then
        failwithf "Could not create window: %s" (SDL.SDL_GetError())

    let renderer = SDL.SDL_CreateRenderer(window, -1, SDL.SDL_RendererFlags.SDL
_RENDERER_ACCELERATED)
    if renderer = NativePtr.zero then
        failwithf "Could not create renderer: %s" (SDL.SDL_GetError())

    let texture = SDL.SDL_CreateTexture(renderer,
                                        SDL.SDL_PIXELFORMAT_RGBA32,
                                        int SDL.SDL_TextureAccess.SDL_TEXTUREAC
CESS_STREAMING,
                                        SCREEN_WIDTH, SCREEN_HEIGHT)
    if texture = NativePtr.zero then
        failwithf "Could not create texture: %s" (SDL.SDL_GetError())

    // Load boot ROM if provided
    if bootRomData.Length > 0 then
        loadBootRom bootRomData emulatorState.Mmu
```

```
    let mutable running = true
    let mutable event = SDL.SDL_Event()

    while running do
        // Handle events (e.g., window close)
        while SDL.SDL_PollEvent(&event) = 1 do
            match event.type with
            | SDL.SDL_EventType.SDL_QUIT -> running <- false
            | _ -> ()

        // Execute CPU instruction
        let cycles = Cpu.executeInstruction emulatorState.Cpu
emulatorState.Mmu emulatorState.MasterInterruptEnable

        // Update PPU
        Ppu.updatePpu cycles emulatorState.Ppu emulatorState.Mmu

        // Update timers, sound, etc. (future chapters)

        // Handle interrupts
        Cpu.handleInterrupts emulatorState.Cpu emulatorState.Mmu
&emulatorState.MasterInterruptEnable

        // Render if frame is ready
        if emulatorState.Ppu.FrameReady then
            let pixelsPtr = NativePtr.ofNativeInt (NativePtr.sizeOf<byte> *
0) // Placeholder, actual update below
            let pitch = SCREEN_WIDTH * 4 // RGBA
            SDL.SDL_UpdateTexture(texture, NativePtr.zero, emulatorState.Ppu.Cu
rrentFrameBuffer, pitch) |> ignore

            SDL.SDL_RenderClear(renderer) |> ignore
            SDL.SDL_RenderCopy(renderer, texture, NativePtr.zero, NativePtr.zer
o) |> ignore
            SDL.SDL_RenderPresent(renderer)

            emulatorState.Ppu.FrameReady <- false // Reset flag

        emulatorState.TotalCycles <- emulatorState.TotalCycles + cycles

    // Clean up SDL
    SDL.SDL_DestroyTexture(texture)
    SDL.SDL_DestroyRenderer(renderer)
    SDL.SDL_DestroyWindow(window)
    SDL.SDL_Quit()
```

Explanation: - **`EmulatorState`:** We add a `Ppu` field to hold the `PpuState`. - **`createEmulatorState`:** Initializes the `PpuState` when the emulator is created. - **SDL Initialization:** The `runEmulator` function now includes boilerplate for initializing SDL, creating a window, a renderer, and a texture. - `SDL.SDL_Init(SDL.SDL_INIT_VIDEO)`: Initializes the video subsystem. - `SDL.SDL_CreateWindow`: Creates the display window. - `SDL.SDL_CreateRenderer`: Creates a 2D rendering context for the window. - `SDL.SDL_CreateTexture`: Creates a texture that we can update with our PPU's pixel data. We specify `SDL_PIXELFORMAT_RGBA32` to match our `byte[]` frame

buffer. - **`updatePpu` Call:** After executing CPU instructions, we call `Ppu.updatePpu` with the cycles consumed. This drives the PPU's internal clock. - **Rendering `CurrentFrameBuffer`:** - When `ppu.FrameReady` is `true` (indicating a full frame has been rendered by the PPU), we use `SDL.SDL_UpdateTexture` to copy our `ppu.CurrentFrameBuffer` data to the SDL texture. - `SDL.SDL_RenderClear`, `SDL.SDL_RenderCopy`, and `SDL.SDL_RenderPresent` then clear the renderer, copy the texture to it, and display it on the screen. - `ppu.FrameReady` is reset to `false` until the next frame is complete. - **SDL Cleanup:** Proper cleanup of SDL resources is added.

4. Project File (`.fsproj`) Updates

To use SDL2-CS, you need to add the NuGet package.

File: `src/GameBoy/GameBoy.fsproj`

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <WarnOn>FS3390;NU1605</WarnOn>
    <RootNamespace>GameBoy</RootNamespace>
    <GenerateProgramFile>false</GenerateProgramFile>
  </PropertyGroup>

  <ItemGroup>
    <Compile Include="Cpu.fs" />
    <Compile Include="Memory.fs" />
    <Compile Include="Ppu.fs" />
    <Compile Include="Emulator.fs" />
    <Compile Include="Program.fs" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="SDL2-CS" Version="2.28.0" />
  </ItemGroup>

</Project>
```

Explanation: - We've added a `<PackageReference>` for `SDL2-CS` version `2.28.0`. This is the latest stable version as of 2026-05-05. - Ensure `Ppu.fs` is listed before `Emulator.fs` in the `ItemGroup` to maintain correct compilation order.

Installation of Native SDL2 Libraries: For `SDL2-CS` to work, you need the native SDL2 runtime libraries. - **Windows:** Download `SDL2.dll` from the official SDL website (https://libsdl.org/download-2.0.php) and place it in your project's `bin/Debug/net8.0` (or `bin/Release/net8.0`) directory alongside your executable. - **macOS:** Install via Homebrew: `brew install sdl2`. - **Linux:** Install

via package manager: `sudo apt-get install libsdl2-2.0-0` (Debian/Ubuntu) or `sudo yum install SDL2` (Fedora/RHEL).

Testing & Verification

With the PPU's background rendering implemented, we can now see visual output.

1. **Prepare a ROM:**
    - Find a simple Game Boy ROM that displays a static background, such as a "hello world" test ROM or the official Game Boy boot ROM (if you've implemented its loading). Blargg's CPU test ROMs often have simple backgrounds.
    - Let's assume you have a `test.gb` file.
2. **Update `Program.fs` to load the ROM:**

    File: `src/GameBoy/Program.fs`

    ```fsharp module GameBoy.Program

    open GameBoy.Emulator open System.IO

    [] let main argv = let bootRomPath = "bootrom.bin" // Path to your boot ROM (optional) let gameRomPath = "test.gb" // Path to your Game Boy ROM

    ```

```
let bootRomData =
    if File.Exists(bootRomPath) then
        File.ReadAllBytes(bootRomPath)
    else
        printfn "Boot ROM not found at %s. Continuing without it."
bootRomPath
        Array.empty<byte>

let gameRomData =
    if File.Exists(gameRomPath) then
        File.ReadAllBytes(gameRomPath)
    else
        failwithf "Game ROM not found at %s" gameRomPath

let emulator = createEmulatorState bootRomData gameRomData

printfn "Starting Game Boy emulator..."
runEmulator emulator bootRomData // Pass boot ROM data to runEmulator
printfn "Emulator stopped."

0 // Return 0 for success
```

    ```
    ```

3. **Run the Emulator:** Navigate to your `src/GameBoy` directory in the terminal and run: `bash dotnet run`

Expected Behavior: - A new window titled "F# Game Boy Emulator" should appear. - If you're running a simple test ROM, you should see static background graphics. For instance, the Game Boy boot ROM will show the Nintendo logo (though the scrolling is not yet implemented, so it might appear static or partially rendered). - If you run a game ROM, you might see the initial background of the game, possibly without sprites or correct scrolling.

Quick Debugging Checks: - **Blank Window:** - Check if `SDL2.dll` (or equivalent) is in the correct directory. - Ensure `LCDC` bit 7 is set to enable the LCD. - Verify `renderScanline` is actually being called. - Check for SDL errors in the console. - **Scrambled Graphics:** - Incorrect `readTilePixel` logic, especially the `tileDataAddress` and bit manipulation. - Wrong `BGP` palette mapping. - Incorrect `bgTileMapAddress` or `bgTileDataAddress` selection based on `LCDC`. - **No Background:** - Check if `LCDC` bit 0 (background enable) is set. - Ensure `VRAM` is being loaded correctly by the ROM and accessed by the PPU.

Production Considerations

Performance

The PPU's pixel-by-pixel rendering is a performance-critical area. Our current `renderScanline` function does a lot of work for each pixel. - **Caching:** Tile data (the 8x8 pixel patterns) rarely changes. We could pre-render each 8x8 tile into a small pixel buffer once it's written to VRAM and then just copy from these cached tile buffers during `renderScanline`. - **Batching:** Instead of setting each pixel individually, we could potentially use SDL's `SDL_RenderDrawPoints` or similar functions if we pre-calculate a scanline's worth of pixels. - **JIT Optimization:** F# and .NET's JIT compiler are very good, but tight loops like pixel rendering can still be bottlenecks. Profile your application to identify hot spots.

Synchronization

Accurate timing between the CPU and PPU is paramount. - **Cycle Counting:** We're currently passing CPU cycles directly to `updatePpu`. This is a good start. Ensure that all CPU instructions correctly report their cycle counts. - **PPU Cycle Accuracy:** The PPU's mode transitions and `LY` increments need to happen at precise cycle counts. Slight inaccuracies here can lead to visual glitches, screen tearing, or incorrect interrupt timings. The cycle counts used in `updatePpu` are

approximations; consult Game Boy PPU documentation for exact timings per mode.

Maintainability

The PPU code will grow. - **Modularity:** Keep functions small and focused (e.g., `readTilePixel`, `renderBackgroundPixel`, `renderSpritePixel`). - **Clear Register Mapping:** Use named constants or discriminated unions for register bits where appropriate to make `Lcdc &&& 0x08uy` more readable.

Common Issues & Solutions

1. **Issue: Blank screen or erratic display behavior.**

- **Cause:** The LCD is likely disabled, or PPU timing is completely off.
- **Solution:** * Verify `LCDC` bit 7 (`0x80uy`) is set. The boot ROM usually sets this. * Log the `ppu.Ly` and `ppu.Stat` values. `LY` should increment from 0 to 153. `STAT` mode bits (0-1) should cycle through 2, 3, 0, then 1 for VBlank. * Ensure `updatePpu` is called with the correct number of CPU cycles.

1. **Issue: Background is displayed, but colors are wrong or patterns are distorted.**

- **Cause:** Incorrect palette mapping or misinterpretation of tile data bits.
- **Solution:** * Double-check `getPaletteColor` logic. `BGP` register bits define the colors. * Review `readTilePixel` bit manipulation. The Game Boy uses two bytes per tile row, where bit 0 of each byte forms the LSB of the pixel color, and bit 1 forms the MSB. Ensure the combining `(bit2 <<< 1) ||| bit1` is correct. * Verify `bgTileDataAddress` and `bgTileMapAddress` selection based on `LCDC` bits.

1. **Issue: `SDL2-CS` fails to initialize or window doesn't appear.**

- **Cause:** Native SDL2 library not found or installed incorrectly.
- **Solution:**
- **Windows:** Ensure `SDL2.dll` is in the executable's directory (`bin/Debug/net8.0`).
- **macOS:** Run `brew install sdl2`.
- **Linux:** Run `sudo apt-get install libsdl2-2.0-0` or equivalent for your distribution. * Check the error message from `SDL_GetError()`.

Summary & Next Step

In this chapter, we've laid the critical groundwork for the Game Boy's visual output. We've: - Integrated PPU registers into our `MMU`. - Designed and implemented the core `PpuState` and its update logic. - Developed functions to read tile data and render the background layer pixel by pixel. - Set up `SDL2-CS` to display our PPU's frame buffer on screen.

By now, you should be able to load a simple ROM and see its background displayed, albeit without sprites or complex scrolling. This is a huge step towards a fully functional emulator!

Next, we'll dive into the more dynamic aspects of the PPU: rendering sprites, handling LCD control register details, and implementing more accurate PPU timing and interrupts.

🧠 Check Your Understanding

- What are the four main PPU modes, and what is the primary activity in each?
- How does the `LCDC` register influence which tile map and tile data region are used for background rendering?
- Explain the purpose of the `SCY` and `SCX` registers in background rendering.

⚡ Mini Task

- Modify the `getPaletteColor` function to use a custom set of colors (e.g., shades of green for a classic Game Boy feel) instead of grayscale.
- Add logging to print the `LY` register value every time it increments, and observe its cycle from 0 to 153.

🚀 Scenario

You're running a complex Game Boy game, and the background appears to "tear" horizontally, with parts of the image shifted. Upon closer inspection, you notice the tearing occurs at seemingly random scanlines. What are the most likely causes for this issue, and where would you begin debugging in your PPU implementation? Consider CPU-PPU synchronization and interrupt handling.

📌 TL;DR

- The Game Boy PPU draws a 160x144 display by rendering scanlines in modes: OAM Scan, Drawing Pixels, HBlank, and VBlank.
- VRAM (`0x8000-0x9FFF`) stores tile data (pixel patterns) and tile maps (indices to tiles for background).
- PPU I/O registers like `LCDC`, `STAT`, `SCY`, `SCX`, `LY`, `LYC`, and `BGP` control display features and status.
- Background rendering involves reading `SCY`/`SCX`, determining tile map/data regions from `LCDC`, fetching tile indices, and then translating 2-bit pixel values to colors using `BGP`.
- `SDL2-CS` is used for displaying the PPU's `CurrentFrameBuffer` to a window.

🧠 Core Flow

1. CPU executes instructions, returning cycle count.
2. `Ppu.updatePpu` accumulates cycles, advances PPU mode, and increments `LY`.
3. During "Drawing Pixels" mode, `renderScanline` fetches tile data from VRAM via `MMU`, applies scrolling from `SCX`/`SCY`, and draws pixels to `CurrentFrameBuffer`.
4. Upon entering VBlank (after `LY` reaches 144), `ppu.FrameReady` is set to `true`.
5. In `Emulator.runEmulator`, if `ppu.FrameReady` is true, the `CurrentFrameBuffer` is copied to an SDL texture and rendered to the screen.

🚀 Key Takeaway

Emulating graphics hardware like the PPU demands meticulous attention to hardware specifications and accurate timing. The iterative process of accumulating CPU cycles, transitioning PPU modes, and carefully mapping memory-backed registers to visual output is fundamental to bringing an emulator to life.

References

- Pan Docs - The Ultimate Game Boy Technical Manual
- F# Language Reference
- SDL2-CS NuGet Package
- SDL Documentation

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 08

Picture Processing Unit (PPU) Part 2: Sprites, Scrolling, and LCD Control

This chapter builds upon our foundational Picture Processing Unit (PPU) work, where we established background tile rendering. Now, we'll introduce the dynamic elements that bring games to life: sprites (movable objects), background scrolling, and the crucial LCD Control Register, which dictates how the display operates. By the end of this milestone, your emulator will be able to render basic sprites, scroll the background, and respond to fundamental display settings, making it capable of running more visually complex Game Boy ROMs.

Understanding these components is vital because they represent the core mechanisms by which games animate characters, display scores, and create immersive environments. Incorrect implementation of any of these can lead to visual glitches, tearing, or even unplayable games. We'll focus on accurately reflecting the Game Boy's hardware behavior using F#'s expressive types.

Project Overview

This project is about building a Game Boy emulator from scratch in F#. We're taking a ground-up approach to understand low-level system design and computer architecture. Each chapter focuses on emulating a specific hardware component, verifying its behavior, and integrating it into the larger system. This chapter specifically enhances the visual output, moving beyond static backgrounds to dynamic game elements.

Tech Stack

For this chapter, we continue to leverage:

- **F# (version 8.0):** The primary language for its strong type system, functional paradigms, and conciseness, which helps in modeling hardware states immutably.
- **.NET SDK (version 8.0):** Provides the runtime and tooling for F# development, ensuring cross-platform compatibility.

- **Silk.NET.Windowing (version 2.19.0):** A modern, cross-platform library used for creating windows and handling graphics, serving as our display layer. It is a robust alternative to older SDL.NET bindings.

We aim to use the latest stable releases as of 2026-05-05 to ensure we're building with current best practices.

Build Plan

To achieve dynamic visuals, our plan for this chapter is structured into several key milestones:

1. **Define Sprite Structure:** Model Game Boy sprites and their attributes using F# records.
2. **Extend PPU State:** Incorporate new registers like `LCDC`, `SCX`, `SCY`, `OBP0`, `OBP1` into our existing `PPU` record.
3. **Implement MMU Register Access:** Add logic to the `MMU` to correctly handle CPU reads and writes to these new PPU registers, including critical access restrictions.
4. **Refine PPU Timing and Modes:** Update `stepPPU` to respond to `LCDC` enable/disable, `LYC=LY` coincidence, and `STAT` register interrupt flags.
5. **Enhance `drawScanline` for Background Scrolling:** Implement `SCX` and `SCY` offsets to accurately render the scrolled background.
6. **Implement Sprite Rendering:**
    - Scan Object Attribute Memory (OAM) for sprites visible on the current scanline.
    - Sort sprites based on Game Boy's priority rules (X-coordinate, OAM index).
    - Render each sprite, applying its attributes (flips, palettes, background priority).

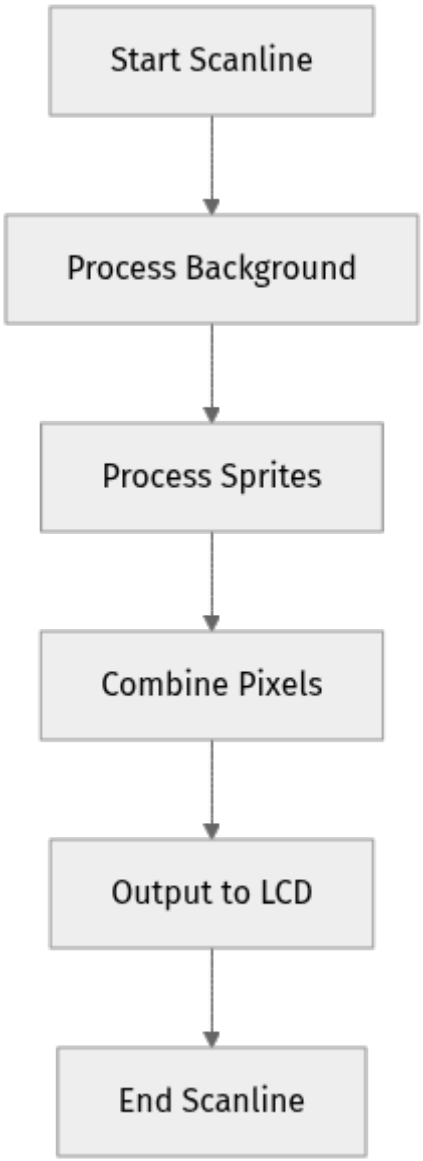
Architecture

The Game Boy's PPU operates as a state machine, processing scanlines sequentially. Our F# implementation models this by updating an immutable `PPU` state within the `MMU` (Memory Management Unit). The `CPU` interacts with the `PPU` by reading and writing to its memory-mapped registers via the `MMU`.

PPU Rendering Flow with Sprites and Scrolling

The core rendering loop, which processes one scanline at a time, will be enhanced. For each pixel on a scanline:

- First, determine the background pixel, accounting for `SCX` and `SCY`.
- Then, identify relevant sprites that overlap with the current scanline.
- For each relevant sprite, calculate its pixel.
- Finally, combine the background and sprite pixels based on sprite priority rules (X-coordinate, OAM index, and background priority bit).



🧠 `Important:` The Game Boy processes pixels from left to right for each scanline. Sprite rendering needs to consider this order, along with their X-coordinates and other attributes, to correctly layer them over the background.

Data Structures and Register Mapping

We'll extend our `PPU` state to include the new registers and a way to represent sprite data.

- **PPU State:** Add `scx`, `scy`, `lcdc`, `stat`, `lyc`, `obp0`, `obp1`, `dma` and `oam` fields.
- **Sprite Structure:** A record to hold parsed OAM data for each sprite.
- **OAM (Object Attribute Memory):** Located at `0xFE00-0xFE9F`, this 160-byte region holds attributes for up to 40 sprites. Each sprite uses 4 bytes.
    - Byte 0: Y-position (minus 16)
    - Byte 1: X-position (minus 8)
    - Byte 2: Tile Index (0-255)
    - Byte 3: Attributes (Palette, X/Y Flip, BG Priority, CGB Palette)

Register Overview

- **LCDC (LCD Control):** `0xFF40`
    - Bit 7: LCD and PPU enable (0=Off, 1=On)
    - Bit 6: Window Tile Map Display Select (0=9800-9BFF, 1=9C00-9FFF)
    - Bit 5: Window Display Enable (0=Off, 1=On)
    - Bit 4: BG & Window Tile Data Select (0=8800-97FF, 1=8000-8FFF)
    - Bit 3: BG Tile Map Display Select (0=9800-9BFF, 1=9C00-9FFF)
    - Bit 2: OBJ (Sprite) Size (0=8x8, 1=8x16)
    - Bit 1: OBJ (Sprite) Display Enable (0=Off, 1=On)
    - Bit 0: BG and Window Display Enable (0=Off, 1=On)
- **SCY (Scroll Y):** `0xFF42` (8-bit) - Y position of the top-left pixel of the background map to be displayed.
- **SCX (Scroll X):** `0xFF43` (8-bit) - X position of the top-left pixel of the background map to be displayed.
- **STAT (LCD Status):** `0xFF41` - Contains PPU mode, LYC=LY coincidence flag, and interrupt enable flags.
- **LYC (LY Compare):** `0xFF45` - Compared against `LY` (current scanline).
- **OBP0/OBP1 (Object Palettes):** `0xFF48`, `0xFF49` - Define colors for sprites.

- **DMA (DMA Transfer):** `0xFF46` - Triggers a transfer from main memory to OAM.

Step-by-Step Implementation

We'll start by defining our sprite structure and then integrate sprite rendering, scrolling, and LCD control into our existing PPU logic.

1. Define `PpuMode` and `Sprite` Structure

First, ensure `PpuMode` is explicitly defined with the correct Game Boy hardware values, typically in `Types.fs`. Then, update `PPU.fs` with the `Sprite` record and the extended `PPU` state.

```
// src/GameBoyEmulator/Types.fs

module GameBoyEmulator.Types

// ... (existing types like PixelColor, ColorPalette)

/// Game Boy PPU Modes as defined by hardware (STAT register bits 0-1)
type PpuMode =
    | HBlank = 0x00
    | VBlank = 0x01
    | OamScan = 0x02
    | Drawing = 0x03

// ... (other shared types)
```

Explanation: The `PpuMode` discriminated union maps directly to the 2-bit mode field in the `STAT` register, providing clear, type-safe representation of the PPU's operational state.

Now, let's define the `Sprite` record and update the `PPU` state.

```
// src/GameBoyEmulator/PPU.fs

module GameBoyEmulator.PPU

open System
open GameBoyEmulator.MMU // Assuming MMU is needed for memory access
open GameBoyEmulator.Types // Assuming shared types like PixelColor,
ColorPalette, PpuMode

// ... (existing types like PixelColor, ColorPalette, Tile)

/// Represents a single Game Boy sprite (Object)
type Sprite =
    { Y          : byte
      X          : byte
      TileIndex  : byte
      PaletteNum : byte // DMG palette is 0 or 1 (OBP0 or OBP1)
      XFlip      : bool
      YFlip      : bool
      BgPriority : bool // True if sprite is behind BG/Window (color 1-3)
      OamIndex   : int  // Original index in OAM for priority tie-breaking
    }

/// PPU state, updated with new registers
type PPU =
    { scanline  : int // Current PPU scanline (0-153)
      cycles    : int // PPU cycles accumulated for current mode
      mode      : PpuMode // Current PPU operating mode
      lcdBuffer : PixelColor array // The frame buffer (160 * 144 pixels)
      vram      : byte array // Video RAM (0x8000-0x9FFF)
      // New PPU registers
      lcdc      : byte // LCD Control Register (0xFF40)
      stat      : byte // LCD Status Register (0xFF41)
      scy       : byte // Scroll Y Register (0xFF42)
      scx       : byte // Scroll X Register (0xFF43)
      ly        : byte // LCD Y-Coordinate Register (0xFF44), already present
      lyc       : byte // LY Compare Register (0xFF45)
      bgp       : byte // BG Palette Data (0xFF47), already present
      obp0      : byte // OBJ Palette 0 Data (0xFF48)
      obp1      : byte // OBJ Palette 1 Data (0xFF49)
      dma       : byte // DMA Transfer and Start Address Register (0xFF46) -
will be handled by MMU
      mutable oam : byte array // Object Attribute Memory (0xFE00-0xFE9F)
      // ... other existing fields
    }

// ... (existing functions)
```

Explanation:

- `Sprite` record: Directly mirrors the 4-byte OAM entry, but parsed into more readable fields. `OamIndex` is added for priority resolution, as sprites with the same X-coordinate are prioritized by their OAM index.
- `PPU` record: We've added `lcdc`, `stat`, `scy`, `scx`, `lyc`, `obp0`, `obp1`, `dma`, and `oam` to the PPU's internal state. While `dma` is primarily handled by the MMU, the PPU needs access to `oam` for sprite data. We make `oam` `mutable`

because DMA transfers directly into it, and the CPU can write to it, which is a common pattern for hardware registers in an emulator.

2. Initialize PPU State

Update your `initPPU` function to set initial values for these new registers. This ensures the PPU starts in a well-defined state, mimicking a real Game Boy's power-on behavior.

```
// src/GameBoyEmulator/PPU.fs

// ... (inside PPU module)

let initPPU (cartridge: Cartridge.Cartridge) : PPU = // Assuming cartridge is passed for completeness
    { scanline = 0
      cycles = 0
      mode = PpuMode.OamScan // Start in OAM Scan mode
      lcdBuffer = Array.init (160 * 144) (fun _ -> White) // Clear screen to white
      vram = Array.zeroCreate 0x2000 // 8KB VRAM
      lcdc = 0x91uy // Default power-on value for LCDC
      stat = 0x80uy // Default power-on value for STAT (Bit 7 always 1)
      scy = 0x00uy
      scx = 0x00uy
      lyc = 0x00uy
      ly = 0x00uy
      bgp = 0xFCuy // Default palette (White, LightGray, DarkGray, Black)
      obp0 = 0xFFuy // Default sprite palette 0
      obp1 = 0xFFuy // Default sprite palette 1
      dma = 0x00uy
      oam = Array.zeroCreate 0xA0 // 160 bytes for OAM (40 sprites * 4 bytes)
      // ... other existing fields
    }
```

Explanation:

- `lcdc = 0x91uy`: This is the typical power-on value for the Game Boy's LCDC register, meaning LCD is on, BG/Window display is on, 8x8 sprites, etc.
- `stat = 0x80uy`: Bit 7 is always 1 as per hardware spec.
- `oam = Array.zeroCreate 0xA0`: Allocates 160 bytes for OAM.

3. Implement Register Read/Write in MMU

The CPU interacts with PPU registers through the MMU. We need to add logic to `MMU.fs` to handle reads and writes to `0xFF40-0xFF49`. This is a critical step for the CPU to be able to control the PPU.

```
// src/GameBoyEmulator/MMU.fs

module GameBoyEmulator.MMU

open System
open GameBoyEmulator.Types
open GameBoyEmulator.PPU
open GameBoyEmulator.Cartridge

// ... (existing types like MMU)

let writeByte (mmu: MMU) (addr: word) (value: byte) : MMU =
    match addr with
    // ... existing memory regions
    | addr when addr >= 0x8000us && addr <= 0x9FFFus -> // VRAM
        // CPU can only access VRAM outside PPU drawing mode (Mode 3)
        if mmu.ppu.mode <> PpuMode.Drawing then
            { mmu with ppu = { mmu.ppu with vram = (mmu.ppu.vram |> Array.mapi
(fun i v -> if i = (int addr - 0x8000) then value else v)) } }
        else mmu // Ignore write if PPU is drawing
    | addr when addr >= 0xFE00us && addr <= 0xFE9Fus -> // OAM
        // CPU can only access OAM outside PPU OAM Scan (Mode 2) and Drawing
(Mode 3) modes
        if mmu.ppu.mode <> PpuMode.OamScan && mmu.ppu.mode <> PpuMode.Drawing t
hen
            { mmu with ppu = { mmu.ppu with oam = (mmu.ppu.oam |> Array.mapi (f
un i v -> if i = (int addr - 0xFE00) then value else v)) } }
        else mmu // Ignore write if PPU is scanning OAM or drawing
    | 0xFF40us -> { mmu with ppu = { mmu.ppu with lcdc = value } } // LCDC
    | 0xFF41us -> // STAT (only bits 3-6 writable by CPU, bits 0-2 for mode,
bit 7 for LYC=LY coincidence)
        { mmu with ppu = { mmu.ppu with stat = (value &&& 0x78uy) |||
(mmu.ppu.stat &&& 0x87uy) } }
    | 0xFF42us -> { mmu with ppu = { mmu.ppu with scy = value } } // SCY
    | 0xFF43us -> { mmu with ppu = { mmu.ppu with scx = value } } // SCX
    | 0xFF45us -> { mmu with ppu = { mmu.ppu with lyc = value } } // LYC
    | 0xFF46us -> // DMA Transfer
        // This is a special register. Writing to it triggers a DMA transfer
from (value * 0x100) to OAM
        let sourceAddr = word (value) <<< 8 // Source address is value * 0x100
        let newMmu = { mmu with ppu = { mmu.ppu with dma = value } } // Store
DMA value
        // Perform DMA transfer: copy 160 bytes from sourceAddr to OAM
(0xFE00-0xFE9F)
        let oamData = Array.init 0xA0 (fun i -> newMmu |> readByte (sourceAddr
+ word i))
        { newMmu with ppu = { newMmu.ppu with oam = oamData } }
    | 0xFF47us -> { mmu with ppu = { mmu.ppu with bgp = value } } // BGP
    | 0xFF48us -> { mmu with ppu = { mmu.ppu with obp0 = value } } // OBP0
    | 0xFF49us -> { mmu with ppu = { mmu.ppu with obp1 = value } } // OBP1
    // ... other I/O registers
    | _ -> mmu // Default: no change for unhandled addresses


let readByte (mmu: MMU) (addr: word) : byte =
    match addr with
    // ... existing memory regions
    | 0xFF40us -> mmu.ppu.lcdc
    | 0xFF41us -> // STAT register read. Bits 0-2 contain the current PPU mode.
                  // Bit 7 is always 1. Bits 3-6 are interrupt enable flags.
        let modeBits = byte mmu.ppu.mode // Get the byte value of the current
```

```
PpuMode
        (mmu.ppu.stat &&& 0xFCuy) ||| modeBits // Clear bits 0-1 (mode) and OR
in the current mode
    | 0xFF42us -> mmu.ppu.scy
    | 0xFF43us -> mmu.ppu.scx
    | 0xFF44us -> mmu.ppu.ly // LY is read-only
    | 0xFF45us -> mmu.ppu.lyc
    | 0xFF46us -> mmu.ppu.dma
    | 0xFF47us -> mmu.ppu.bgp
    | 0xFF48us -> mmu.ppu.obp0
    | 0xFF49us -> mmu.ppu.obp1
    | addr when addr >= 0xFE00us && addr <= 0xFE9Fus -> // OAM
        // CPU can only access OAM outside PPU OAM Scan and Drawing modes
        if mmu.ppu.mode <> PpuMode.OamScan && mmu.ppu.mode <> PpuMode.Drawing t
hen
            mmu.ppu.oam.[int addr - 0xFE00]
        else 0xFFuy // Return 0xFF during restricted OAM access
    // ... other I/O registers
    | _ -> // For other addresses, read from internal RAM or cartridge
        // ... existing read logic
        if addr < 0x8000us then mmu.cartridge.readByte addr // Read from ROM
        elif addr >= 0xC000us && addr <= 0xDFFFus then mmu.ram.[int addr] //
WRAM
        elif addr >= 0xE000us && addr <= 0xFDFFus then mmu.ram.[int addr - 0x20
00] // Echo RAM
        elif addr >= 0xFF00us && addr <= 0xFF7Fus then mmu.ram.[int
addr] // IO Registers (some handled above)
        elif addr >= 0xFF80us && addr <= 0xFFFEus then mmu.ram.[int addr] //
HRAM
        else 0xFFuy // Fallback for unhandled addresses or invalid reads
```

Explanation:

- **VRAM/OAM Access Restrictions:** Critical for accuracy! The CPU cannot access VRAM or OAM during certain PPU modes (Drawing, OAM Scan). Writes are ignored, reads return `0xFF`. This prevents visual tearing or glitches that would occur if the CPU modified display data while the PPU was actively reading it.
- **STAT Register (`0xFF41us`):**
- **Write:** Only bits 3-6 are writable by the CPU (interrupt enable flags). Bits 0-2 (mode) and 7 (LYC=LY coincidence flag) are read-only or set by the PPU itself. The write logic masks out the mode and coincidence bits before applying the new value.
- **Read:** The Game Boy's STAT register always has bit 7 set to 1. Bits 0-2 reflect the current PPU mode. The new logic `(mmu.ppu.stat &&& 0xFCuy) ||| (byte mmu.ppu.mode)` correctly combines the stored `stat` flags (bits 2-7) with the current PPU mode (bits 0-1).
- **DMA Transfer (0xFF46):** Writing a value `N` to `0xFF46` initiates a Direct Memory Access transfer. 160 bytes from `N * 0x100` are copied into OAM (`0xFE00-0xFE9F`). This typically takes 160 CPU cycles and halts the CPU,

but for simplicity, we perform the copy instantly for now. A more accurate emulator would simulate the cycle cost and CPU halt.

- **Register Mapping:** Each PPU-related register is now mapped to update or read from the `mmu.ppu` state.

4. Update `stepPPU` for LCDC and LYC

Modify `stepPPU` to check `lcdc` for LCD enable/disable and `lyc` for coincidence. This ensures the PPU correctly responds to CPU commands regarding display state and interrupt conditions.

```
// src/GameBoyEmulator/PPU.fs

// ... (inside PPU module)

let stepPPU (mmu: MMU.MMU) (cycles: int) : MMU.MMU * Interrupt =
    let mutable currentMmu = mmu
    let mutable ppu = currentMmu.ppu
    let mutable interrupt = NoInterrupt

    // Check if LCD is enabled (Bit 7 of LCDC)
    if (ppu.lcdc &&& 0x80uy) = 0x00uy then // LCD is off
        // When LCD is off, PPU is reset
        if ppu.ly <> 0uy || ppu.cycles <> 0 || ppu.mode <> PpuMode.HBlank
then // Only reset if not already in reset state
            ppu <- { ppu with ly = 0uy; cycles = 0; mode = PpuMode.HBlank } //
PPU stays in HBlank (mode 0)
            currentMmu <- { currentMmu with ppu = ppu }
        // No rendering or STAT interrupts when LCD is off
        (currentMmu, NoInterrupt)
    else
        // ... (existing cycle accumulation)
        ppu <- { ppu with cycles = ppu.cycles + cycles }

        // LYC = LY Coincidence Check
        let lycEqLy = ppu.ly = ppu.lyc
        let statCoincidenceBit = if lycEqLy then 0x04uy else
0x00uy // Bit 2 of STAT
        let oldStat = ppu.stat
        ppu <- { ppu with stat = (oldStat &&& 0xFBuy) |||
statCoincidenceBit } // Update bit 2 of STAT (clear then set)

        // If LYC=LY coincidence interrupt is enabled (STAT bit 6) and
coincidence occurs
        if lycEqLy && ((oldStat &&& 0x40uy) <> 0uy) then
            // Only trigger interrupt if the coincidence bit just became true
(edge-triggered)
            if (oldStat &&& 0x04uy) = 0uy then
                interrupt <- Interrupt.set Interrupt.LcdStat interrupt

        // PPU Modes and Line Rendering
        let newPpu, newInterrupt =
            match ppu.mode with
            | PpuMode.OamScan -> // Mode 2: OAM Read (80 cycles)
                if ppu.cycles >= 80 then
                    let nextPpu = { ppu with cycles = ppu.cycles - 80; mode = P
puMode.Drawing }
                    // Trigger STAT interrupt if enabled for Mode 2 (OAM Scan)
                    let currentInterrupt = if ((ppu.stat &&& 0x20uy) <> 0uy) th
en (Interrupt.set Interrupt.LcdStat interrupt) else interrupt
                    (nextPpu, currentInterrupt)
                else (ppu, interrupt)

            | PpuMode.Drawing -> // Mode 3: VRAM Read (172 cycles)
                if ppu.cycles >= 172 then // This is where we'll actually draw
the line
                    let ppuAfterDraw = drawScanline ppu currentMmu.cartridge
                    let nextPpu = { ppuAfterDraw with cycles = ppuAfterDraw.cyc
les - 172; mode = PpuMode.HBlank }
                    // Trigger STAT interrupt if enabled for Mode 3 (Drawing) -
not typical, but possible
                    let currentInterrupt = if ((ppu.stat &&& 0x10uy) <> 0uy) th
```

```
en (Interrupt.set Interrupt.LcdStat interrupt) else interrupt
                    (nextPpu, currentInterrupt)
                else (ppu, interrupt)

            | PpuMode.HBlank -> // Mode 0: H-Blank (204 cycles)
                if ppu.cycles >= 204 then
                    let ppuAfterHBlank = { ppu with cycles = ppu.cycles - 204;
ly = ppu.ly + 1uy }
                    // Trigger STAT interrupt if enabled for Mode 0 (HBlank)
                    let currentInterrupt = if ((ppu.stat &&& 0x08uy) <> 0uy) th
en (Interrupt.set Interrupt.LcdStat interrupt) else interrupt

                    if ppuAfterHBlank.ly = 144uy then
                        // Enter VBlank
                        let ppuVBlank = { ppuAfterHBlank with mode = PpuMode.VB
lank }
                        // Trigger VBlank interrupt
                        let vblankInterrupt = Interrupt.set Interrupt.VBlank cu
rrentInterrupt
                        // Trigger STAT interrupt if enabled for Mode 1
(VBlank)
                        let statVblankInterrupt = if ((ppu.stat &&& 0x10uy) <>
0uy) then (Interrupt.set Interrupt.LcdStat vblankInterrupt) else vblankInterrup
t
                        (ppuVBlank, statVblankInterrupt)
                    else
                        // Next scanline, back to OAM Scan
                        let ppuOamScan = { ppuAfterHBlank with mode = PpuMode.O
amScan }
                        // Trigger STAT interrupt if enabled for Mode 2 (OAM
Scan)
                        let statOamScanInterrupt = if ((ppu.stat &&& 0x20uy)
<> 0uy) then (Interrupt.set Interrupt.LcdStat currentInterrupt) else currentInt
errupt
                        (ppuOamScan, statOamScanInterrupt)
                else (ppu, interrupt)

            | PpuMode.VBlank
-> // Mode 1: V-Blank (4560 cycles total for lines 144-153)
                if ppu.cycles >= 456 then // Each VBlank line takes 456 cycles
                    let ppuVBlankLine = { ppu with cycles = ppu.cycles - 456; l
y = ppu.ly + 1uy }
                    if ppuVBlankLine.ly > 153uy then // Last VBlank line (line
153 ends, next is line 0)
                        let ppuReset = { ppuVBlankLine with ly = 0uy; mode = Pp
uMode.OamScan }
                        // Trigger STAT interrupt if enabled for Mode 2 (OAM
Scan)
                        let statOamScanInterrupt = if ((ppu.stat &&& 0x20uy)
<> 0uy) then (Interrupt.set Interrupt.LcdStat interrupt) else interrupt
                        (ppuReset, statOamScanInterrupt)
                    else (ppuVBlankLine, interrupt) // Continue VBlank lines
                else (ppu, interrupt)

        currentMmu <- { currentMmu with ppu = newPpu }
        (currentMmu, newInterrupt)
```

Explanation:

- **LCD Enable Check:** The first thing `stepPPU` does is check `lcdc` bit 7. If the LCD is off, the PPU halts, `LY` resets to 0, and no rendering or interrupts occur. This is crucial for games that toggle the display.
- **LYC=LY Coincidence:** We now compare `ppu.ly` with `ppu.lyc` and update bit 2 of the `stat` register accordingly. If `stat` bit 6 is also set, and a coincidence just occurred (meaning the `LYC=LY` flag changed from false to true), a `LcdStat` interrupt is requested. This is an edge-triggered interrupt.
- **Mode STAT Interrupts:** Added checks for `stat` bits 3, 4, 5, which enable `LcdStat` interrupts for HBlank, VBlank, and OAM Scan modes respectively. This is vital for timing-sensitive games. The `newPpu, newInterrupt` pattern helps manage state updates and potential interrupt requests more functionally.

5. Implement `drawScanline` with Scrolling and Sprites

This is the core of our PPU update. We'll modify `drawScanline` to incorporate `SCX`, `SCY`, `LCDC` and render sprites. The `readTile` function signature is updated as requested.

```
// src/GameBoyEmulator/PPU.fs

// ... (inside PPU module)

/// Helper to read a palette color from a byte and index
let getPaletteColor (paletteByte: byte) (index: int) : PixelColor =
    let shift = index * 2
    let colorId = (paletteByte >>> shift) &&& 0x03uy
    match colorId with
    | 0uy -> White
    | 1uy -> LightGray
    | 2uy -> DarkGray
    | 3uy -> Black
    | _   -> White // Should not happen


/// Reads a tile from VRAM, considering tile data selection (0x8000-0x8FFF or
0x8800-0x97FF)
let readTile (ppu: PPU) (tileIndex: byte) (yOffset: int) : byte array =
    let signedTileIndex = sbyte tileIndex
    let tileAddress =
        if (ppu.lcdc &&& 0x10uy) = 0x10uy then // LCDC bit 4: BG & Window Tile
Data Select (0=8800-97FF, 1=8000-8FFF)
            // Use 0x8000-0x8FFF range (unsigned addressing)
            0x8000us + word (tileIndex * 16uy)
        else
            // Use 0x8800-0x97FF range (signed addressing), base address is
0x9000
            0x9000us + word (signedTileIndex * 16sbyte)

    let lineAddress = tileAddress + word (byte (yOffset * 2)) // Each line
takes 2 bytes
    let byte1 = ppu.vram.[int (lineAddress - 0x8000us)]
    let byte2 = ppu.vram.[int (lineAddress - 0x8000us + 1us)]
    [| byte1; byte2 |]

/// Extracts pixel data from a tile line (2 bytes)
let getTileLinePixels (tileLineBytes: byte array) : byte array =
    let byte1 = tileLineBytes.[0]
    let byte2 = tileLineBytes.[1]
    Array.init 8 (fun i ->
        let bit1 = (byte1 >>> (7 - i)) &&& 0x01uy
        let bit2 = (byte2 >>> (7 - i)) &&& 0x01uy
        (bit2 <<< 1) ||| bit1 // Combine into 2-bit color index
    )

/// Draws a single scanline, incorporating background scrolling and sprites
let private drawScanline (ppu: PPU) (cartridge: Cartridge.Cartridge) : PPU =
    let ly = int ppu.ly
    let mutable pixels = Array.zeroCreate 160 // 160 pixels for the current
scanline

    // Check LCD Enable (Bit 7 of LCDC)
    if (ppu.lcdc &&& 0x80uy) = 0x00uy then
        // If LCD is off, clear the line to white
        for x = 0 to 159 do
            pixels.[x] <- White
        { ppu with lcdBuffer = ppu.lcdBuffer |> Array.mapi (fun i v -> if i / 1
60 = ly then pixels.[i % 160] else v) }
    else
        // --- Background Rendering ---
```

```
        // Check BG Display Enable (Bit 0 of LCDC)
        let bgDisplayEnabled = (ppu.lcdc &&& 0x01uy) = 0x01uy

        if bgDisplayEnabled then
            let tileMapBaseAddr =
                if (ppu.lcdc &&& 0x08uy) = 0x08uy then // LCDC bit 3: BG Tile
Map Display Select (0=9800-9BFF, 1=9C00-9FFF)
                    0x9C00us // 0x9C00-0x9FFF
                else
                    0x9800us // 0x9800-0x9BFF

            let bgPalette = ppu.bgp

            let scrolledY = (ly + int ppu.scy) &&& 0xFF // Apply SCY and wrap
around 256
            let tileRow = scrolledY / 8

            for x = 0 to 159 do
                let scrolledX = (x + int ppu.scx) &&& 0xFF // Apply SCX and
wrap around 256
                let tileCol = scrolledX / 8

                let tileMapAddr = tileMapBaseAddr + word (byte (tileRow * 32 +
tileCol))
                let tileIndex = ppu.vram.[int (tileMapAddr - 0x8000us)]

                let tileYOffset = scrolledY % 8 // Y offset within the tile
                let tileLineBytes = readTile ppu tileIndex tileYOffset
                let tileLinePixels = getTileLinePixels tileLineBytes
                let pixelColorIndex = tileLinePixels.[scrolledX % 8]

                pixels.[x] <- getPaletteColor bgPalette (int pixelColorIndex)
        else
            // If BG is disabled, fill with white (color 0)
            for x = 0 to 159 do
                pixels.[x] <- White

        // --- Sprite Rendering ---
        // Check OBJ Display Enable (Bit 1 of LCDC)
        let objDisplayEnabled = (ppu.lcdc &&& 0x02uy) = 0x02uy
        if objDisplayEnabled then
            let objSize = if (ppu.lcdc &&& 0x04uy) = 0x04uy then 16 else 8 //
LCDC bit 2: OBJ Size (0=8x8, 1=8x16)

            // Collect up to 10 sprites for the current scanline (Game Boy
hardware limit)
            let mutable visibleSprites = List.empty<Sprite>
            for i = 0 to 39 do // 40 possible sprites
                let oamOffset = i * 4
                let spriteY = int ppu.oam.[oamOffset] - 16 // Y position minus
16
                let spriteX = int ppu.oam.[oamOffset + 1] - 8 // X position
minus 8

                // Check if sprite is on current scanline
                if ly >= spriteY && ly < (spriteY + objSize) then
                    let attributes = ppu.oam.[oamOffset + 3]
                    let sprite =
                        { Y = ppu.oam.[oamOffset]
                          X = ppu.oam.[oamOffset + 1]
                          TileIndex = ppu.oam.[oamOffset + 2]
                          PaletteNum = (attributes >>> 4) &&& 0x01uy // DMG
```

```
only has OBP0/OBP1
                        XFlip = ((attributes >>> 5) &&& 0x01uy) = 0x01uy
                        YFlip = ((attributes >>> 6) &&& 0x01uy) = 0x01uy
                        BgPriority = ((attributes >>> 7) &&& 0x01uy) = 0x01uy
                        OamIndex = i
                    }
                visibleSprites <- sprite :: visibleSprites
                if List.length visibleSprites >= 10 then break // Hardware
limit of 10 sprites per line

            // Sort sprites by X-coordinate (lower X = higher priority), then
OAM index (lower index = higher priority)
            let sortedSprites =
                visibleSprites
                |> List.sortBy (fun s -> (s.X, s.OamIndex)) // F# sorts tuples
lexicographically

            // Render sprites onto the pixel buffer
            for sprite in sortedSprites do
                let spriteY = int sprite.Y - 16
                let spriteX = int sprite.X - 8
                let tileIndex = if objSize = 16 then sprite.TileIndex &&& 0xFEu
y else sprite.TileIndex // 8x16 sprites use two consecutive tiles, top tile
index is even

                let tileYInSprite = ly - spriteY
                let actualTileYOffset = if sprite.YFlip then objSize - 1 - tile
YInSprite else tileYInSprite

                let currentTileIndex =
                    if objSize = 16 then
                        if actualTileYOffset < 8 then tileIndex
                        else tileIndex + 1uy
                    else tileIndex

                let tileLineOffset = actualTileYOffset % 8

                let tileLineBytes = readTile ppu currentTileIndex tileLineOffse
t
                let tileLinePixels = getTileLinePixels tileLineBytes

                let objPalette = if sprite.PaletteNum = 0uy then ppu.obp0 else
ppu.obp1

                for px = 0 to 7 do
                    let screenX = spriteX + px
                    if screenX >= 0 && screenX < 160 then
                        let tilePixelX = if sprite.XFlip then 7 - px else px
                        let pixelColorIndex = tileLinePixels.[tilePixelX]

                        // Only draw if pixel is not transparent (color index
0)
                        if pixelColorIndex <> 0uy then
                            // Sprite priority check
                            let currentBgPixelColor = pixels.[screenX]
                            let bgPixelColorIndex =
                                match currentBgPixelColor with
                                | White -> 0uy
                                | LightGray -> 1uy
                                | DarkGray -> 2uy
                                | Black -> 3uy
                                | _ -> 0uy // Should not happen
```

```
                            let drawSprite =
                                if sprite.BgPriority then // Sprite is behind
BG colors 1-3
                                    bgPixelColorIndex = 0uy
                                else // Sprite is always in front of BG
                                    true

                            if drawSprite then
                                pixels.[screenX] <- getPaletteColor objPalette
(int pixelColorIndex)

        // Update the LCD buffer with the rendered scanline
        { ppu with lcdBuffer = ppu.lcdBuffer |> Array.mapi (fun i v -> if i / 1
60 = ly then pixels.[i % 160] else v) }
```

Explanation:

- **`readTile` refinement:** The `readTile` function now correctly uses `LCDC` bit 4 to determine if tiles are fetched from `0x8000-0x8FFF` (unsigned indices) or `0x8800-0x97FF` (signed indices relative to `0x9000`). This is a crucial detail for games using both tile data regions.
- **LCD Enable:** The first check ensures no rendering occurs if the LCD is disabled, preventing unnecessary computation and accurately reflecting hardware behavior.
- **Background Display Enable:** `LCDC` bit 0 controls whether the background is displayed. If disabled, the screen is filled with `White`.
- **Scrolling:**
  - `scrolledY = (ly + int ppu.scy) &&& 0xFF`: The current screen `ly` is offset by `SCY` and wrapped around 256. This gives the effective Y-coordinate in the 256x256 background map.
  - `scrolledX = (x + int ppu.scx) &&& 0xFF`: Similarly, `SCX` offsets the X-coordinate.
  - These `scrolledY` and `scrolledX` values are used to calculate the correct `tileRow`, `tileCol`, and `tileYOffset` within the background map.
- **Sprite Collection:**
  - We iterate through all 40 potential sprites in OAM.
  - For each, we check if its Y-position (`spriteY`) falls within the current `ly` scanline, considering the `objSize` (8x8 or 8x16, from `LCDC` bit 2).
  - Up to 10 sprites are collected for the current line, as per Game Boy hardware limits.

- **Sprite Sorting:**
  - Visible sprites are sorted. The Game Boy has a fixed priority: sprites with smaller X-coordinates are drawn on top. If X-coordinates are equal, sprites with lower OAM indices (0-39) have higher priority. This is handled by `List.sortBy (fun s -> (s.X, s.OamIndex))`.
- **Sprite Rendering Loop:**
  - For each sorted sprite, we calculate its pixel data.
  - `tileIndex`: For 8x16 sprites, the `TileIndex` in OAM refers to the top tile (must be an even number). We adjust `tileIndex` based on `tileYInSprite`.
  - `XFlip`/`YFlip`: These attributes reverse the tile's pixels horizontally or vertically.
  - `BgPriority`: `LCDC` bit 7 in the sprite attributes (`BgPriority`) determines if the sprite is drawn behind background colors 1-3 (non-transparent) or always on top. If `BgPriority` is set and the background pixel is not transparent (color index 0), the sprite pixel (if not transparent itself) will not be drawn.
  - Palette Selection: `OBP0` or `OBP1` is chosen based on the sprite's `PaletteNum` attribute.
  - Transparent Pixels: Sprite pixel color index `0` is always transparent and doesn't overwrite existing background pixels.
- **Final Pixel Combination:** The background pixel is determined first, then sprites are drawn on top, respecting their priority rules.

6. Update PPU Initialization and `MMU` References

Ensure your `MMU` construction in `main` or `Emulator` module passes the `ppu` state to the `MMU`. This establishes the `MMU` as the central point for all memory and I/O access.

```
// src/GameBoyEmulator/Emulator.fs (or wherever you initialize MMU and PPU)

module GameBoyEmulator.Emulator

open System
open GameBoyEmulator.CPU
open GameBoyEmulator.MMU
open GameBoyEmulator.PPU
open GameBoyEmulator.Cartridge
open GameBoyEmulator.Types
open Silk.NET.Windowing // Using Silk.NET.Windowing as a modern SDL.NET
alternative

// ... (existing code)

let createEmulator (romPath: string) : Emulator =
    let cartridge = Cartridge.loadCartridge romPath

    let ppu = PPU.initPPU cartridge
    let mmu = MMU.initMMU cartridge ppu // Pass initial PPU state to MMU
    let cpu = CPU.initCPU()

    { cpu = cpu; mmu = mmu; cartridge = cartridge; cycles = 0 }
```

And ensure the `MMU` type holds the `PPU` record:

```
// src/GameBoyEmulator/MMU.fs

module GameBoyEmulator.MMU

open GameBoyEmulator.PPU
open GameBoyEmulator.Cartridge
open GameBoyEmulator.Types

type MMU =
    { rom      : byte array
      ram      : byte array
      ppu      : PPU.PPU // MMU directly holds the PPU state
      cartridge: Cartridge.Cartridge
      // ... other fields
    }

let initMMU (cartridge: Cartridge.Cartridge) (initialPPU: PPU.PPU) : MMU =
    { rom = cartridge.rom
      ram = Array.zeroCreate 0x10000 // 64KB RAM, will be managed by MMU
      ppu = initialPPU // Initialize with the PPU state
      cartridge = cartridge
      // ... other fields
    }
```

This setup ensures that `MMU` is the central point of truth for the `PPU`'s state, allowing the CPU to read/write PPU registers via the `MMU`'s `readByte`/`writeByte` functions, and `PPU.stepPPU` to modify its own internal state within the `MMU`.

Testing & Verification

With sprites and scrolling implemented, you'll want to verify with specific test ROMs. This is an iterative process of running, observing, and debugging.

1. **Blargg's Test ROMs:** These are industry-standard for Game Boy emulator development.
    - `cpu_instrs/individual/09-op r,r.gb`: This ROM often displays a simple 'OK' message and scrolls. Verify the scrolling behavior.
    - `sprite_priority.gb`: This ROM specifically tests sprite rendering, including priority rules. It shows various patterns where sprites should overlap or be hidden by the background.
    - `oam_bug.gb`: While we haven't implemented OAM DMA timing accurately, this ROM can reveal issues with OAM access or sprite sorting.
    - `lcd_align.gb`: Useful for checking basic LCD timing and alignment.
2. **Visual Inspection:**
    - Load simple games (e.g., Tetris, Dr. Mario).
    - Observe how sprites (the falling blocks, score digits) are rendered.
    - Check if background elements scroll smoothly when the game moves.
    - Look for visual artifacts like flickering sprites, incorrect layering, or misaligned scrolling.
3. **Debugging Checks:**

- **Log Register Values:** Print `ppu.scx`, `ppu.scy`, `ppu.lcdc`, `ppu.ly` at the start of each scanline. This helps track if the CPU is writing the expected values.
- **Memory Dump OAM:** At key points, dump the `ppu.oam` array to see if sprite attributes are correctly stored.
- **Conditional Breakpoints:** Set breakpoints in your `drawScanline` function, especially within the sprite rendering loop, to inspect individual sprite data and pixel blending logic.

Production Considerations

🔥 `Optimization / Pro tip:` The `drawScanline` function is performance-critical, executing 154 times per frame (144 visible lines + 10 VBlank lines for `LY` updates).

- **Sprite Caching:** Instead of re-parsing OAM for every pixel, collect and sort visible sprites once per scanline. We already do this by collecting `visibleSprites`.
- **Early Exit:** If `OBJ Display Enable` (LCDC bit 1) is off, skip the entire sprite rendering loop.
- **Palette Lookups:** Pre-calculate the full palette colors if performance becomes an issue, rather than doing bit shifts and `match` expressions for every pixel.
- **Immutability vs. Performance:** While F# favors immutability, the pixel buffer (`pixels`) is often mutated for performance within a single function scope. This is a pragmatic tradeoff in performance-critical inner loops. The `ppu.lcdBuffer` is then updated immutably at the end of the scanline, preserving the overall functional structure.

⚠️ `What can go wrong:`

- **Off-by-one errors:** Sprite Y/X positions are offset by -16 and -8 respectively from their OAM values. Incorrectly applying these offsets will cause sprites to appear shifted.
- **Incorrect Priority:** The Game Boy's sprite priority rules are specific (X-coordinate, then OAM index, then `BgPriority` bit). Any deviation will cause sprites to be drawn in the wrong order.
- **DMA Timing:** While we simplified DMA to an instant copy, in a highly accurate emulator, DMA takes 160 machine cycles and halts the CPU. Incorrect DMA timing can lead to corrupted OAM or CPU desynchronization in some games.
- **LCDC Bit Misinterpretation:** Each bit in `LCDC` has a specific, critical function. Misinterpreting any bit (e.g., sprite size, background enable) will lead to significant visual bugs.

Common Issues & Solutions

1. **Sprites are not visible or are flickering:**

- **Issue:** `LCDC` bit 1 (OBJ Display Enable) might be off, or `objSize` (LCDC bit 2) is wrong. Sprite X/Y positions are often off-screen (-16 for Y, -8 for X) or outside the current scanline's range.
- **Solution:** Verify `LCDC` value in your debugger. Ensure your sprite Y/X calculations correctly account for the -16/-8 offsets. Check the `ly` range for sprite visibility. 2. **Sprites are drawn in the wrong order:**
- **Issue:** Incorrect sprite sorting logic. The Game Boy has a specific hardware priority.
- **Solution:** Ensure sprites are first sorted by `sprite.X` (ascending), then by `sprite.OamIndex` (ascending) for tie-breaking, before rendering. 3. **Background scrolling is jerky or incorrect:**
- **Issue:** `SCX`/`SCY` wrapping or offset calculations are wrong. The background map is 256x256 pixels, and the viewport is 160x144.
- **Solution:** Remember that `SCX` and `SCY` are 8-bit values and wrap around 256. The `(&&& 0xFF)` operation is crucial for this. Double-check `tileRow` and `tileCol` calculations. 4. **VRAM/OAM access restrictions causing glitches:**
- **Issue:** CPU writes to VRAM/OAM during PPU `Drawing` or `OamScan` modes are not being ignored or returning `0xFF` for reads.
- **Solution:** Re-check the `MMU.writeByte` and `MMU.readByte` functions for `0x8000-0x9FFF` (VRAM) and `0xFE00-0xFE9F` (OAM) to ensure they correctly check `mmu.ppu.mode` and enforce restrictions.

Summary & Next Step

In this chapter, you've significantly advanced your Game Boy emulator's visual capabilities. You've implemented:

- **Sprite Rendering:** Parsing OAM data, handling sprite attributes (position, tile index, palettes, flips, priority), and drawing up to 10 sprites per scanline.
- **Background Scrolling:** Using `SCX` and `SCY` registers to dynamically offset the background display.

- **LCD Control Register (`LCDC`):** Integrated its various bits to enable/disable display components (BG, sprites), set sprite size, and control tile data/map selection.
- **PPU Timing Refinements:** Enhanced the `stepPPU` function to accurately handle `LYC=LY` coincidence and `STAT` interrupts for different PPU modes, including the correct `STAT` register read behavior.

Your emulator can now render much more complex game scenes, with moving characters and scrolling backgrounds. The visual foundation is becoming robust.

Next, we'll shift our focus to interaction, implementing input handling to allow the user to control games.

🧠 Check Your Understanding

- What is the purpose of the `BgPriority` bit in a sprite's attributes, and how does it affect rendering?
- Why is it important to sort sprites before drawing them, and what are the Game Boy's rules for sprite priority?
- What happens if the LCD Control Register (LCDC) bit 7 (LCD Enable) is set to 0, and how should your PPU handle this?

⚡ Mini Task

- Add a debug function to your `PPU` module that, when called, prints the current values of `LCDC`, `SCX`, `SCY`, `LY`, `OBP0`, and `OBP1` to the console. Integrate this into your main loop for easy inspection.

🚀 Scenario

A new Game Boy ROM you're testing displays its main character correctly, but all background elements are missing. You've checked that the background tiles are loaded into VRAM. What are the most likely PPU register settings or code issues that could cause this, and how would you debug it?

📌 TL;DR

- Sprites add dynamic objects, managed by OAM and `LCDC` attributes.

- `SCX` and `SCY` registers enable background scrolling by offsetting tile map lookups.
- `LCDC` is the master control, enabling/disabling display features and defining sprite size and tile data sources.

🧠 Core Flow

1. PPU checks `LCDC` enable bit; if off, PPU halts.
2. PPU updates `LYC=LY` coincidence flag in `STAT` and triggers `LcdStat` interrupt if enabled.
3. For each scanline, PPU renders background pixels, applying `SCX`/`SCY` offsets based on `LCDC` settings.
4. PPU scans OAM, filters up to 10 visible sprites, sorts them by X then OAM index.
5. PPU renders sorted sprites over background, respecting `BgPriority` and transparency.

🚀 Key Takeaway

Accurate emulation of the Game Boy's PPU requires meticulous attention to hardware-specific details like register bit flags, memory access restrictions, and strict sprite priority rules, which collectively dictate the final rendered frame.

References

- Pan Docs - LCD
- Pan Docs - OAM
- F# Language Reference
- Silk.NET.Windowing Documentation
- Game Boy CPU Manual (SM83)

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 09

Input Handling: Connecting Keyboard to Game Boy Buttons

Handling user input is crucial for any interactive application, especially an emulator. In this chapter, we'll bridge the gap between your physical keyboard and the Game Boy's virtual buttons. You'll learn how to capture keyboard events, map them to the Game Boy's specific input registers, and integrate this into your emulator's main loop. By the end, your emulator will respond to your key presses, allowing you to control Game Boy games.

Introduction

So far, our Game Boy emulator can execute CPU instructions, manage memory, and even render basic graphics. But a game isn't much fun without interaction! This chapter focuses on implementing input handling, specifically translating keyboard presses into the Game Boy's button states.

We'll leverage the SDL.NET library, which we've already used for graphics, to capture keyboard events. The core challenge is to correctly model the Game Boy's unique input mechanism and integrate our key mapping logic seamlessly into the existing emulator architecture, ensuring that the Game Boy's CPU can "read" our button presses.

By the end of this chapter, you'll be able to press keys on your keyboard and see the Game Boy emulator react, bringing your games to life. This is a significant milestone, as it completes the fundamental interactive loop of the emulator.

Planning & Design

The Game Boy's input system is relatively simple but requires careful modeling. It uses a single I/O register, `0xFF00` (P1), to detect button presses. This register is somewhat unique because it's both read and written by the CPU. The CPU writes to `P1` to select whether it wants to read the "direction" buttons (Right, Left, Up, Down) or the "action" buttons (A, B, Select, Start). When the CPU reads `P1`, the bits corresponding to the selected buttons will be `0` if pressed and `1` if not.

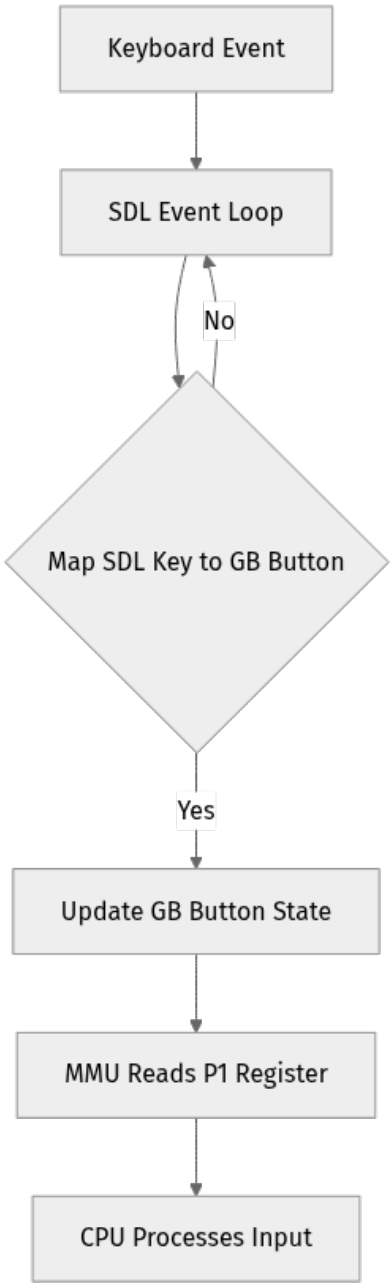
Our design needs to address: 1. **Game Boy Button Representation:** How do we model the eight Game Boy buttons in F#? 2. **Keyboard Mapping:** Which physical keyboard keys correspond to which Game Boy buttons? 3. **Input State**

Management: How do we track the current pressed/released state of all Game Boy buttons? 4. **SDL Event Integration:** How do we capture keyboard events from SDL and update our internal Game Boy button state? 5. **MMU Interaction:** How do we expose this button state to the MMU so the CPU can read the `P1` register correctly?

Architecture Overview

We'll introduce a new `Input` module responsible for managing the Game Boy's button state and handling SDL keyboard events. This module will maintain the current state of all buttons. The `MMU` will then query this `Input` module when the CPU attempts to read the `P1` register.

Here's a conceptual flow:



This ensures a clean separation of concerns: the `Input` module knows about SDL and keyboard keys, while the `MMU` only knows about Game Boy button states, abstracting away the underlying physical input.

Step-by-Step Implementation

We'll modify our `Emulator.fs`, `MMU.fs`, and potentially create a new `Input.fs` module.

1. Define Game Boy Buttons and Input State

First, let's define the Game Boy buttons and a way to store their current state. We'll add this to a new file, `Input.fs`.

File: `Input.fs`

```
module GbEmulator.Input

open System.Collections.Generic

/// Represents the 8 Game Boy buttons.
type GbButton =
    | Right
    | Left
    | Up
    | Down
    | A
    | B
    | Select
    | Start

/// Represents the current state of all Game Boy buttons.
/// A Dictionary is used for efficient lookup by button type.
type GbInputState =
    {
        ButtonStates: Dictionary<GbButton, bool> // true if pressed, false if
released
    }

/// Initializes a new Game Boy input state with all buttons released.
let createInputState () =
    let buttonStates = Dictionary<GbButton, bool>()
    [GbButton.Right; GbButton.Left; GbButton.Up; GbButton.Down;
     GbButton.A; GbButton.B; GbButton.Select; GbButton.Start]
    |> List.iter (fun btn -> buttonStates.Add(btn, false))
    { ButtonStates = buttonStates }

/// Updates the state of a specific Game Boy button.
let updateButtonState (state: GbInputState) (button: GbButton) (isPressed:
bool) =
    state.ButtonStates.[button] <- isPressed
    state // Return the modified state (Dictionary is mutable, but we return
the record for consistency)

/// Checks if a specific Game Boy button is currently pressed.
let isButtonPressed (state: GbInputState) (button: GbButton) =
    state.ButtonStates.[button]
```

Explanation: - `GbButton` is a discriminated union representing the 8 distinct Game Boy buttons. - `GbInputState` uses a `Dictionary<GbButton, bool>` to store whether each button is currently pressed (`true`) or released (`false`). Using a `Dictionary` provides quick access to button states. - `createInputState` initializes this dictionary with all buttons set to `false` (released). - `updateButtonState` modifies the state of a given button. Note that `Dictionary`

is a mutable type, so we're performing an in-place update. In F#, it's common to encapsulate such mutable state within a module or class to maintain functional purity at a higher level. - `isButtonPressed` provides a simple lookup for a button's state.

2. Define Keyboard Key Mappings

Now, let's define which physical keyboard keys map to which `GbButton`. We'll keep this in `Input.fs` as well.

File: `Input.fs` (Append to existing content)

```
open SDL2

/// Maps SDL_Keycode values to GbButton values.
let private keyMappings =
    Map.ofList [
        (SDL.SDL_Keycode.SDLK_RIGHT, GbButton.Right)
        (SDL.SDL_Keycode.SDLK_LEFT, GbButton.Left)
        (SDL.SDL_Keycode.SDLK_UP, GbButton.Up)
        (SDL.SDL_Keycode.SDLK_DOWN, GbButton.Down)
        (SDL.SDL_Keycode.SDLK_z, GbButton.A) // A button
        (SDL.SDL_Keycode.SDLK_x, GbButton.B) // B button
        (SDL.SDL_Keycode.SDLK_a, GbButton.Select) // Select button
        (SDL.SDL_Keycode.SDLK_s, GbButton.Start) // Start button
    ]

/// Attempts to get the GbButton corresponding to an SDL_Keycode.
let tryGetGbButton (keycode: SDL.SDL_Keycode) : GbButton option =
    Map.tryFind keycode keyMappings
```

Explanation: - `keyMappings` is an F# `Map` that stores the association between `SDL_Keycode` (from `SDL2-CS`) and our `GbButton` type. - `tryGetGbButton` is a helper function that safely looks up a `GbButton` for a given `SDL_Keycode`, returning an `option` type (`Some GbButton` if found, `None` otherwise). This prevents crashes if an unmapped key is pressed.

3. Integrate Input State into MMU

The `MMU` (Memory Management Unit) is responsible for handling reads and writes to `0xFF00`. We need to modify `MMU.fs` to allow the `MMU` to query the current `GbInputState`.

File: `MMU.fs`

```
module GbEmulator.MMU

open GbEmulator.Cpu
open GbEmulator.Input // Add this line to import our Input module

/// Represents the Memory Management Unit's state.
type MmuState =
    {
        Rom: byte array
        Ram: byte array
        Vram: byte array
        Wram: byte array
        Oam: byte array
        IoRegisters: byte array // General purpose I/O registers
        IeRegister: byte // Interrupt Enable register (0xFFFF)
        // Add a field to hold the input state
        InputState: GbInputState
    }

/// Creates a new MMU state.
let createMmu (rom: byte array) (inputState: GbInputState) = // Modify
signature
    {
        Rom = rom
        Ram = Array.zeroCreate 0x2000 // 8KB WRAM (0xC000-0xDFFF)
        Vram = Array.zeroCreate 0x2000 // 8KB VRAM (0x8000-0x9FFF)
        Wram = Array.zeroCreate 0x2000 // Work RAM
        Oam = Array.zeroCreate 0xA0 // Object Attribute Memory (OAM)
        IoRegisters = Array.zeroCreate 0x80 // I/O Registers (0xFF00-0xFF7F)
        IeRegister = 0x00uy // Interrupt Enable
        InputState = inputState // Initialize with the passed-in input state
    }

/// Reads a byte from memory at the given address.
let readByte (mmu: MmuState) (addr: uint16) : byte =
    match addr with
    | _ when addr >= 0x0000us && addr <= 0x7FFFus -> mmu.Rom.[int addr] //
Cartridge ROM
    | _ when addr >= 0x8000us && addr <= 0x9FFFus -> mmu.Vram.[int (addr - 0x80
00us)] // VRAM
    | _ when addr >= 0xC000us && addr <= 0xDFFFus -> mmu.Wram.[int (addr - 0xC0
00us)] // WRAM
    | _ when addr >= 0xE000us && addr <= 0xFDFFus -> mmu.Wram.[int (addr - 0xE0
00us)] // Echo RAM (mirror of WRAM)
    | _ when addr >= 0xFE00us && addr <= 0xFE9Fus -> mmu.Oam.[int (addr - 0xFE0
0us)] // OAM
    | _ when addr >= 0xFF00us && addr <= 0xFF7Fus -> // I/O Registers
        match addr with
        | 0xFF00us -> // P1/JOYP register
            // The Game Boy CPU writes to P1 to select which buttons to read.
            // Bit 5: Select Direction Buttons (P15)
            // Bit 4: Select Action Buttons (P14)
            let p15 = (mmu.IoRegisters.[int (0xFF00us - 0xFF00us)] &&& 0x20uy)
= 0x00uy // Is P15 low?
            let p14 = (mmu.IoRegisters.[int (0xFF00us - 0xFF00us)] &&& 0x10uy)
= 0x00uy // Is P14 low?

            let mutable result = 0xFFuy // All bits high (buttons not pressed)

            // If P15 is low, read direction buttons
            if p15 then
```

```
                if Input.isButtonPressed mmu.InputState Input.GbButton.Right th
en result <- result &&& (~~~ 0x01uy)
                if Input.isButtonPressed mmu.InputState Input.GbButton.Left  th
en result <- result &&& (~~~ 0x02uy)
                if Input.isButtonPressed mmu.InputState Input.GbButton.Up    th
en result <- result &&& (~~~ 0x04uy)
                if Input.isButtonPressed mmu.InputState Input.GbButton.Down  th
en result <- result &&& (~~~ 0x08uy)

            // If P14 is low, read action buttons
            if p14 then
                if Input.isButtonPressed mmu.InputState Input.GbButton.A      t
hen result <- result &&& (~~~ 0x01uy)
                if Input.isButtonPressed mmu.InputState Input.GbButton.B      t
hen result <- result &&& (~~~ 0x02uy)
                if Input.isButtonPressed mmu.InputState Input.GbButton.Select t
hen result <- result &&& (~~~ 0x04uy)
                if Input.isButtonPressed mmu.InputState Input.GbButton.Start  t
hen result <- result &&& (~~~ 0x08uy)

            // Combine with the control bits (P15, P14) from the CPU's last
write
            result <- result ||| (mmu.IoRegisters.[int (0xFF00us - 0xFF00us)] &
&& 0xF0uy)
            result

        | _ -> mmu.IoRegisters.[int (addr - 0xFF00us)] // Other I/O registers
    | 0xFFFFus -> mmu.IeRegister // Interrupt Enable register
    | _ -> 0xFFuy // Unmapped memory returns 0xFF

/// Writes a byte to memory at the given address.
let writeByte (mmu: MmuState) (addr: uint16) (value: byte) : MmuState =
    match addr with
    | _ when addr >= 0x0000us && addr <= 0x7FFFus -> { mmu with Rom =
mmu.Rom } // ROM is read-only
    | _ when addr >= 0x8000us && addr <= 0x9FFFus ->
        mmu.Vram.[int (addr - 0x8000us)] <- value; mmu
    | _ when addr >= 0xC000us && addr <= 0xDFFFus ->
        mmu.Wram.[int (addr - 0xC000us)] <- value; mmu
    | _ when addr >= 0xE000us && addr <= 0xFDFFus ->
        mmu.Wram.[int (addr - 0xE000us)] <- value; mmu // Write to WRAM echo
    | _ when addr >= 0xFE00us && addr <= 0xFE9Fus ->
        mmu.Oam.[int (addr - 0xFE00us)] <- value; mmu
    | _ when addr >= 0xFF00us && addr <= 0xFF7Fus ->
        // P1/JOYP register (0xFF00) write: CPU selects which buttons to read
        // Only bits 4 and 5 are writable (P14, P15)
        if addr = 0xFF00us then
            mmu.IoRegisters.[int (addr - 0xFF00us)] <- value &&&
0x30uy // Keep only bits 4 and 5
        else
            mmu.IoRegisters.[int (addr - 0xFF00us)] <- value
        mmu
    | 0xFFFFus -> { mmu with IeRegister = value }
    | _ -> mmu // Do nothing for unmapped writes
```

Explanation of `MMU.fs` changes: - We added `open GbEmulator.Input` to use our new input module. - `MmuState` now includes an `InputState: GbInputState` field. - `createMmu` now takes a `GbInputState` as an argument, which it stores. -

Crucially, the `readByte` function's `0xFF00us` case is updated: - It first reads the current value in `IoRegisters.[0]` (which corresponds to `0xFF00`) to determine if the CPU wants to read direction buttons (Bit 5 low) or action buttons (Bit 4 low). - `p15` and `p14` flags check if the respective bits are low (0). - `result` is initialized to `0xFFuy` (all bits high, indicating no buttons pressed). - Based on `p15` and `p14`, it queries `mmu.InputState` using `Input.isButtonPressed`. If a button is pressed, the corresponding bit in `result` is set to `0`. (e.g., `~~~ 0x01uy` is `0xFEuy`, which effectively clears bit 0). - Finally, the previously written P14 and P15 bits from the `IoRegisters` are re-applied to the result, as these are control bits, not button states. - The `writeByte` for `0xFF00us` is also updated to only allow writing to bits 4 and 5, as the lower bits are read-only (reflecting button state).

4. Integrate SDL Event Handling into the Main Loop

Now, we need to modify our main emulator loop (likely in `Emulator.fs` or `Program.fs`) to process SDL keyboard events and update the `GbInputState`.

File: `Emulator.fs` (or `Program.fs` if your main loop is there)

First, ensure `SDL2-CS` is installed.

```
dotnet add package SDL2-CS --version 2.28.0 # Or latest stable
```

(Using `2.28.0` as a placeholder for a recent stable version as of 2026-05-05. Verify the latest on NuGet.)

```
module GbEmulator.Emulator

open SDL2
open System
open System.Diagnostics
open GbEmulator.Cpu
open GbEmulator.MMU
open GbEmulator.Ppu
open GbEmulator.Input // Add this line

/// Represents the overall state of the Game Boy emulator.
type EmulatorState =
    {
        Cpu: CpuState
        Mmu: MmuState
        Ppu: PpuState
        Cycles: int64 // Total CPU cycles elapsed
        Input: GbInputState // Keep a reference to the mutable input state
    }

/// Initializes a new emulator state.
let createEmulator (romData: byte array) =
    let inputState = Input.createInputState() // Create input state
    let mmu = MMU.createMmu romData inputState // Pass input state to MMU
    let cpu = Cpu.createCpu()
    let ppu = Ppu.createPpu()
    { Cpu = cpu; Mmu = mmu; Ppu = ppu; Cycles = 0L; Input = inputState } //
Store input state
```

Now, modify the main loop. This example assumes your main loop is in `Emulator.fs` and interacts with SDL.

File: `Emulator.fs` (Modify the main loop function)

```
// ... (previous code) ...

/// Processes SDL events, updating the emulator's input state.
let handleSdlEvents (emulator: EmulatorState) : EmulatorState =
    let mutable event = SDL.SDL_Event()
    let mutable currentEmulator = emulator // Use mutable local variable for
updates

    while SDL.SDL_PollEvent(&event) = 1 do
        match event.type with
        | SDL.SDL_EventType.SDL_QUIT ->
            // Signal to quit the emulator
            currentEmulator <- { currentEmulator with Cpu = { currentEmulator.C
pu with Running = false } }
        | SDL.SDL_EventType.SDL_KEYDOWN ->
            let keycode = event.key.keysym.sym
            match Input.tryGetGbButton keycode with
            | Some gbButton ->
                Input.updateButtonState currentEmulator.Input gbButton true //
Update button state
            | None -> () // Ignore unmapped keys
        | SDL.SDL_EventType.SDL_KEYUP ->
            let keycode = event.key.keysym.sym
            match Input.tryGetGbButton keycode with
            | Some gbButton ->
                Input.updateButtonState currentEmulator.Input gbButton
false // Update button state
            | None -> () // Ignore unmapped keys
        | _ -> () // Ignore other event types
    currentEmulator

/// The main emulator loop.
let runEmulator (emulator: EmulatorState) : unit =
    let mutable currentEmulator = emulator
    let targetFps = 60.0
    let targetFrameTimeMs = 1000.0 / targetFps
    let mutable lastFrameTime = Stopwatch.GetTimestamp()

    while currentEmulator.Cpu.Running do
        let frameStartTime = Stopwatch.GetTimestamp()

        // 1. Handle SDL events (input, quit)
        currentEmulator <- handleSdlEvents currentEmulator // Call our new
event handler

        // 2. Emulate CPU cycles for one frame (roughly 70224 cycles for 60Hz)
        let cyclesThisFrame = 0 // Reset for calculation below
        let maxCycles = 70224 // Game Boy CPU cycles per frame at 60Hz
        let mutable currentCycles = 0

        while currentCycles < maxCycles && currentEmulator.Cpu.Running do
            let (newCpu, newMmu, cyclesExecuted) = Cpu.executeNextInstruction c
urrentEmulator.Cpu currentEmulator.Mmu
            currentEmulator <- { currentEmulator with Cpu = newCpu; Mmu = newMm
u; Cycles = currentEmulator.Cycles + int64 cyclesExecuted }
            currentCycles <- currentCycles + cyclesExecuted

            // PPU step (update PPU for the cycles executed)
            let (newPpu, newMmuPpu) = Ppu.step currentEmulator.Ppu newMmu cycle
sExecuted
            currentEmulator <- { currentEmulator with Ppu = newPpu; Mmu = newMm
```

```
uPpu }

        // 3. Render frame if PPU signals (handled by PPU.step)
        // The PPU module will handle rendering to the SDL window itself
        // if a full frame is ready.

        // 4. Frame rate synchronization
        let frameEndTime = Stopwatch.GetTimestamp()
        let elapsedTicks = frameEndTime - frameStartTime
        let elapsedMs = float elapsedTicks / float Stopwatch.Frequency * 1000.0

        let sleepTime = targetFrameTimeMs - elapsedMs
        if sleepTime > 0.0 then
            System.Threading.Thread.Sleep(int sleepTime)

    printfn "Emulator stopped."
    SDL.SDL_Quit()
```

Explanation of `Emulator.fs` changes: - We added `open GbEmulator.Input`. - `EmulatorState` now includes `Input: GbInputState` to hold the reference to our mutable input state. - `createEmulator` now initializes `Input.createInputState()` and passes it to `MMU.createMmu`. - A new function `handleSdlEvents` is introduced. This function: - Polls for SDL events using `SDL.SDL_PollEvent`. - Handles `SDL_QUIT` to stop the emulator. - For `SDL_KEYDOWN` and `SDL_KEYUP` events, it uses `Input.tryGetGbButton` to find the corresponding `GbButton`. - If a mapped button is found, it calls `Input.updateButtonState` to set the button's state (pressed or released). - The `runEmulator` main loop now calls `handleSdlEvents` at the beginning of each frame.

⚡ **Quick Note:** While `Dictionary` is mutable, we're passing it around within a record. In F#, it's a common pragmatic choice for performance-critical areas like input state, especially when it's tightly managed within a single logical component (the `Input` module). The `EmulatorState` holds a reference to this mutable `GbInputState`, and functions like `handleSdlEvents` modify it directly.

Testing & Verification

To verify our input handling, we need a Game Boy ROM that visually indicates button presses.

1. **Build and Run:** Compile your project: `bash dotnet build dotnet run -- path/to/your/rom.gb` Replace `path/to/your/rom.gb` with a Game Boy ROM.
2. **Test with Blargg's Test ROMs:** Blargg's CPU instruction test ROMs, specifically `cpu_instrs/individual/01-special.gb`, often include a

screen that displays the state of the P1 register, which is perfect for verifying input.

- Download Blargg's CPU instruction test ROMs.
- Run `01-special.gb`. It usually shows a sequence of tests. Look for a screen that changes when you press keys.

3. **Manual Playtesting:** Load a simple Game Boy game (e.g., Tetris, Dr. Mario).
   - Press the configured keys (Z for A, X for B, etc.).
   - Observe if the game responds correctly to your inputs. Can you move the cursor, make selections, or play the game as expected?
4. **Debugging `P1` Register:** If things aren't working, consider adding a temporary logging statement in your `MMU.fs` within the `readByte` function for address `0xFF00us`. This will let you see the exact value the CPU is reading from the input register.

   `fsharp // Inside MMU.fs, in the readByte function for 0xFF00us: // ... result <- result ||| (mmu.IoRegisters.[int (0xFF00us - 0xFF00us)] &&& 0xF0uy) // Debugging line: // printfn "P1 Read: 0x%02x, P15: %b, P14: %b" result p15 p14 result` This helps confirm if the `result` byte reflects your button presses correctly. Remember that `0` means pressed, `1` means released for the button bits.

Production Considerations

Performance

Input handling is generally not a major performance bottleneck for emulators. SDL's event polling is efficient. The key is to avoid complex logic within the event loop itself. Our current approach of simple dictionary lookups and bitwise operations is very fast.

Maintainability

- **Clear Key Mapping:** The `keyMappings` `Map` is easy to understand and modify. For a real production emulator, you'd likely want to externalize this configuration (e.g., to a JSON file) to allow user customization.
- **Modular Design:** Separating input logic into `Input.fs` keeps the `MMU` and `Emulator` modules cleaner and focused on their core responsibilities.

Customization

A common feature in emulators is configurable key bindings. To achieve this, you would: 1. Load key mappings from a configuration file (e.g., `appsettings.json`, or a custom `.ini` file). 2. Provide a UI for users to change these mappings and save them. Our current hardcoded `keyMappings` provides the foundation for this.

Common Issues & Solutions

1. Keys "Sticking" or Not Registering Releases

Issue: Buttons remain pressed even after releasing the key, or key presses are missed. **Cause:** - `SDL_KEYUP` events are not being processed correctly. - The `handleSdlEvents` loop might not be called frequently enough, or the event queue might be overflowing (unlikely for a simple emulator). **Solution:** - Ensure `handleSdlEvents` is called every frame in your main loop. - Double-check the `SDL_KEYUP` event handling logic in `handleSdlEvents` to confirm it correctly calls `Input.updateButtonState` with `false`. - Verify your `keyMappings` are consistent for both `KEYDOWN` and `KEYUP` events.

2. Incorrect Button Behavior (e.g., A button acts like Right)

Issue: Pressing a key triggers the wrong Game Boy button. **Cause:** - Incorrect `keyMappings` in `Input.fs`. - Mistakes in the bitwise logic within `MMU.readByte` for `0xFF00us`, where the button states are translated into the P1 register value. **Solution:** - Carefully review `Input.keyMappings` against your desired keyboard layout. - Re-examine the `MMU.readByte` logic for `0xFF00us`. Remember: - Bit 0 (0x01) is Right/A - Bit 1 (0x02) is Left/B - Bit 2 (0x04) is Up/Select - Bit 3 (0x08) is Down/Start - `0` means pressed, `1` means released. So, `result <- result &&& (~~~ 0x01uy)` is the correct way to set a bit to `0` if pressed.

3. Emulator Not Responding to Any Keys

Issue: No input is registered at all. **Cause:** - SDL event polling is not happening. - The `handleSdlEvents` function is not being called in the main loop. - The `MMU` isn't correctly querying the `InputState` or the `InputState` isn't being passed to the `MMU`. **Solution:** - Verify `currentEmulator <- handleSdlEvents currentEmulator` is present and correctly placed in `runEmulator`. - Ensure `GbEmulator.Input` is `open`ed in `MMU.fs` and `Emulator.fs`. - Confirm that `MMU.createMmu` receives and stores the `GbInputState` correctly. - Check that the `0xFF00us` case in `MMU.readByte` actually calls `Input.isButtonPressed`.

Summary & Next Step

In this chapter, you've successfully implemented keyboard input for your Game Boy emulator. We defined the Game Boy's buttons, created a mapping from SDL keyboard events to these buttons, managed the button press state, and integrated this state into the `MMU` so the CPU can read it. You can now interact with Game Boy ROMs using your keyboard!

Your emulator now has the core components for interactivity: CPU, MMU, PPU, and Input. The next logical step is to add sound, which will bring another dimension of realism to your emulator.

🧠 Check Your Understanding

- Why does the Game Boy's P1 register require both reading and writing by the CPU for input handling?
- What are the advantages of using a `Map` for key mappings over a series of `if/else` statements or a `match` expression?
- Explain the purpose of `result <- result &&& (~~~ 0x01uy)` in the MMU's `readByte` function for the P1 register.

⚡ Mini Task

- Modify the `Input.fs` module to include an alternative set of key mappings (e.g., using WASD for directions and JKL; for action buttons) and a way to switch between them.

🚀 Scenario

You're running a Game Boy test ROM that displays the raw value of the P1 register (0xFF00). When you press the 'A' button (mapped to 'Z'), the P1 register value changes from `0xFF` to `0xEF`. When you press 'Right' (mapped to 'Right Arrow'), it changes from `0xFF` to `0xFE`. However, when you press both 'A' and 'Right' simultaneously, the register value becomes `0xEE`. Explain why this happens based on the Game Boy's P1 register behavior and bitwise operations.

📌 TL;DR

- Game Boy input uses I/O register `0xFF00` (P1), which the CPU writes to select button groups (directions or actions) and reads to get button states.
- We modeled Game Boy buttons with an F# Discriminated Union and managed their state (pressed/released) using a mutable `Dictionary` within an `Input` module.
- SDL keyboard events (`KEYDOWN`, `KEYUP`) are captured and mapped to `GbButton`s to update the `InputState`.
- The `MMU`'s `readByte` function for `0xFF00` now queries the `InputState` based on the CPU's selection bits (P14, P15) and returns the correct byte, where a `0` indicates a pressed button.

🧠 Core Flow

1. SDL event loop captures keyboard `KEYDOWN`/`KEYUP` events.
2. Mapped `SDL_Keycode` to `GbButton` using `Input.keyMappings`.
3. `Input.updateButtonState` modifies the `GbInputState` (a dictionary of button `bool`s).
4. Game Boy CPU reads `0xFF00` (P1) via `MMU.readByte`.
5. `MMU.readByte` queries the `GbInputState` based on CPU's P14/P15 selection bits and constructs the P1 register value, returning `0` for pressed buttons.

🚀 Key Takeaway

Effective emulator input handling requires correctly modeling the emulated hardware's input registers and seamlessly integrating modern input events into that model, ensuring that the emulated CPU "sees" the input as it would on real hardware.

References

1. F# Language Reference: https://learn.microsoft.com/en-us/dotnet/fsharp/
2. .NET Documentation: https://learn.microsoft.com/en-us/dotnet/
3. SDL Documentation: https://wiki.libsdl.org/

4. Pan Docs (Game Boy Technical Reference): https://gbdev.io/pandocs/
5. SDL2-CS NuGet Package: https://www.nuget.org/packages/SDL2-CS/

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 10

Advanced MMU: Memory Bank Controllers (MBCs)

Introduction

So far, our Game Boy emulator has a basic Memory Management Unit (MMU) that can handle the fixed 64KB memory map. This is sufficient for very small ROMs, but most commercial Game Boy games exceed this limit, often by megabytes. How did the original hardware manage this? Through a clever piece of hardware called a **Memory Bank Controller (MBC)**.

In this chapter, we'll extend our MMU to support MBCs. This is a critical milestone because it unlocks the ability to load and run a vast majority of Game Boy ROMs. We'll focus on implementing the **MBC1** type, which is one of the most common and fundamental MBCs. By the end of this chapter, your emulator will be able to dynamically switch between different ROM and external RAM banks, allowing it to access much larger cartridge data.

Planning & Design

The Game Boy's CPU (a custom Z80 derivative) can only address 64KB of memory at any given time. However, Game Boy cartridges can contain up to 8MB of ROM and 32KB of external RAM. To bridge this gap, MBCs act as hardware components within the cartridge that intercept memory accesses to specific regions and "bank in" different parts of the larger ROM or RAM.

Game Boy Memory Map Review

Let's quickly recap the relevant memory regions:

- **0x0000 - 0x3FFF**: ROM Bank 0 (fixed, always accessible)
- **0x4000 - 0x7FFF**: Switchable ROM Bank (controlled by MBC)
- **0xA000 - 0xBFFF**: Switchable External RAM Bank (controlled by MBC)

The key insight is that writes to specific addresses within the ROM regions (0x0000-0x7FFF) are not actually writing to ROM data. Instead, they are interpreted by the MBC as commands to switch banks, enable/disable RAM, or configure other MBC features.

MBC Architecture

We'll introduce a new F# module, `Gb.Mbc`, to encapsulate all MBC-related logic. The `Gb.Mbc` module will hold the current state of the MBC (e.g., which ROM bank is selected, if RAM is enabled) and provide functions to:

1. **Initialize:** Determine the MBC type from the cartridge header and set up initial state.
2. **Handle Writes:** Process writes to MBC control registers within the ROM address space (0x0000-0x7FFF). These writes will update the internal `MbcState`.
3. **Map Addresses:** Translate a logical Game Boy address (e.g., 0x4000) into a physical offset within the loaded cartridge ROM or external RAM data, based on the current `MbcState`.

The `Gb.Mmu` will then delegate these responsibilities to the `Gb.Mbc` module when accessing the relevant memory regions.

F# Type Design for MBCs

We'll need a way to represent different MBC types and their state. A discriminated union is perfect for this:

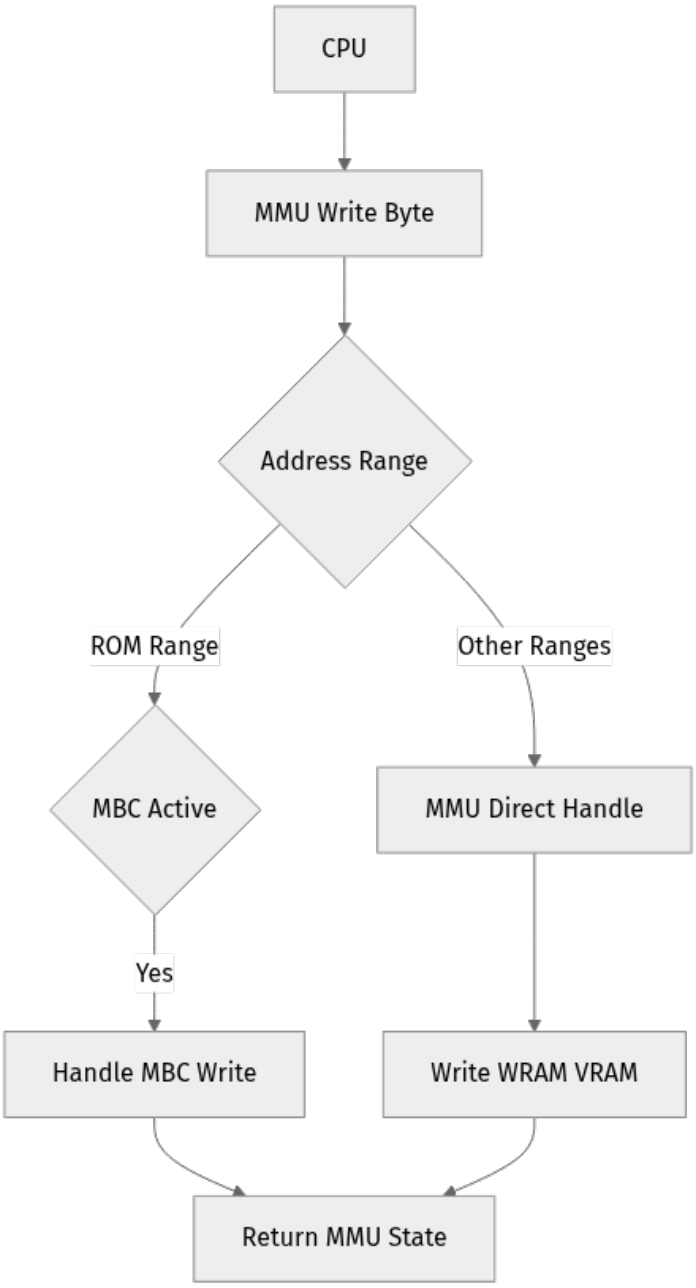
```
type MbcType =
    | NoMbc
    | Mbc1
    | Mbc1Ram
    | Mbc1RamBattery
    // ... other MBC types as we implement them

type MbcState = {
    MbcType : MbcType
    RomBanks : byte [] // The entire ROM data from the cartridge
    RamBanks : byte [] // The entire external RAM data
    CurrentRomBank : int
    CurrentRamBank : int
    RamEnabled : bool
    BankingMode : int // 0 for ROM banking, 1 for RAM banking (MBC1 specific)
}
```

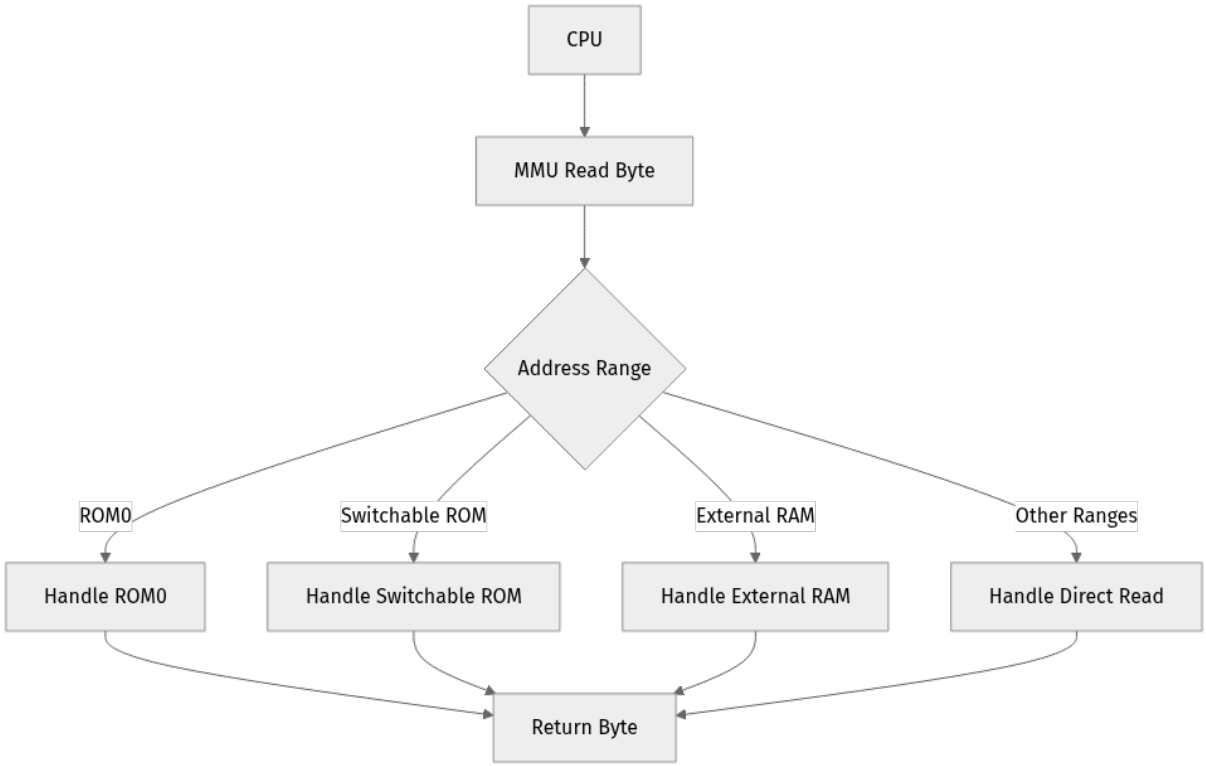
The `MbcState` will be part of our overall `Gb.Mmu` state, allowing the MMU to pass it around and update it.

Data Flow

Here's how the MMU and MBC will interact for a memory write:



And for a memory read:



Step-by-Step Implementation

We'll start by defining the `MbcState` and a helper to determine the `MbcType`. Then, we'll build out the `Gb.Mbc` module to handle MBC1 logic. Finally, we'll integrate it into our existing `Gb.Mmu`.

1. Define MBC Types and State

First, let's create a new file `Gb.Mbc.fs` in your `src/Gb` directory.

`src/Gb/Gb.Mbc.fs`

```
module Gb.Mbc

open System

/// Represents the various Memory Bank Controller types supported by Game Boy
cartridges.
type MbcType =
    | NoMbc // Cartridge without an MBC (small ROMs only)
    | Mbc1
    | Mbc1Ram
    | Mbc1RamBattery
    | Mbc2
    | Mbc2Battery
    | Mbc3
    | Mbc3Ram
    | Mbc3RamBattery
    | Mbc3TimerBattery
    | Mbc3TimerRamBattery
    | Mbc5
    | Mbc5Ram
    | Mbc5RamBattery
    | Mbc5Rumble
    | Mbc5RumbleRam
    | Mbc5RumbleRamBattery
    | MbcUnknown of byte // For unknown MBC types

/// Represents the current state of the Memory Bank Controller.
/// This state changes based on writes to specific memory addresses.
type MbcState = {
    MbcType : MbcType
    RomBanks : byte [] // The entire ROM data from the cartridge
    RamBanks : byte [] // The entire external RAM data (if any)
    CurrentRomBank : int // The currently selected switchable ROM bank
(0x4000-0x7FFF)
    CurrentRamBank :
int // The currently selected switchable external RAM bank (0xA000-0xBFFF)
    RamEnabled : bool // True if external RAM is enabled for read/write
    BankingMode : int // MBC1 specific: 0 for ROM banking mode, 1 for RAM
banking mode
}

/// Determines the MBC type from the cartridge header byte at 0x0147.
let getMbcType (mbcByte : byte) =
    match mbcByte with
    | 0x00uy -> NoMbc
    | 0x01uy -> Mbc1
    | 0x02uy -> Mbc1Ram
    | 0x03uy -> Mbc1RamBattery
    | 0x05uy -> Mbc2
    | 0x06uy -> Mbc2Battery
    | 0x0Fuy -> Mbc3TimerBattery
    | 0x10uy -> Mbc3TimerRamBattery
    | 0x11uy -> Mbc3
    | 0x12uy -> Mbc3Ram
    | 0x13uy -> Mbc3RamBattery
    | 0x19uy -> Mbc5
    | 0x1Auy -> Mbc5Ram
    | 0x1Buy -> Mbc5RamBattery
    | 0x1Cuy -> Mbc5Rumble
    | 0x1Duy -> Mbc5RumbleRam
    | 0x1Euy -> Mbc5RumbleRamBattery
```

```
    | _       -> MbcUnknown mbcByte

/// Initializes the MBC state based on the cartridge ROM and type.
let init (mbcType : MbcType) (romData : byte []) (ramSize : int) =
    let ramBanks = Array.zeroCreate ramSize // Create RAM array if RAM exists
    { MbcType = mbcType
      RomBanks = romData
      RamBanks = ramBanks
      CurrentRomBank = 1 // ROM bank 1 is default on startup (0 is fixed)
      CurrentRamBank = 0
      RamEnabled = false
      BankingMode = 0 } // Default to ROM banking mode for MBC1
```

Explanation:

- `MbcType` enumerates known MBCs. We've included many for future expansion, but will focus on `Mbc1`.
- `MbcState` is a record holding all dynamic MBC configuration.
  - `RomBanks` and `RamBanks` store the raw data.
  - `CurrentRomBank` and `CurrentRamBank` track which bank is currently mapped into the CPU's address space.
  - `RamEnabled` controls external RAM access.
  - `BankingMode` is specific to MBC1 and dictates how the `CurrentRomBank` and `CurrentRamBank` values are combined.
- `getMbcType` reads the byte at `0x0147` in the cartridge header, which defines the MBC type.
- `init` sets up the initial `MbcState` for a given cartridge. Note `CurrentRomBank` starts at `1` because `0` is fixed.

2. Implement MBC1 Logic

Now, let's add the core logic for MBC1 within the `Gb.Mbc` module. This involves functions to handle writes to control registers and to map addresses.

`src/Gb/Gb.Mbc.fs` (add to existing file)

```
// ... (previous code)

/// Handles writes to MBC control registers (addresses 0x0000-0x7FFF).
/// This function updates the MbcState based on the address and value.
let handleMbc1Write (state : MbcState) (addr : int) (value : byte) =
    match addr with
    | _ when addr >= 0x0000 && addr <= 0x1FFF -> // RAM Enable/Disable
        { state with RamEnabled = (value &&& 0x0Fuy) = 0x0Auy }

    | _ when addr >= 0x2000 && addr <= 0x3FFF -> // ROM Bank Number (Lower 5
bits)
        let newRomBank = int (value &&& 0x1Fuy) // Mask to get lower 5 bits
        let newRomBank = if newRomBank = 0 then 1 else newRomBank // Bank 0 is
always fixed, so 0 becomes 1
        let currentHigherBits = (state.CurrentRomBank >>> 5) &&& 0x03 // Get
current higher 2 bits
        let finalRomBank = (currentHigherBits <<< 5) ||| newRomBank
        { state with CurrentRomBank = finalRomBank }

    | _ when addr >= 0x4000 && addr <= 0x5FFF -> // RAM Bank Number / ROM Bank
Number (Higher 2 bits)
        let newBankValue = int (value &&& 0x03uy) // Mask to get lower 2 bits
        if state.BankingMode = 0 then // ROM Banking Mode
            let currentLowerBits = state.CurrentRomBank &&&
0x1F // Get current lower 5 bits
            let finalRomBank = (newBankValue <<< 5) ||| currentLowerBits
            { state with CurrentRomBank = finalRomBank }
        else // RAM Banking Mode
            { state with CurrentRamBank = newBankValue }

    | _ when addr >= 0x6000 && addr <= 0x7FFF -> // Banking Mode Select
        { state with BankingMode = int (value &&& 0x01uy) }

    | _ -> state // No relevant MBC1 action for other addresses

/// General MBC write handler that dispatches to specific MBC types.
let write (state : MbcState) (addr : int) (value : byte) =
    match state.MbcType with
    | Mbc1 | Mbc1Ram | Mbc1RamBattery -> handleMbc1Write state addr value
    | NoMbc -> state // No MBC, writes to ROM area are ignored
    | _ ->
        // For other MBC types or unknown, we just ignore writes to these
control registers for now
        // A full emulator would implement these specifically.
        state

/// Maps a logical ROM address (0x4000-0x7FFF) to a physical offset in the
RomBanks array.
let mapRomAddress (state : MbcState) (addr : int) =
    let romBankSize = 0x4000 // 16KB per ROM bank
    let bankOffset = (state.CurrentRomBank % (state.RomBanks.Length / romBankSi
ze)) * romBankSize
    let addressInBank = addr - 0x4000 // Offset within the 16KB bank
    bankOffset + addressInBank

/// Maps a logical external RAM address (0xA000-0xBFFF) to a physical offset in
the RamBanks array.
let mapRamAddress (state : MbcState) (addr : int) =
    if state.RamEnabled then
        let ramBankSize = 0x2000 // 8KB per RAM bank
        let bankOffset = (state.CurrentRamBank % (state.RamBanks.Length / ramBa
```

```
nkSize)) * ramBankSize
        let addressInBank = addr - 0xA000 // Offset within the 8KB bank
        Some (bankOffset + addressInBank)
    else
        None // RAM is not enabled

/// Reads a byte from the mapped external RAM, if enabled.
let readRam (state : MbcState) (addr : int) =
    match mapRamAddress state addr with
    | Some physicalAddr when physicalAddr < state.RamBanks.Length ->
        state.RamBanks.[physicalAddr]
    | _ ->
        0xFFuy // Return 0xFF if RAM is disabled or address is out of bounds
```

Explanation of MBC1 Logic:

- **RAM Enable (0x0000-0x1FFF):** Writing `0x0A` (10 decimal) to any address in this range enables external RAM. Any other value disables it. We mask the value (`value &&& 0x0Fuy`) just in case.
- **ROM Bank Number (Lower 5 bits) (0x2000-0x3FFF):** This range sets the lower 5 bits of the `CurrentRomBank`. Since ROM bank 0 is always fixed, writing `0` actually selects bank `1`. We combine these 5 bits with the higher 2 bits (set in the next range) to form the full 7-bit ROM bank number.
- **RAM Bank Number / ROM Bank Number (Higher 2 bits) (0x4000-0x5FFF):** This range's behavior depends on `BankingMode`.
    - If `BankingMode` is `0` (ROM Banking Mode), these 2 bits become the higher bits of the `CurrentRomBank`, allowing access to larger ROMs (up to 2MB with 7 bits).
    - If `BankingMode` is `1` (RAM Banking Mode), these 2 bits select one of 4 external RAM banks (up to 32KB).
- **Banking Mode Select (0x6000-0x7FFF):** Writing `0` selects ROM Banking Mode, `1` selects RAM Banking Mode.
- `mapRomAddress`: Calculates the physical offset in `RomBanks` for a given logical address in the switchable ROM bank region. It uses `CurrentRomBank`.
- `mapRamAddress`: Calculates the physical offset in `RamBanks` for a given logical address in the external RAM region, only if RAM is enabled. It returns an `option` type to indicate if RAM access is valid.
- `readRam`: A helper to safely read from external RAM.

3. Integrate MBC into MMU

Now, we need to update our `Gb.Mmu` module to use the `Gb.Mbc` functionality.

First, ensure `Gb.Mmu.fs` references `Gb.Mbc`. You might need to adjust your `.fsproj` file to ensure `Gb.Mbc.fs` is compiled before `Gb.Mmu.fs`.

`Gb.fsproj` (example order)

```
<ItemGroup>
    <Compile Include="Gb/Gb.Mbc.fs" />
    <Compile Include="Gb/Gb.Mmu.fs" />
    <Compile Include="Gb/Gb.Cpu.fs" />
    <!-- ... other files ... -->
</ItemGroup>
```

Next, modify `Gb.Mmu.fs`.

`src/Gb/Gb.Mmu.fs`

```
module Gb.Mmu

open System
open Gb.Mbc // Add this line to import the MBC module

/// Represents the Game Boy's entire memory state.
type MmuState = {
    // ... (existing fields like VRAM, WRAM, OAM, IoRegisters, etc.) ...
    BootRom : byte []
    CartridgeRom : byte [] // This will store the *entire* ROM data
    MbcState : MbcState // Add the MBC state here
}

/// Initializes the MMU with a given boot ROM and cartridge ROM.
let init (bootRomData : byte []) (cartridgeRomData : byte []) =
    let mbcType = Gb.Mbc.getMbcType cartridgeRomData.[0x0147] // Get MBC type
from cartridge header
    let ramSize =
        match cartridgeRomData.[0x0149] with // Cartridge RAM size byte
        | 0x00uy -> 0 // No RAM
        | 0x01uy -> 0x00002000 // 8KB (MBC2 specific)
        | 0x02uy -> 0x00008000 // 8KB
        | 0x03uy -> 0x00020000 // 32KB
        | 0x04uy -> 0x00080000 // 128KB
        | 0x05uy -> 0x00040000 // 64KB
        | _      -> 0 // Unknown or unsupported RAM size

    let mbcState = Gb.Mbc.init mbcType cartridgeRomData ramSize

    { // ... (existing initializations for VRAM, WRAM etc.) ...
      BootRom = bootRomData
      CartridgeRom = cartridgeRomData // Store the full ROM
      MbcState = mbcState }

/// Reads a byte from memory at the given address.
let readByte (state : MmuState) (addr : int) =
    match addr with
    // ... (existing boot ROM and other fixed memory region checks) ...
    | _ when addr >= 0x0000 && addr <= 0x3FFF -> // ROM Bank 0 (fixed)
        if state.BootRomEnabled && addr < state.BootRom.Length then
            state.BootRom.[addr]
        else
            state.CartridgeRom.[addr]

    | _ when addr >= 0x4000 && addr <= 0x7FFF -> // Switchable ROM Bank
        // Delegate to MBC to map the address
        let physicalAddr = Gb.Mbc.mapRomAddress state.MbcState addr
        state.CartridgeRom.[physicalAddr]

    | _ when addr >= 0xA000 && addr <= 0xBFFF -> // External RAM (switchable)
        match Gb.Mbc.readRam state.MbcState addr with
        | value -> value
        | exception _ -> 0xFFuy // Fallback if readRam throws (e.g., if MBC
type isn't fully implemented)

    // ... (rest of existing readByte match cases) ...

/// Writes a byte to memory at the given address.
let writeByte (state : MmuState) (addr : int) (value : byte) =
    match addr with
    // ... (existing boot ROM and other fixed memory region checks) ...
```

```
    | _ when addr >= 0x0000 && addr <= 0x7FFF -> // MBC Control Registers (ROM
area)
        // Writes to these addresses control the MBC, not the ROM data itself
        let newMbcState = Gb.Mbc.write state.MbcState addr value
        { state with MbcState = newMbcState }

    | _ when addr >= 0xA000 && addr <= 0xBFFF -> // External RAM (switchable)
        if state.MbcState.RamEnabled then
            match Gb.Mbc.mapRamAddress state.MbcState addr with
            | Some physicalAddr when physicalAddr < state.MbcState.RamBanks.Len
gth ->
                state.MbcState.RamBanks.[physicalAddr] <- value
                state // No change to MmuState, only MbcState.RamBanks is
mutable
            | _ -> state // RAM disabled or out of bounds, ignore write
        else
            state // RAM is not enabled, ignore write

    // ... (rest of existing writeByte match cases) ...
```

Key Changes in `Gb.Mmu.fs`:

1. **`open Gb.Mbc`**: Imports the new module.
2. **`MmuState` update**: Added `MbcState : MbcState` to hold the MBC's dynamic configuration.
3. **`init` function**:
   - Reads the MBC type from `cartridgeRomData.[0x0147]`.
   - Determines RAM size from `cartridgeRomData.[0x0149]`.
   - Initializes `MbcState` using `Gb.Mbc.init`.
4. **`readByte` function**:
   - For `0x4000-0x7FFF` (switchable ROM), it now calls `Gb.Mbc.mapRomAddress` to get the correct physical offset into `CartridgeRom`.
   - For `0xA000-0xBFFF` (External RAM), it calls `Gb.Mbc.readRam` to safely read from the mapped RAM if enabled.
5. **`writeByte` function**:
   - For `0x0000-0x7FFF` (ROM area), it now calls `Gb.Mbc.write` to update the `MbcState`. This is crucial as these writes are MBC commands.
   - For `0xA000-0xBFFF` (External RAM), it checks `state.MbcState.RamEnabled` and then uses `Gb.Mbc.mapRamAddress` to write to the correct location in `state.MbcState.RamBanks` (which is a mutable array).

⚡ Quick Note: Mutable RAM

Notice that `state.MbcState.RamBanks.[physicalAddr] <- value` is a mutable operation. While F# strongly favors immutability, hardware emulation often requires mutable arrays for memory regions like RAM for performance and directness. We encapsulate this mutability within the `MbcState` record and the `Gb.Mmu` module, ensuring that the mutation is explicit and contained. The `writeByte` function still returns the same `MmuState` record, but its internal `MbcState.RamBanks` array has been modified.

4. Update Cartridge Loading

Finally, ensure your main application (e.g., `Program.fs` or `Gb.Emulator.fs`) loads the full cartridge ROM data and passes it to the `Mmu.init` function. Previously, you might have only loaded the first 64KB. Now, you need to load the entire file.

`src/Gb/Gb.Emulator.fs` (or similar main file)

```
module Gb.Emulator

open System.IO
open Gb.Cpu
open Gb.Mmu
open Gb.Ppu
open Gb.Input
// ... other modules

type EmulatorState = {
    Cpu : CpuState
    Mmu : MmuState
    Ppu : PpuState
    Input : InputState
    // ... other components
    Cycles : int64}

/// Loads a Game Boy ROM file into the emulator.
let loadRom (filePath : string) =
    let romData = File.ReadAllBytes filePath // Read the entire ROM file
    let bootRomData = File.ReadAllBytes "path/to/your/
bootrom.gb" // Ensure you have a boot ROM

    let initialMmu = Mmu.init bootRomData romData
    let initialCpu = Cpu.init
    let initialPpu = Ppu.init
    let initialInput = Input.init

    { Cpu = initialCpu
      Mmu = initialMmu
      Ppu = initialPpu
      Input = initialInput
      Cycles = 0L }

// ... rest of your emulator loop
```

Testing & Verification

Now that we've implemented MBC1, it's time to test it.

1. Obtain Test ROMs

The most reliable way to test MBC functionality is using specific test ROMs, such as those from **Blargg's Game Boy CPU/MMU Tests**. You can find these by searching for "Blargg Game Boy MBC1 tests".

Specifically, look for ROMs like: * `mbc1/rom_512kb.gb` * `mbc1/rom_1mb.gb` * `mbc1/rom_2mb.gb` * `mbc1/ram_256kb.gb` (if your MBC1 implementation supports RAM)

2. Run Test ROMs

Load one of these test ROMs into your emulator. If your MBC1 implementation is correct, the ROM should boot up and display messages indicating successful bank switching or RAM access. Blargg's tests are designed to output specific messages to the serial port (which we haven't emulated yet) or to the screen. You'll primarily rely on observing the screen output.

3. Debugging Checks

- **Console Logging:** Add `printfn` statements in `Gb.Mbc.fs` within `handleMbc1Write` to log when a bank switch occurs: `fsharp // ... inside handleMbc1Write | _ when addr >= 0x2000 && addr <= 0x3FFF -> let newRomBank = int (value &&& 0x1Fuy) let newRomBank = if newRomBank = 0 then 1 else newRomBank let currentHigherBits = (state.CurrentRomBank >>> 5) &&& 0x03 let finalRomBank = (currentHigherBits <<< 5) ||| newRomBank printfn "MBC1: ROM Bank changed to %d" finalRomBank // <--- Add this { state with CurrentRomBank = finalRomBank }` This will give you visibility into whether your MBC logic is being triggered and if the `CurrentRomBank` is being updated as expected.
- **Memory Inspection:** If your emulator has a debugging mode or can dump memory, inspect the contents of `0x4000-0x7FFF` after a bank switch. The data there should change.
- **RAM Access:** If testing with MBC1+RAM, ensure that writing to `0xA000-0xBFFF` only works when `RamEnabled` is true, and that reads return the expected values.

Production Considerations

MBCs are fundamental to Game Boy emulation. Getting them right is critical for compatibility.

- **Performance:** The MMU's `readByte` and `writeByte` functions are called millions of times per second. While the indirection through `Gb.Mbc` is necessary, ensure that the banking logic within `mapRomAddress` and `mapRamAddress` is as efficient as possible. Avoid unnecessary allocations or complex computations in these hot paths.
- **Completeness:** While we focused on MBC1, a production-grade emulator needs to support MBC3 (for RTC and larger ROMs) and MBC5 (for larger ROMs and rumble). Each MBC has its own quirks and control registers.

Design your `Gb.Mbc` module to easily extend with new `MbcType` cases and corresponding handlers.

- **Maintainability:** The use of discriminated unions for `MbcType` makes it explicit which MBC is active. When adding new MBCs, you'll extend the `MbcType` union and add new functions like `handleMbc3Write` and `mapMbc3Address`, then dispatch to them from the general `write` and `read` functions in `Gb.Mbc`.

Common Issues & Solutions

1. **Incorrect Bank Calculation:**

- **Issue:** Off-by-one errors or incorrect masking when calculating `CurrentRomBank` or `CurrentRamBank`. Especially common with MBC1's `newRomBank = if newRomBank = 0 then 1 else newRomBank` rule.
- **Solution:** Double-check the bitwise operations (`&&&`, `|||`, `>>>`, `<<<`) and ensure the "ROM bank 0 becomes 1" rule is strictly followed. Use a debugger to step through the `handleMbc1Write` function and verify `CurrentRomBank` after each write. 2. **RAM Not Enabling/Disabling:**
- **Issue:** External RAM isn't behaving as expected (e.g., writes don't persist, reads return `0xFF`).
- **Solution:** Verify the `RamEnabled` flag is correctly set by writes to `0x0000-0x1FFF`. Ensure the exact magic value `0x0A` is being checked for. Also, confirm that `mapRamAddress` returns `None` when `RamEnabled` is `false`. 3. **Cartridge ROM/RAM Size Mismatch:**
- **Issue:** Emulator crashes or reads garbage when accessing ROM/RAM.
- **Solution:** Ensure `init` correctly reads the cartridge header bytes `0x0148` (ROM size) and `0x0149` (RAM size) and allocates `RomBanks` and `RamBanks` arrays of the appropriate size. My `init` function for `Gb.Mbc` uses a simplified `ramSize` calculation; for a full emulator, you'd need to map `0x0148` to actual ROM bank counts and `0x0149` to actual RAM sizes more rigorously as per Pan Docs.
    - `⚡ Real-world insight:` Cartridge header parsing is a surprisingly common source of bugs in emulators. The Pan Docs are your authoritative source for these bytes.

Summary & Next Step

In this chapter, you've taken a significant leap forward by implementing Memory Bank Controller (MBC) support, specifically for MBC1. Your emulator can now:

- Correctly identify the MBC type from a cartridge header.
- Handle writes to MBC control registers to switch ROM and RAM banks.
- Map logical Game Boy addresses to physical offsets within larger ROM and external RAM data.

This means your emulator can now run many more Game Boy games that rely on MBC1 for memory management. You've also seen how to integrate mutable state (for RAM) carefully within a functional F# codebase.

Next, we'll continue refining our emulator by diving into the **Picture Processing Unit (PPU) Part 1**, focusing on rendering the background and tiles.

🧠 Check Your Understanding

- What problem do Memory Bank Controllers solve for the Game Boy?
- Explain the role of the `BankingMode` register in MBC1.
- Why is it important to handle writes to `0x0000-0x7FFF` differently when an MBC is present?

⚡ Mini Task

- Research the MBC3 type (e.g., using Pan Docs) and list at least two key differences in its banking mechanism compared to MBC1.

🚀 Scenario

You're debugging a Game Boy ROM that uses MBC1. The game loads, displays the Nintendo logo, but then freezes on a black screen when it tries to load the main game content. You suspect an MBC issue. What are the first three things you would check in your MBC1 implementation to diagnose this problem?

📌 TL;DR

- MBCs extend Game Boy memory by banking ROM and RAM.

- MBC1 uses writes to ROM addresses as control commands for bank switching.
- `Gb.Mmu` now delegates ROM/external RAM access to `Gb.Mbc`.

🧠 Core Flow

1. MMU `init` reads cartridge header to determine MBC type and RAM size.
2. MMU `readByte` for `0x4000-0x7FFF` and `0xA000-0xBFFF` calls `Gb.Mbc` to map physical addresses.
3. MMU `writeByte` for `0x0000-0x7FFF` calls `Gb.Mbc.write` to update MBC state.

🚀 Key Takeaway

Effective emulation of hardware often requires carefully modeling memory management units and their controllers, translating logical CPU addresses to physical storage locations through stateful logic.

References

- Game Boy Pan Docs - Memory Banking Controllers
- F# Language Reference - Records
- F# Language Reference - Discriminated Unions

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 11

Audio Processing Unit (APU) Basics: Square Wave Channels

In this chapter, we're diving into the fascinating world of sound emulation for our Game Boy project. While often overlooked, a truly accurate emulator needs to replicate the distinct chiptune sounds that define the Game Boy experience. We'll start by tackling the foundational elements of the Audio Processing Unit (APU), specifically focusing on its two square wave channels.

This milestone is critical because it brings our emulator to life in a new dimension. Hearing the familiar bleeps and boops of a Game Boy game validates our CPU and MMU work in a very tangible way. By the end of this chapter, you'll have a basic APU implementation capable of generating square wave sounds, hooked into your emulator's main loop, and outputting audio via SDL2's direct audio queuing API.

Project Overview

Our overarching goal is to build a functional Game Boy emulator from scratch. So far, we've established the CPU core, memory management unit (MMU), and picture processing unit (PPU), allowing us to execute ROMs and render graphics. This chapter extends our emulator's capabilities by introducing sound, a crucial component for a complete and immersive emulation experience. We're building a modular system where each hardware component is distinct yet integrated, reflecting real hardware design.

Tech Stack

For this chapter, our primary tools remain consistent with the project:

- **F#**: The core language for its functional paradigm, which helps manage complex state transitions predictably.
- **.NET 8.0**: The runtime and SDK, providing a robust, cross-platform environment.
- **SDL2**: A cross-platform development library designed to provide low-level access to audio, keyboard, mouse, joystick, and graphics hardware. We'll specifically leverage its audio capabilities. For F#/.NET, we use `SDL2-CS` bindings, often found via NuGet as `VelcroPhysics.SDL.Net`.

Latest Versions (as of 2026-05-05):

- **F#:** The latest stable version of F# is typically included with the **.NET SDK 8.0**.
  - Official F# Language Reference: https://learn.microsoft.com/en-us/dotnet/fsharp/
- **.NET SDK:** The latest Long Term Support (LTS) version is **.NET 8.0**.
  - Download .NET SDK: https://dotnet.microsoft.com/download
- **SDL2:** The latest official SDL2 stable release is **2.30.2**. For F#/.NET, we use `SDL2-CS` (often found via NuGet as `VelcroPhysics.SDL.Net` or similar bindings).
  - Official SDL Documentation: https://wiki.libsdl.org/
  - SDL2-CS GitHub (flibitijibibo): https://github.com/flibitijibibo/SDL2-CS
- **Game Boy APU Documentation (Pan Docs):** The definitive resource for Game Boy hardware specifics.
  - Pan Docs - Sound Controller: https://gbdev.io/pandocs/Sound_Controller.html

---

APU Design and Build Plan

The Game Boy's APU is a separate hardware component responsible for generating sound. It features four distinct sound channels: two pulse (square) waves, one wave (custom waveform), and one noise channel. Each channel has its own set of registers that control its frequency, volume, duty cycle, envelope, and length.

For this chapter, we'll concentrate on the two square wave channels (often referred to as Channel 1 and Channel 2). These are controlled by specific memory-mapped I/O registers from `0xFF10` to `0xFF14` (Channel 1) and `0xFF15` to `0xFF19` (Channel 2).

Architecture Overview

The core challenge of APU emulation lies in synchronizing the sound generation with the CPU's execution. The CPU runs at ~4.19 MHz, while audio typically samples at rates like 44.1 kHz or 48 kHz. We need to generate audio samples at the correct rate based on the APU's internal state, which is updated by CPU writes to APU registers.

Our design will involve: 1. **`ApuState`:** An F# record to hold the current state of all APU channels and control registers. 2. **Square Wave Channel State:** A sub-record for each square wave channel, containing its specific parameters (frequency, duty cycle, volume, timers). 3. **Register Mapping:** Functions to handle CPU writes to APU I/O registers, parsing the bitfields and updating the `ApuState`. 4. **Sample Generation:** A function that, given the current APU state, can generate a single audio sample at the desired output sample rate. 5. **Audio Output:** Using SDL2's `SDL_QueueAudio` function to manage an audio buffer and play the generated samples directly. This is preferred over `SDL_Mixer` for raw sample streaming as it provides more granular control and avoids overhead designed for pre-recorded audio files.

Here's a high-level data flow for the APU integration:

This diagram illustrates how CPU instructions trigger memory writes, which the MMU routes to the APU. The APU's internal state is then updated, leading to sample generation and eventual output through SDL2.

F# Domain Modeling

We'll define immutable records to represent the state of our APU and its individual channels. This aligns perfectly with F#'s functional paradigm.

`SquareWaveChannel`: - `Enabled`: `bool` - `Frequency`: `int` (derived from NR13/NR14 registers) - `DutyCycle`: `int` (0-3, representing 12.5%, 25%, 50%, 75% high) - `Volume`: `int` (0-15) - `EnvelopeVolume`: `int` (current volume after envelope processing) - `EnvelopeDirection`: `int` (0: decrease, 1: increase) - `EnvelopePeriod`: `int` (how often volume changes) - `LengthCounter`: `int` (how many CPU cycles until channel stops) - `WavePosition`: `int` (internal counter for waveform generation, 0-7 for 8-step duty) - `Timer`: `int` (internal frequency timer) - `CurrentTimer`: `int` (current countdown for frequency) - `SweepPeriod`: `int` (Channel 1 only, how often frequency changes) - `SweepShift`: `int` (Channel 1 only, how much frequency changes) - `SweepDirection`: `int` (Channel 1 only, 0: increase, 1: decrease) - `ShadowFrequency`: `int` (Channel 1 only, used by sweep unit) - `SweepEnabled`: `bool` (Channel 1 only, true if sweep is active)

`ApuState`: - `Channel1`: `SquareWaveChannel` - `Channel2`: `SquareWaveChannel` - `MasterVolumeLeft`: `int` - `MasterVolumeRight`: `int` - `Enabled`: `bool` (Master APU enable, NR52 bit 7) - `SampleRate`: `int` (e.g., 44100 Hz) - `CpuCyclesPerSample`: `float` (how many CPU cycles correspond to one audio sample) - `PendingCpuCycles`: `float` (accumulator for CPU cycles, to decide

when to generate a sample) - `SampleBuffer`: `ResizeArray<float>` (buffer to store generated samples)

Setup Requirements for SDL2 Audio

To use SDL2 for audio output, you'll need the core `SDL2` library. 1. **Install `SDL2`:**

- **Windows:** Download `SDL2-devel-2.x.x-vc.zip` from the SDL releases page. Extract it and copy `SDL2.dll` (from `lib/x64` or `lib/x86`) into your project's output directory (e.g., `bin/Debug/net8.0`).
- **macOS:** `brew install sdl2`
- **Linux:** `sudo apt-get install libsdl2-dev` 2. **Add NuGet package:** In your F# application project (`GbEmulator.App` or similar), add the `VelcroPhysics.SDL.Net` NuGet package, which provides the necessary C# bindings for SDL2.

  `bash dotnet add GbEmulator.App package VelcroPhysics.SDL.Net`

Step-by-Step Implementation

We'll start by defining our APU state and basic register write logic.

0. Define Constants in `GbEmulator.Domain/Constants.fs`

First, ensure you have a `Constants.fs` file in your `GbEmulator.Domain` project with common constants, including the CPU speed.

File: `GbEmulator.Domain/Constants.fs`

```
namespace GbEmulator.Domain

module Constants =
    let CpuSpeedHz = 4_194_304 // Game Boy CPU runs at ~4.19 MHz
    let FramesPerSecond = 60
    let CyclesPerInstruction =
4 // Average cycles per instruction (varies, but 4 is a common base)
    let WramSize = 0x2000 // 8KB
    let HramSize = 0x80 // 128 bytes
    let OamSize = 0xA0 // 160 bytes
    let IoSize = 0x100 // 256 bytes for I/O registers
    let VramSize = 0x2000 // 8KB
```

📌 `Key Idea:` Centralizing constants like `CpuSpeedHz` ensures consistency across the emulator and simplifies future adjustments.

1. Define APU State in `GbEmulator.Domain/Apu.fs`

Create a new file `Apu.fs` in your `GbEmulator.Domain` project. This file will hold all the necessary types and initialization logic for our APU.

File: `GbEmulator.Domain/Apu.fs`

```
namespace GbEmulator.Domain

open System
open System.Collections.Generic // For ResizeArray

// --- Square Wave Channel State ---
type SquareWaveChannel = {
    Enabled         : bool
    Frequency       : int // Derived from NR13/NR14. Game Boy frequency is
131072 / (2048 - frequency) Hz
    DutyCycle       : int // 0=12.5%, 1=25%, 2=50%, 3=75%
    Volume          : int // Initial volume for envelope (0-15)
    EnvelopeVolume  : int // Current volume after envelope processing (0-15)
    EnvelopeDirection : int // 0=Decrease, 1=Increase
    EnvelopePeriod  :
int // Number of 1/64s of a second between volume changes (0=no envelope)
    LengthCounter   : int // Number of 1/256s of a second before channel stops
    LengthEnabled   : bool // True if length counter is enabled
    WavePosition    : int // Internal counter for waveform generation (0-7 for
8-step duty cycle)
    Timer           : int // Internal frequency timer (reload value)
    CurrentTimer    : int // Current countdown for frequency
    SweepPeriod     : int // Channel 1 only: number of 1/128s of a second
between sweep updates
    SweepShift      : int // Channel 1 only: number of bits to shift frequency
for sweep
    SweepDirection  : int // Channel 1 only: 0=Increase, 1=Decrease (frequency)
    ShadowFrequency : int // Channel 1 only: used by sweep unit
    SweepEnabled    : bool // Channel 1 only: true if sweep is active
}

// --- APU Master State ---
type ApuState = {
    Channel1        : SquareWaveChannel
    Channel2        : SquareWaveChannel
    MasterVolumeLeft : int // Master volume for left output (0-7)
    MasterVolumeRight : int // Master volume for right output (0-7)
    Enabled         : bool // Master APU enable (NR52 bit 7)
    SampleRate      : int // Target audio sample rate (e.g., 44100 Hz)
    CpuCyclesPerSample : float // How many CPU cycles correspond to one audio
sample
    PendingCpuCycles : float // Accumulator for CPU cycles, to decide when to
generate a sample
    SampleBuffer    : ResizeArray<float> // Buffer to store generated samples
}

// Initial state for a square wave channel
let private initialSquareWaveChannel = {
    Enabled = false; Frequency = 0; DutyCycle = 0; Volume = 0; EnvelopeVolume
= 0;
    EnvelopeDirection = 0; EnvelopePeriod = 0; LengthCounter = 0;
LengthEnabled = false;
    WavePosition = 0; Timer = 0; CurrentTimer = 0; SweepPeriod = 0; SweepShift
= 0;
    SweepDirection = 0; ShadowFrequency = 0; SweepEnabled = false
}

// Initialize the APU state
let initApu (sampleRate: int) = {
    Channel1 = initialSquareWaveChannel
    Channel2 = initialSquareWaveChannel
```

```
    MasterVolumeLeft = 0
    MasterVolumeRight = 0
    Enabled = false
    SampleRate = sampleRate
    CpuCyclesPerSample = Constants.CpuSpeedHz |> float / float sampleRate
    PendingCpuCycles = 0.0
    SampleBuffer = ResizeArray<float>()
}
```

📌 `Key Idea:` We define detailed state records for each square wave channel and the overall APU. This structured approach makes it easier to manage the complex interactions of the APU registers, leveraging F#'s immutability for predictable state changes. `ResizeArray<float>` is chosen for the sample buffer to allow efficient appending of new samples without constant reallocations, which is important for performance-critical audio loops.

2. Add APU to Emulator State

To integrate the APU into our emulator, its state must be part of the global `EmulatorState`. Open `GbEmulator.Core/Emulator.fs` and add `ApuState` to your `EmulatorState` record and the `initEmulator` function.

File: `GbEmulator.Core/Emulator.fs`

```
namespace GbEmulator.Core

open GbEmulator.Domain
open GbEmulator.Domain.Cpu
open GbEmulator.Domain.Memory
open GbEmulator.Domain.Ppu
open GbEmulator.Domain.Apu // Add this line

// ... other record definitions

type EmulatorState = {
    CpuState: CpuState
    MmuState: MmuState
    PpuState: PpuState
    ApuState: ApuState // Add this field
    // ... other fields
    Running: bool
}

// ... in initEmulator function ...
let initEmulator (romBytes: byte[]) (sampleRate: int) =
    let mmu = Mmu.initMmu romBytes
    let cpu = Cpu.initCpu
    let ppu = Ppu.initPpu
    let apu = Apu.initApu sampleRate // Initialize APU with the target sample
rate

    { CpuState = cpu; MmuState = mmu; PpuState = ppu; ApuState = apu; Running
= true } // Add apu
```

⚡ `Quick Note:` We pass the `sampleRate` to `Apu.initApu` because the APU needs to know the target audio output rate to calculate `CpuCyclesPerSample` accurately. This is crucial for correct timing synchronization.

3. Handle APU Register Writes in `GbEmulator.Domain/Mmu.fs`

The MMU is responsible for routing memory writes to the correct hardware components. We need to extend `Mmu.writeByte` to update the APU state when writes occur to the APU's I/O registers (`0xFF10` - `0xFF3F`).

First, in `GbEmulator.Domain/Apu.fs`, add functions to update the APU state based on register writes. For now, we'll focus on the raw register values, and later derive the actual sound parameters.

File: `GbEmulator.Domain/Apu.fs` (continued)

```
// GbEmulator.Domain/Apu.fs (continued)

// Helper for bit manipulation
let private isBitSet value bit = (value &&& (1 <<< bit)) <> 0

// --- APU Register Write Handlers ---

// NR10 - Channel 1 Sweep register
let writeNr10 (value: byte) (channel: SquareWaveChannel) =
    { channel with
        SweepPeriod = (value >>> 4) &&& 0x7 // Bits 4-6
        SweepDirection = (value >>> 3) &&& 0x1 // Bit 3 (0=increase,
1=decrease)
        SweepShift = value &&& 0x7 // Bits 0-2
    }

// NR11/NR21 - Channel 1/2 Length & Duty Cycle
let writeNrX1 (value: byte) (channel: SquareWaveChannel) =
    { channel with
        DutyCycle = (value >>> 6) &&& 0x3 // Bits 6-7
        LengthCounter = 64 - (value &&& 0x3F) // Bits 0-5. Length is 64 - N.
    }

// NR12/NR22 - Channel 1/2 Volume & Envelope
let writeNrX2 (value: byte) (channel: SquareWaveChannel) =
    { channel with
        Volume = (value >>> 4) &&& 0xF // Bits 4-7
        EnvelopeDirection = (value >>> 3) &&& 0x1 // Bit 3 (0=decrease,
1=increase)
        EnvelopePeriod = value &&& 0x7 // Bits 0-2
        // If initial volume is 0, channel is muted immediately.
        Enabled = ((value >>> 4) &&& 0xF) <> 0 // Channel enabled only if
initial volume is not 0
        EnvelopeVolume = (value >>> 4) &&& 0xF // Set current envelope volume
to initial volume
    }

// NR13/NR23 - Channel 1/2 Frequency Low (lower 8 bits of 11-bit frequency)
let writeNrX3 (value: byte) (channel: SquareWaveChannel) =
    // The frequency is an 11-bit value, so we combine with the upper bits from
NR14
    let newFrequency = (channel.Frequency &&& 0x700) ||| (int value) // Keep
upper 3 bits, set lower 8
    { channel with
        Frequency = newFrequency
        // Timer calculation: (2048 - frequency) * 4 CPU cycles. This is the
period of one full waveform.
        Timer = (2048 - newFrequency) * 4
        CurrentTimer = (2048 - newFrequency) * 4 // Reset current timer
    }

// NR14/NR24 - Channel 1/2 Frequency High (upper 3 bits) & Control
let writeNrX4 (value: byte) (channel: SquareWaveChannel) =
    // Update upper 3 bits of frequency
    let newFrequency = (channel.Frequency &&& 0xFF) ||| (((int (value &&&
0x7)) <<< 8)) // Bits 0-2
    { channel with
        Frequency = newFrequency
        LengthEnabled = isBitSet value 6 // Bit 6: Use length counter
        // Trigger bit (Bit 7): When set, reloads length counter, volume
envelope, and frequency timer.
```

```
        // This is where the channel is "restarted".
        Enabled = if isBitSet value 7 then ((channel.Volume &&& 0xF) <> 0)
else channel.Enabled // Trigger enables if volume > 0
        LengthCounter = if isBitSet value 7 then (if channel.LengthCounter = 0
then 64 else channel.LengthCounter) else channel.LengthCounter
        EnvelopeVolume = if isBitSet value 7 then channel.Volume else channel.E
nvelopeVolume
        CurrentTimer = if isBitSet value 7 then (2048 - newFrequency) * 4 else
channel.CurrentTimer
        Timer = if isBitSet value 7 then (2048 - newFrequency) * 4 else
channel.Timer
        ShadowFrequency = if isBitSet value 7 then newFrequency else channel.Sh
adowFrequency // For sweep unit
        SweepEnabled = if isBitSet value 7 then (channel.SweepPeriod <> 0 || ch
annel.SweepShift <> 0) else channel.SweepEnabled
    }

// NR50 - Channel Control (Volume & Vin Panning)
let writeNr50 (value: byte) (apu: ApuState) =
    { apu with
        MasterVolumeLeft = (value >>> 4) &&& 0x7 // Bits 4-6
        MasterVolumeRight = value &&& 0x7 // Bits 0-2
    }

// NR51 - Selection of Sound Output Terminal
let writeNr51 (value: byte) (apu: ApuState) =
    // For now, we'll ignore panning, but this register controls which channels
go to which speaker.
    // Bit 0: Channel 1 to Right, Bit 1: Channel 2 to Right, etc.
    // Bit 4: Channel 1 to Left, Bit 5: Channel 2 to Left, etc.
    // This is a simplification; a real implementation would use this to mix
channels.
    apu

// NR52 - Sound ON/OFF
let writeNr52 (value: byte) (apu: ApuState) =
    let newEnabled = isBitSet value 7
    if not newEnabled then
        // If master APU is disabled, all channels are disabled and their
states are reset.
        // This is a simplification for now. A full reset would zero out all
channel registers.
        { apu with
            Enabled = newEnabled
            Channel1 = { apu.Channel1 with Enabled = false }
            Channel2 = { apu.Channel2 with Enabled = false }
        }
    else
        { apu with Enabled = newEnabled }

// Main APU write function: Routes writes to specific channel handlers
let writeByte (addr: int) (value: byte) (apu: ApuState) : ApuState =
    match addr with
    | 0xFF10 -> { apu with Channel1 = writeNr10 value apu.Channel1 } // NR10 -
Channel 1 Sweep
    | 0xFF11 -> { apu with Channel1 = writeNrX1 value apu.Channel1 } // NR11 -
Channel 1 Length & Duty
    | 0xFF12 -> { apu with Channel1 = writeNrX2 value apu.Channel1 } // NR12 -
Channel 1 Volume & Envelope
    | 0xFF13 -> { apu with Channel1 = writeNrX3 value apu.Channel1 } // NR13 -
Channel 1 Frequency Low
    | 0xFF14 -> { apu with Channel1 = writeNrX4 value apu.Channel1 } // NR14 -
```

```
Channel 1 Frequency High & Control
    | 0xFF16 -> { apu with Channel2 = writeNrX1 value apu.Channel2 } // NR21 -
Channel 2 Length & Duty
    | 0xFF17 -> { apu with Channel2 = writeNrX2 value apu.Channel2 } // NR22 -
Channel 2 Volume & Envelope
    | 0xFF18 -> { apu with Channel2 = writeNrX3 value apu.Channel2 } // NR23 -
Channel 2 Frequency Low
    | 0xFF19 -> { apu with Channel2 = writeNrX4 value apu.Channel2 } // NR24 -
Channel 2 Frequency High & Control
    | 0xFF20 -> apu // NR30 - Channel 3 Sound On/Off (Wave Channel) -
Placeholder
    | 0xFF21 -> apu // NR31 - Channel 3 Length (Wave Channel) - Placeholder
    | 0xFF22 -> apu // NR32 - Channel 3 Volume (Wave Channel) - Placeholder
    | 0xFF23 -> apu // NR33 - Channel 3 Frequency Low (Wave Channel) -
Placeholder
    | 0xFF24 -> apu // NR34 - Channel 3 Frequency High (Wave Channel) -
Placeholder
    | 0xFF25 -> writeNr50 value apu // NR50 - Channel Control (Volume & Vin
Panning)
    | 0xFF26 -> writeNr51 value apu // NR51 - Selection of Sound Output
Terminal
    | 0xFF27 -> writeNr52 value apu // NR52 - Sound ON/OFF
    | 0xFF30 .. 0xFF3F -> apu // Wave RAM - Placeholder for Channel 3's
waveform data
    | _ -> apu
```

🧠 `Important:` The `writeNrX4` (trigger register) is crucial. When bit 7 is set, it effectively "restarts" the channel, reloading its length counter, volume envelope, and frequency timer. This is a common Game Boy sound programming technique. We also set `Enabled = true` on trigger, but the channel can still be muted if its initial volume (NRx2) is zero.

Now, integrate this into `GbEmulator.Domain/Mmu.fs`. This requires modifying the `writeByte` function to accept and return `ApuState` alongside `MmuState`.

File: `GbEmulator.Domain/Mmu.fs`

```
namespace GbEmulator.Domain

open GbEmulator.Domain.Apu // Add this line
// ... other open statements

type MmuState = {
    // ... existing fields
    Rom: byte[]
    Wram: byte[]
    Hram: byte[]
    Oam: byte[]
    Io: byte[] // For I/O registers
    Vram: byte[] // Ensure Vram is here if not already
}

// ... in initMmu ...
let initMmu (romBytes: byte[]) =
    {
        // ... existing initialization
        Rom = romBytes
        Wram = Array.zeroCreate Constants.WramSize
        Hram = Array.zeroCreate Constants.HramSize
        Oam = Array.zeroCreate Constants.OamSize
        Io = Array.zeroCreate Constants.IoSize // Initialize I/O registers
        Vram = Array.zeroCreate Constants.VramSize // Initialize Vram
    }

// Modify the writeByte function to handle APU registers
let writeByte (addr: int) (value: byte) (mmu: MmuState) (apu: ApuState) : MmuSt
ate * ApuState =
    match addr with
    | 0x0000 .. 0x7FFF -> // ROM
        // Handle MBC banking here later
        (mmu, apu)
    | 0x8000 .. 0x9FFF -> // VRAM
        let newVram = Array.copy mmu.Vram
        newVram.[addr - 0x8000] <- value
        ({ mmu with Vram = newVram }, apu)
    | 0xA000 .. 0xBFFF -> // External RAM
        // Handle external RAM banking here later
        (mmu, apu)
    | 0xC000 .. 0xDFFF -> // WRAM
        let newWram = Array.copy mmu.Wram
        newWram.[addr - 0xC000] <- value
        ({ mmu with Wram = newWram }, apu)
    | 0xE000 .. 0xFDFF -> // Echo RAM (mirror of C000-DDFF)
        // Writes to echo RAM also write to WRAM
        let wramAddr = addr - 0xE000 + 0xC000
        let newWram = Array.copy mmu.Wram
        newWram.[wramAddr - 0xC000] <- value
        ({ mmu with Wram = newWram }, apu)
    | 0xFE00 .. 0xFE9F -> // OAM
        let newOam = Array.copy mmu.Oam
        newOam.[addr - 0xFE00] <- value
        ({ mmu with Oam = newOam }, apu)
    | 0xFF00 -> // P1 - Joypad
        // Joypad is read-only for writes, except for selecting button/
direction bits
        ({ mmu with Io = (Array.copy mmu.Io) |> fun arr -> arr.[0x00] <- (arr.
[0x00] &&& 0xCF) ||| (value &&& 0x30); arr }, apu)
    | 0xFF01 -> // SB - Serial Transfer Data
```

```
        ({ mmu with Io = (Array.copy mmu.Io) |> fun arr -> arr.[0x01] <-
value; arr }, apu)
    | 0xFF04 -> // DIV - Divider Register
        // Resetting DIV is a special case, it always writes 0
        ({ mmu with Io = (Array.copy mmu.Io) |> fun arr -> arr.[0x04] <- 0x00;
arr }, apu) // DIV is reset to 0 on write
    | 0xFF05 -> // TIMA - Timer Counter
        ({ mmu with Io = (Array.copy mmu.Io) |> fun arr -> arr.[0x05] <-
value; arr }, apu)
    | 0xFF06 -> // TMA - Timer Modulo
        ({ mmu with Io = (Array.copy mmu.Io) |> fun arr -> arr.[0x06] <-
value; arr }, apu)
    | 0xFF07 -> // TAC - Timer Control
        ({ mmu with Io = (Array.copy mmu.Io) |> fun arr -> arr.[0x07] <-
value; arr }, apu)
    | 0xFF0F -> // IF - Interrupt Flag
        ({ mmu with Io = (Array.copy mmu.Io) |> fun arr -> arr.[0x0F] <-
value; arr }, apu)
    | 0xFF10 .. 0xFF27 -> // APU Registers
        // Route APU register writes to the APU module
        (mmu, Apu.writeByte addr value apu)
    | 0xFF28 .. 0xFF2F -> // Unused I/O (APU)
        (mmu, apu)
    | 0xFF30 .. 0xFF3F -> // Wave RAM (APU)
        // Route Wave RAM writes to the APU module (for Channel 3)
        // For now, these are placeholders, but Apu.writeByte will handle them
eventually.
        (mmu, Apu.writeByte addr value apu)
    | 0xFF40 .. 0xFF4B -> // PPU Registers
        // Update the PPU state based on register writes
        let (newIo, newLcdControl, newScrollY, newScrollX, newWindowY, newWindo
wX) =
            Ppu.writePpuRegister (addr - 0xFF40) value mmu.Io
        // Update MMU's Io array for the PPU register
        let updatedIo = (Array.copy mmu.Io) |> fun arr -> arr.[addr - 0xFF00] <
- value; arr
        ( { mmu with Io = updatedIo }, apu)
    | 0xFF4C .. 0xFF7F -> // Unused I/O, but some are used in CGB mode
        ({ mmu with Io = (Array.copy mmu.Io) |> fun arr -> arr.[addr - 0xFF00]
<- value; arr }, apu)
    | 0xFF80 .. 0xFFFE -> // HRAM
        let newHram = Array.copy mmu.Hram
        newHram.[addr - 0xFF80] <- value
        ({ mmu with Hram = newHram }, apu)
    | 0xFFFF -> // IE - Interrupt Enable
        ({ mmu with Io = (Array.copy mmu.Io) |> fun arr -> arr.[0xFF] <-
value; arr }, apu)
    | _ ->
        printfn "MMU: Unhandled write to 0x%x with value 0x%x" addr value
        (mmu, apu)
```

⚡ Quick Note: `Mmu.writeByte` now returns `MmuState * ApuState`. This is a common functional pattern for updating multiple related states. This means the `Cpu.executeInstruction` function will need to be refactored to accept and return `ApuState` as well, ensuring APU state is always current. We also added a small fix for the joypad write to only allow writing to the upper nibble (direction/button select).

4. Implement Sample Generation and APU Tick

Back in `GbEmulator.Domain/Apu.fs`, we need functions to advance the APU's internal state and generate audio samples.

File: `GbEmulator.Domain/Apu.fs` (continued)

```
// GbEmulator.Domain/Apu.fs (continued)

// Duty cycle waveforms (8 steps, true=high, false=low)
let private dutyCycles =
    [|
        [|false; false; false; false; false; false; false; true|] // 12.5%
        [|true; false; false; false; false; false; false; true|] // 25%
        [|true; false; false; false; false; true; false; true|] // 50%
        [|false; true; true; true; true; true; true; false|] // 75%
    |]

// Generate a single sample for a square wave channel
let generateSquareWaveSample (channel: SquareWaveChannel) : float =
    if not channel.Enabled || channel.EnvelopeVolume = 0 then
        0.0 // Muted if channel disabled or volume is 0
    else
        let dutyPattern = dutyCycles.[channel.DutyCycle]
        // Normalize volume to a float between 0 and 1
        let normalizedVolume = float channel.EnvelopeVolume / 15.0
        if dutyPattern.[channel.WavePosition] then
            normalizedVolume // High state
        else
            -normalizedVolume // Low state (for bipolar output, centered at 0)

// Update internal timers and generate samples
let tick (cpuCycles: int) (apu: ApuState) : ApuState =
    if not apu.Enabled then
        // If master APU is disabled, clear sample buffer and return
        apu.SampleBuffer.Clear()
        { apu with PendingCpuCycles = 0.0 }
    else
        let mutable currentApu = apu
        let mutable currentChannel1 = apu.Channel1
        let mutable currentChannel2 = apu.Channel2

        // Accumulate CPU cycles
        currentApu <- { currentApu with PendingCpuCycles = currentApu.PendingCp
uCycles + float cpuCycles }

        // Process frame sequencer (length counter, envelope, sweep -
simplified for now)
        // This is a complex part of the APU, typically running at 512 Hz
(every 8192 CPU cycles)
        // For simplicity, we'll implement a basic frequency timer and waveform
step.
        // Full frame sequencer logic is for a later, more accurate APU
chapter.

        // --- Channel 1 & 2 Frequency Timer and Wave Position Update ---
        let updateChannelTimer (channel: SquareWaveChannel) =
            if channel.Enabled && channel.Timer > 0 then // Only tick if
enabled and timer is valid
                let newCurrentTimer = channel.CurrentTimer - cpuCycles
                if newCurrentTimer <= 0 then
                    // Timer expired, reload and advance wave position
                    let timerReload = channel.Timer // Use the pre-calculated
timer value
                    let newWavePosition = (channel.WavePosition + 1) % 8
                    { channel with CurrentTimer = timerReload +
newCurrentTimer; WavePosition = newWavePosition }
                else
```

```
                    { channel with CurrentTimer = newCurrentTimer }
            else
                channel // Not enabled or timer is 0, no change

        currentChannel1 <- updateChannelTimer currentChannel1
        currentChannel2 <- updateChannelTimer currentChannel2

        // --- Sample Generation Loop ---
        while currentApu.PendingCpuCycles >= currentApu.CpuCyclesPerSample do
            // Generate a sample from each enabled channel
            let sample1 = generateSquareWaveSample currentChannel1
            let sample2 = generateSquareWaveSample currentChannel2

            // Basic mixing: sum and clip (or normalize)
            let mixedSample = (sample1 + sample2) / 2.0 // Simple average for
now
            currentApu.SampleBuffer.Add(mixedSample)

            currentApu <- { currentApu with PendingCpuCycles = currentApu.Pendi
ngCpuCycles - currentApu.CpuCyclesPerSample }

        { currentApu with Channel1 = currentChannel1; Channel2 = currentChannel
2 }
```

⚠️ `What can go wrong:` The `tick` function for the APU is extremely simplified here. A real Game Boy APU has a complex "frame sequencer" that manages length counters, volume envelopes, and sweep units at specific clock rates (e.g., 512 Hz). Our current `tick` only advances the frequency timer and waveform position. This will produce static square waves, not the dynamic sounds with fading volumes and changing pitches found in real games. We'll address this in a future chapter for higher accuracy.

⚡ `Real-world insight:` The Game Boy's APU is notoriously tricky to emulate accurately due to its tight synchronization requirements and the specific behavior of its frame sequencer. Many emulators start with this simplified approach and gradually add more cycle-accurate logic for length, envelope, and sweep.

5. Update Emulator's Main Loop

To ensure `ApuState` is consistently updated, we need to pass it through functions that modify memory-mapped I/O registers. This means refactoring `Cpu.executeInstruction` to accept and return `ApuState`.

First, refactor `GbEmulator.Domain/Cpu.fs`:

Locate your `Cpu.executeInstruction` function and modify its signature and internal calls to `Mmu.writeByte`.

File: `GbEmulator.Domain/Cpu.fs` (partial, illustrating refactor)

```
namespace GbEmulator.Domain

open GbEmulator.Domain.Memory
open GbEmulator.Domain.Apu
// ... other open statements

// Define the return type to include ApuState
type CpuExecutionResult = {
    CpuState: CpuState
    MmuState: MmuState
    ApuState: ApuState // New: Include ApuState in the result
    Cycles: int
}

// Refactor executeInstruction signature
let executeInstruction (cpu: CpuState) (mmu: MmuState) (apu: ApuState) : CpuExe
cutionResult =
    // ... existing instruction fetch and decode logic ...

    // Example: When an instruction performs a memory write (e.g., LD (HL), A)
    // Assume `addr` and `value` are determined by the instruction
    let (newMmu, newApu) = Mmu.writeByte addr value currentMmu currentApu //
Pass and receive ApuState

    // ... continue with instruction execution, updating cpu state, mmu, and
apu ...

    // When returning the result:
    { CpuState = updatedCpu; MmuState = newMmu; ApuState = newApu; Cycles = ins
tructionCycles }
```

⚡ `Real-world insight:` Propagating state updates through multiple modules (`Cpu`, `Mmu`, `Apu`) in a functional way requires careful thought. In F#, you typically pass the current state to a function and it returns a new state. This is why `Mmu.writeByte` now returns `MmuState * ApuState`, and `Cpu.executeInstruction` needs to incorporate `ApuState` in its signature and return type. This ensures the `ApuState` reflects any register writes immediately.

Now, update `GbEmulator.Core/Emulator.fs`:

Modify the `step` function to correctly call the refactored `Cpu.executeInstruction` and handle the updated `ApuState`.

File: `GbEmulator.Core/Emulator.fs` (continued)

```
namespace GbEmulator.Core

open GbEmulator.Domain
open GbEmulator.Domain.Cpu
open GbEmulator.Domain.Memory
open GbEmulator.Domain.Ppu
open GbEmulator.Domain.Apu
open System

let step (state: EmulatorState) : EmulatorState =
    // ... existing code ...

    // Call the refactored Cpu.executeInstruction with ApuState
    let cpuExecResult = Cpu.executeInstruction state.CpuState state.MmuState st
ate.ApuState
    let newCpuState = cpuExecResult.CpuState
    let newMmuState = cpuExecResult.MmuState
    let updatedApuFromCpu = cpuExecResult.ApuState // APU state after CPU
writes (via MMU)

    let newPpuState = Ppu.tick cpuExecResult.Cycles state.PpuState
newMmuState.Io // PPU tick
    let finalApuState = Apu.tick cpuExecResult.Cycles updatedApuFromCpu // APU
tick for internal timing

    { state with
        CpuState = newCpuState
        MmuState = newMmuState
        PpuState = newPpuState
        ApuState = finalApuState // Update with the final APU state
    }
```

6. SDL2 Audio Integration in `GbEmulator.App/Program.fs`

This is where we actually play the sound. We'll use SDL2's direct audio queuing.

File: `GbEmulator.App/Program.fs`

```
namespace GbEmulator.App

open System
open System.Runtime.InteropServices
open SDL2
open GbEmulator.Core
open GbEmulator.Domain
open GbEmulator.Domain.Ppu
open System.IO // For File.ReadAllBytes

// --- SDL2 Audio P/Invoke declarations ---
[<DllImport("SDL2.dll", CallingConvention = CallingConvention.Cdecl)>]
extern uint SDL_OpenAudioDevice(string device, int iscapture, ref SDL.SDL_Audio
Spec desired, out SDL.SDL_AudioSpec obtained, int allowed_changes)

[<DllImport("SDL2.dll", CallingConvention = CallingConvention.Cdecl)>]
extern int SDL_QueueAudio(uint dev, IntPtr data, uint len)

[<DllImport("SDL2.dll", CallingConvention = CallingConvention.Cdecl)>]
extern uint SDL_GetQueuedAudioSize(uint dev)

[<DllImport("SDL2.dll", CallingConvention = CallingConvention.Cdecl)>]
extern void SDL_PauseAudioDevice(uint dev, int pause_on)

[<DllImport("SDL2.dll", CallingConvention = CallingConvention.Cdecl)>]
extern void SDL_CloseAudioDevice(uint dev)

// Define audio parameters
let AudioFrequency = 44100 // Hz, common sample rate
let AudioChannels = 2 // Stereo output
let AudioBufferSize = 2048 // Samples per internal SDL buffer (power of 2 is
good)
let AudioFormat = 0x8010us // AUDIO_S16SYS (Signed 16-bit, system endian)

// --- Main Program Loop ---
[<EntryPoint>]
let main argv =
    // Check for ROM path argument
    if argv.Length = 0 then
        printfn "Usage: dotnet run --project GbEmulator.App --
<path_to_rom.gb>"
        exit 1

    // ... existing SDL initialization ...
    if SDL.SDL_Init(SDL.SDL_INIT_VIDEO ||| SDL.SDL_INIT_AUDIO) < 0 then
        printfn "SDL_Init Error: %s" (SDL.SDL_GetError())
        exit 1

    // Global variable for audio device ID
    let mutable audioDeviceId: uint = 0u

    let desiredAudioSpec =
        { SDL.SDL_AudioSpec.freq = AudioFrequency
          SDL.SDL_AudioSpec.format = AudioFormat
          SDL.SDL_AudioSpec.channels = byte AudioChannels
          SDL.SDL_AudioSpec.samples = UInt16 AudioBufferSize // Buffer size in
samples
          SDL.SDL_AudioSpec.padding = 0us
          SDL.SDL_AudioSpec.size = 0u
          SDL.SDL_AudioSpec.callback = IntPtr.Zero // No callback for queueing
          SDL.SDL_AudioSpec.userdata = IntPtr.Zero
```

```
        }

    let mutable obtainedAudioSpec = desiredAudioSpec // Will be filled by SDL
    audioDeviceId <- SDL_OpenAudioDevice(null, 0, &desiredAudioSpec, &obtainedA
udioSpec, SDL.SDL_AUDIO_ALLOW_FORMAT_CHANGE)

    if audioDeviceId = 0u then
        printfn "SDL_OpenAudioDevice Error: %s" (SDL.SDL_GetError())
        SDL.SDL_Quit()
        exit 1

    printfn "SDL_AudioDevice opened successfully. Frequency: %d, Channels: %d"
obtainedAudioSpec.freq (int obtainedAudioSpec.channels)

    // Start audio playback
    SDL_PauseAudioDevice(audioDeviceId, 0) // Unpause

    // ... create window and renderer ...
    let window = SDL.SDL_CreateWindow("Game Boy Emulator", SDL.SDL_WINDOWPOS_CE
NTERED, SDL.SDL_WINDOWPOS_CENTERED, Ppu.ScreenWidth * 3, Ppu.ScreenHeight * 3,
SDL.SDL_WindowFlags.SDL_WINDOW_SHOWN)
    let renderer = SDL.SDL_CreateRenderer(window, -1, SDL.SDL_RendererFlags.SDL
_RENDERER_ACCELERATED)
    let texture = SDL.SDL_CreateTexture(renderer,
SDL.SDL_PIXELFORMAT_ARGB8888, int SDL.SDL_TextureAccess.SDL_TEXTUREACCESS_STREA
MING, Ppu.ScreenWidth, Ppu.ScreenHeight)

    let romPath = argv.[0]
    let romBytes = File.ReadAllBytes romPath
    let mutable emulatorState = Emulator.initEmulator romBytes AudioFrequency /
/ Pass sample rate to APU

    let mutable lastFrameTime = DateTime.Now

    let rec loop (state: EmulatorState) =
        // ... event handling ...
        let event = SDL.SDL_Event()
        if SDL.SDL_PollEvent(&event) <> 0 then
            match event.type with
            | SDL.SDL_EventType.SDL_QUIT -> { state with Running = false }
            | SDL.SDL_EventType.SDL_KEYDOWN ->
                match event.key.keysym.sym with
                | SDL.SDL_Keycode.SDLK_ESCAPE -> { state with Running = false }
                | _ -> state // Handle other key presses for input
            | _ -> state
        else
            let now = DateTime.Now
            let deltaTime = (now - lastFrameTime).TotalSeconds
            lastFrameTime <- now

            // Run emulator steps
            let mutable currentState = state
            // Aim for 60 FPS. CPU Speed Hz / 60 frames per second = cycles per
frame.
            // Run in chunks to avoid single massive loop.
            let cyclesPerFrame = Constants.CpuSpeedHz / Constants.FramesPerSeco
nd
            let mutable cyclesThisFrame = 0

            while cyclesThisFrame < cyclesPerFrame do
                // Execute one CPU instruction, updating CPU, MMU, and APU
states
```

```
                let execResult = Cpu.executeInstruction currentState.CpuState c
urrentState.MmuState currentState.ApuState

                // Update PPU and APU with the cycles consumed by the
instruction
                let newPpuState = Ppu.tick execResult.Cycles currentState.PpuSt
ate currentState.MmuState.Io
                let newApuState = Apu.tick execResult.Cycles execResult.ApuStat
e // APU tick after CPU cycles

                currentState <- { currentState with
                                    CpuState = execResult.CpuState
                                    MmuState = execResult.MmuState
                                    PpuState = newPpuState
                                    ApuState = newApuState // Update with the
final APU state
                                }
                cyclesThisFrame <- cyclesThisFrame + execResult.Cycles

            // PPU rendering
            Ppu.renderFrame currentState.PpuState renderer texture

            // Audio output: Copy samples from APU buffer to SDL2 audio queue
            let apuState = currentState.ApuState
            let bytesQueued = SDL_GetQueuedAudioSize(audioDeviceId) // Get
amount of audio currently in SDL's queue
            let maxQueueBytes = uint32 (AudioFrequency * AudioChannels *
sizeof<int16> * 2 / 60) // Roughly 2 frames of audio
            // Queue more audio only if the buffer isn't too full, to avoid
latency
            if apuState.SampleBuffer.Count > 0 && bytesQueued < maxQueueBytes t
hen
                // Convert float samples to int16
                let samplesToCopy = apuState.SampleBuffer.Count
                let audioData = Array.zeroCreate<int16>(samplesToCopy * AudioCh
annels) // Stereo output

                for i = 0 to samplesToCopy - 1 do
                    let sampleFloat = apuState.SampleBuffer.[i]
                    let sampleInt16 = (sampleFloat * 32767.0) |>
int16 // Scale to 16-bit signed range
                    audioData.[i * AudioChannels] <- sampleInt16 // Left
channel
                    audioData.[i * AudioChannels + 1] <- sampleInt16 // Right
channel (simple stereo)

                // Queue audio data
                use handle = GCHandle.alloc audioData GCHandleType.Pinned
                let ptr = handle.AddrOfPinnedObject()
                let result = SDL_QueueAudio(audioDeviceId, ptr, uint
(audioData.Length * sizeof<int16>()))
                if result < 0 then
                    printfn "SDL_QueueAudio Error: %s" (SDL.SDL_GetError())

                apuState.SampleBuffer.Clear() // Clear APU's internal buffer
after queueing

            // Limit frame rate to 60 FPS
            let frameDuration = (DateTime.Now - now).TotalMilliseconds
            let targetFrameTime = 1000.0 / float Constants.FramesPerSecond
            if frameDuration < targetFrameTime then
                SDL.SDL_Delay(uint32 (targetFrameTime - frameDuration))
```

```
            { currentState with Running = true } // Continue loop

    // Start the main emulator loop
    let finalState = loop emulatorState

    // ... existing SDL cleanup ...
    SDL_PauseAudioDevice(audioDeviceId, 1) // Pause audio
    SDL_CloseAudioDevice(audioDeviceId) // Close audio device
    SDL.SDL_DestroyTexture(texture)
    SDL.SDL_DestroyRenderer(renderer)
    SDL.SDL_DestroyWindow(window)
    SDL.SDL_Quit()
    0
```

🔥 `Optimization / Pro tip:` Using `SDL_QueueAudio` directly is generally preferred for emulators because it gives you precise control over the raw sample data and streaming. `SDL_mixer` is more suited for playing pre-made sound effects and music files, not for generating samples on the fly. The `GCHandle.alloc` is used to pin the F# array in memory so its address can be passed to the unmanaged SDL function. The `use` keyword ensures `handle.Free()` is called automatically when the handle goes out of scope, preventing memory leaks. We've also added a check for `SDL_GetQueuedAudioSize` to prevent overfilling the audio buffer, which can cause latency or glitches.

Testing & Verification

With the APU integrated, it's time to hear our work!

1. **Run with a ROM that has simple sounds:**
    - Good candidates: Tetris, Dr. Mario, Alleyway. These games use square waves extensively for their music and sound effects.
    - Command: `dotnet run --project GbEmulator.App -- path/to/tetris.gb`
2. **Expected Behavior:**
    - You should hear basic square wave sounds. The pitch might be correct, but the volume envelopes and length counters will be very basic or non-existent in this initial implementation.
    - Music might sound flat or continuous, lacking the dynamic changes you'd expect.

3. **Debugging Checks:**
    - **No sound at all?**
        - Check `SDL_Init` and `SDL_OpenAudioDevice` return values in `Program.fs`. Any errors?
        - Is `apu.Enabled` (NR52 bit 7) being set by the ROM? Log its value in `Apu.fs`'s `writeNr52`.
        - Are square wave channels `Enabled` (NRx2 bit 7, and NRx4 bit 7)? Log channel `Enabled` state in `Apu.fs`.
        - Is `apuState.SampleBuffer.Count` increasing? If not, `Apu.tick` isn't generating samples.
        - Are `SDL_QueueAudio` calls successful? Any error messages printed?
    - **Garbled/choppy sound?**
        - ⚠️ `What can go wrong:` This often indicates timing issues or buffer underruns/overruns. Is `AudioFrequency` matching what `Emulator.initEmulator` and `Apu.initApu` expect?
        - Is `CpuCyclesPerSample` calculated correctly?
        - Is the amount of data (in bytes) passed to `SDL_QueueAudio` correct? (It should be `samples * channels * sizeof<int16>`).
        - Are you calling `Emulator.step` enough times per frame to generate enough samples to keep the audio buffer full? Use `SDL_GetQueuedAudioSize` to monitor the buffer level. If it's consistently low, your emulator isn't feeding enough audio fast enough.
    - **Incorrect pitch?**
        - Double-check the `Frequency` calculation from NR13/NR14 registers (`131072 / (2048 - frequency)`).
        - Verify the `Timer` reload value (`(2048 - channel.Frequency) * 4`).
        - Ensure `WavePosition` is advancing correctly.

Production Considerations

- **Performance:** Audio generation is a constant, real-time task. Our `Apu.tick` function must be extremely lightweight. The

`generateSquareWaveSample` is simple enough for now, but more complex channels (wave, noise) and accurate frame sequencer logic will add overhead. Profile your `Apu.tick` function if you encounter performance bottlenecks.

- **Synchronization:** The most critical aspect. The `CpuCyclesPerSample` calculation and `PendingCpuCycles` accumulator are vital for smoothly bridging the CPU's irregular cycle counts with the fixed audio sample rate. Incorrect synchronization leads to crackling, popping, or incorrect pitch. Aim for cycle-accurate APU updates to match the Game Boy's behavior.
- **Buffer Management:** The `SampleBuffer` and `SDL_QueueAudio` implicitly manage buffers. Ensuring a steady stream of samples without underruns (starving the audio device) or overruns (too much data, leading to latency) is key. The `AudioBufferSize` for `SDL_OpenAudioDevice` and the `maxQueueBytes` logic are important here. Too small a buffer can lead to underruns, too large can lead to noticeable audio latency.
- **Accuracy vs. Simplicity:** We've made significant simplifications (no full frame sequencer, basic mixing). A production-quality emulator would require cycle-accurate implementation of all APU components, including sweep, envelope, length counters, and complex mixing logic. This iterative approach allows us to get basic functionality working before tackling the more complex, nuanced behaviors.

Common Issues & Solutions

1. **Issue:** No sound output at all.

- **Solution:** Verify SDL initialization for audio, check `SDL_OpenAudioDevice` return value. Ensure `APU.Enabled` (NR52 bit 7) is set to 1 by the game, and individual channel enable bits (NRx2 bit 7, NRx4 bit 7) are also set. Log the `SampleBuffer.Count` in `ApuState` to confirm samples are being generated. Check if `SDL_QueueAudio` is reporting errors. 2. **Issue:** Sound is choppy, distorted, or has clicks/pops.
- **Solution:** This is almost always a timing or buffering issue. Ensure your `AudioFrequency` and `AudioBufferSize` are reasonable (e.g., 44100 Hz, 2048 samples). Verify that `SDL_QueueAudio` is being called consistently and that the amount of data queued is appropriate for the audio device's buffer. If `SDL_GetQueuedAudioSize` reports too little data, you're underrunning. Increase your `PendingCpuCycles` threshold before queuing, or process more CPU cycles per audio tick. Ensure `float` to `int16` conversion handles

clipping correctly (`* 32767.0` is good for full range). 3. **Issue:** Pitch is off or sounds static.

- **Solution:** Check your `Frequency` and `Timer` calculations in `Apu.fs`. Consult Pan Docs for the exact formulas. A common mistake is incorrect division or multiplication factors. If sounds are static (no volume changes, no pitch slides), it means the length counter, envelope, and sweep units are not being emulated correctly (which is expected with our current basic implementation, but good to identify). This indicates a need for more advanced APU frame sequencer logic.

🧠 Check Your Understanding

- What are the primary challenges when synchronizing the Game Boy's APU with its CPU clock for accurate sound output?
- Explain the purpose of the `NRx4` register's trigger bit (bit 7) and why it's crucial for sound programming.
- How does the functional approach in F# (using immutable records and functions returning new state) help manage the complexity of APU state updates?

⚡ Mini Task

- Add logging to your `Apu.fs` `writeByte` function to print the register address and value being written. Observe these logs when running a game like Tetris to see how the APU registers are manipulated by the game. Focus on `0xFF14` and `0xFF19` to see trigger events.

🚀 Scenario

You're running a Game Boy ROM in your emulator, and the music sounds correct in pitch, but it never fades out or changes volume dynamically. It just plays at a constant loudness until the song changes or the channel is explicitly turned off. What specific components of the Game Boy APU are most likely missing or simplified in your current emulator implementation? How would you begin to diagnose and address this in the code?

📌 TL;DR

- The Game Boy APU has four sound channels; we started with two square wave channels.
- APU state is modeled with F# records, and updated via CPU writes to I/O registers, with state propagated through `Cpu.executeInstruction` and `Mmu.writeByte`.
- `SDL_QueueAudio` is used for direct, streaming output of generated audio samples, preferred for its control over raw sample data.
- Synchronization between CPU cycles and audio sample rate is critical for smooth sound.

🧠 Core Flow

1. CPU executes instructions, potentially writing to APU registers, returning updated `CpuState`, `MmuState`, and `ApuState`.
2. MMU routes register writes to the `Apu.writeByte` function, updating `ApuState`.
3. `Emulator.step` calls `Apu.tick`, advancing APU's internal frequency timers and accumulating audio samples based on consumed CPU cycles.
4. Accumulated samples are converted to `int16` and queued to the SDL audio device for playback, managing buffer levels to prevent glitches.

🚀 Key Takeaway

Emulating sound requires not only understanding hardware registers but also mastering the delicate balance of real-time synchronization and buffering to translate discrete, cycle-accurate updates into a continuous, smooth audio stream. This initial implementation provides basic sound, but true fidelity demands deeper emulation of the APU's dynamic components.

References

- F# Language Reference: https://learn.microsoft.com/en-us/dotnet/fsharp/
- .NET SDK Download: https://dotnet.microsoft.com/download
- Pan Docs - Sound Controller: https://gbdev.io/pandocs/Sound_Controller.html
- SDL Documentation: https://wiki.libsdl.org/

- SDL2-CS (VelcroPhysics.SDL.Net) GitHub: https://github.com/flibitijibibo/SDL2-CS

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 12

Synchronization, Debugging, and Verifying with Test ROMs

Introduction

Building a Game Boy emulator is a complex dance of interacting hardware components. The CPU, Picture Processing Unit (PPU), and Audio Processing Unit (APU) all operate at different speeds and rely on precise timing to function correctly. In this chapter, we'll tackle the critical challenge of **synchronization**, ensuring these components work together harmoniously.

Beyond just making things run, we need to know why they run or, more importantly, why they fail. This is where **debugging** becomes indispensable. We'll implement practical debugging tools—from logging to conditional breakpoints—to help us peer into the emulator's internal state.

Finally, to truly validate our work, we'll introduce **test ROMs**. These specially designed cartridges expose specific hardware behaviors, allowing us to systematically verify the accuracy of our emulation. By the end of this chapter, your emulator will not only be more stable but also equipped with the tools to diagnose and fix even the most subtle timing or logic errors, enabling it to pass initial verification tests.

Planning & Design: Orchestrating Hardware Components

The Game Boy's hardware components operate asynchronously but are bound by a shared clock. The CPU executes instructions, and each instruction takes a certain number of clock cycles. The PPU constantly renders pixels based on these cycles, and the APU generates sound. The challenge is to advance each component accurately based on the CPU's cycle consumption.

Synchronization Strategy

The core idea is to establish a "master clock" or cycle counter. The CPU will be the primary driver, reporting the cycles it consumes for each instruction. The main emulator loop will then distribute these cycles to the PPU and APU.

- **CPU as the Driver:** The `Cpu.executeInstruction` function will return the number of machine cycles (4 CPU clock cycles per machine cycle) it took to complete.
- **PPU Update:** The PPU needs to be updated with the accumulated cycles. It will then internally track its own timing, drawing pixels, advancing scanlines, and triggering interrupts (like VBlank) at precise cycle counts.
- **APU Update:** Similarly, the APU will receive cycle updates and manage its internal timers to generate sound.
- **Frame-based Processing:** We'll aim to process a full frame's worth of cycles (approximately 70224 cycles for a Game Boy frame) in each main loop iteration, then render the completed frame.

Debugging Architecture

Debugging an emulator is like being a detective in a complex system. We need tools to observe the system's internal state without altering its behavior.

- **Conditional Logging:** A central `Debug` module will allow us to print messages, register states, memory accesses, and opcode information. We'll make it configurable to enable/disable logging and redirect output to a file for detailed analysis.
- **Memory Viewers:** While not a full GUI debugger, we can implement functions to dump regions of memory (e.g., VRAM, I/O registers) to the console or log file at specific points.
- **Breakpoints (Conceptual):** For more advanced debugging, one could extend the `Debug` module to allow for programmatic breakpoints (e.g., "break when PC reaches 0x1234" or "break on write to 0xFF40"). For now, we'll rely on logging and manual inspection.

Verification with Test ROMs

Test ROMs are crucial. They are small programs designed to test specific aspects of the Game Boy hardware.

- **Blargg's Test ROMs:** This suite is the de-facto standard for Game Boy emulator development. They cover CPU instructions, memory access, PPU timing, and more.

- **Success Criteria:** Many test ROMs display a "Passed" message on the screen (or in the serial output, which we won't emulate yet) upon successful completion. Visual test ROMs require careful comparison against reference screenshots.

Step-by-Step Implementation

Let's integrate these concepts into our F# emulator.

1. Centralizing Emulator State and Timing

First, we need to ensure our main emulator state record holds all necessary components and timing information. We'll also refine our main update loop.

Open `Emulator.fs` (or the file containing your `EmulatorState` and main loop) and update it:

```
// src/Gameboy/Emulator.fs

module Emulator

open Cpu
open Mmu
open Ppu
open Apu
open Input
open Interrupts
open System // For DateTime in Debug module

// ⚡ Quick Note: These are approximate values.
// Game Boy CPU clock: 4.194304 MHz
// Cycles per frame (approx. 60 FPS): 4194304 / 60 = 69905 cycles.
// Pan Docs states 70224 cycles per frame (456 cycles/scanline * 154 scanlines)
let private CYCLES_PER_FRAME = 70224

type EmulatorState = {
    Cpu: CpuState
    Mmu: MmuState
    Ppu: PpuState
    Apu: ApuState
    Input: InputState
    FrameBuffer: byte array // The current frame's pixel data
    CyclesThisFrame: int // Total cycles accumulated in the current frame
    FrameCounter: int
    IsRunning: bool
}

// 📌 Key Idea: The 'init' function now sets up a complete emulator state.
let init (romData: byte array) =
    let mmuState = Mmu.init romData
    let cpuState = Cpu.init
    let ppuState = Ppu.init
    let apuState = Apu.init
    let inputState = Input.init

    { Cpu = cpuState
      Mmu = mmuState
      Ppu = ppuState
      Apu = apuState
      Input = inputState
      FrameBuffer = Array.zeroCreate (Ppu.SCREEN_WIDTH * Ppu.SCREEN_HEIGHT *
4) // 4 bytes per pixel (RGBA)
      CyclesThisFrame = 0
      FrameCounter = 0
      IsRunning = true }

// 🧠 Important: This is the heart of the emulator's execution loop.
// It orchestrates how CPU, PPU, and APU advance together.
let update (state: EmulatorState) : EmulatorState * byte array option =
    if not state.IsRunning then
        state, None // If not running, return current state and no new frame

    let rec loop (currentState: EmulatorState) (accumulatedCycles: int) =
        if accumulatedCycles >= CYCLES_PER_FRAME then
            // Frame complete, return the frame buffer and reset for next frame
            let nextState =
                { currentState with
                    CyclesThisFrame = 0
```

```
                    FrameCounter = currentState.FrameCounter + 1
                    FrameBuffer = Ppu.getFrameBuffer currentState.Ppu //
Capture current frame
                }
            nextState, Some nextState.FrameBuffer // Return state for next
frame, and the completed frame
        else
            // Execute CPU instruction
            // ⚡ Real-world insight: The CPU is the "master clock" in this
simplified model.
            let (cpuState, mmuState, cyclesConsumed) = Cpu.executeInstruction c
urrentState.Cpu currentState.Mmu

            // Update PPU based on cycles consumed by CPU
            // The PPU's internal logic handles drawing pixels, scanlines, and
VBlank/HBlank timing.
            let ppuState = Ppu.update currentState.Ppu cyclesConsumed mmuState

            // Update APU based on cycles consumed by CPU (placeholder for now)
            let apuState = Apu.update currentState.Apu cyclesConsumed

            // Handle interrupts based on new CPU and PPU state
            let (cpuStateWithInterrupts, mmuStateWithInterrupts) =
                Interrupts.checkAndHandle cpuState mmuState ppuState

            let newState =
                { currentState with
                    Cpu = cpuStateWithInterrupts
                    Mmu = mmuStateWithInterrupts
                    Ppu = ppuState
                    Apu = apuState
                    CyclesThisFrame = accumulatedCycles + cyclesConsumed
                }
            loop newState (accumulatedCycles + cyclesConsumed)

    loop state state.CyclesThisFrame // Start or continue the loop for the
current frame
```

Explanation:

- `CYCLES_PER_FRAME`: Defines the target number of CPU cycles to simulate for one full frame. This is a critical constant for smooth emulation.
- `EmulatorState`: Now includes `FrameBuffer` to store the raw pixel data for the current frame and `CyclesThisFrame` to track progress within the current frame. `IsRunning` is added for control.
- `init`: Initializes all component states.
- `update`: This is the main emulator loop.
    - It recursively calls `loop` until `accumulatedCycles` reaches `CYCLES_PER_FRAME`.
    - `Cpu.executeInstruction` is called, and its returned `cyclesConsumed` is used to advance the `Ppu` and `Apu`.

- `Interrupts.checkAndHandle` is called after each instruction to see if any interrupts (like VBlank from the PPU) need to be serviced by the CPU.
- When `CYCLES_PER_FRAME` is reached, the `FrameBuffer` from the PPU is captured, and the `CyclesThisFrame` counter is reset.

2. Updating PPU and CPU to Support Cycle Counting

Now, let's ensure the PPU and CPU components correctly integrate with this new timing model.

PPU Cycle Update

Open `Ppu.fs` and modify its `update` function. The PPU needs to manage its internal state based on the cycles it receives.

```
// src/Gameboy/Ppu.fs

module Ppu

open Mmu

// ... existing types and constants ...

type PpuState = {
    // ... existing fields ...
    FrameBuffer: byte array // Raw pixel data for the current frame (RGBA)
    Scanline: int // Current scanline (LY register)
    ScanlineCycle: int // Cycles within the current scanline
    Mode: PpuMode // Current PPU mode (HBlank, VBlank, OAM, VRAM)
    VBlankInterruptRequested: bool
    LycInterruptRequested: bool
}

let init =
    {   // ... existing init values ...
        FrameBuffer = Array.zeroCreate (SCREEN_WIDTH * SCREEN_HEIGHT * 4)
        Scanline = 0
        ScanlineCycle = 0
        Mode = OamScan
        VBlankInterruptRequested = false
        LycInterruptRequested = false
    }

// 🧠 Important: The PPU update function is responsible for advancing the
PPU's internal state
// and drawing pixels based on the cycles provided by the CPU.
let update (state: PpuState) (cpuCycles: int) (mmu: MmuState) : PpuState =
    let mutable currentState = state
    let mutable currentMmu = mmu // PPU needs to read from MMU

    let rec processCycles (remainingCycles: int) =
        if remainingCycles <= 0 then
            currentState // All cycles processed
        else
            let cyclesToAdvance = min remainingCycles (PPU_CYCLES_PER_SCANLINE
- currentState.ScanlineCycle)

            // Advance scanline cycles
            currentState <- { currentState with ScanlineCycle = currentState.Sc
anlineCycle + cyclesToAdvance }
            let mutable nextRemainingCycles = remainingCycles - cyclesToAdvance

            // ⚡ Real-world insight: PPU modes change based on scanline cycle
counts.
            // This is critical for accurate timing and rendering.
            match currentState.Mode with
            | OamScan -> // Mode 2: Searching OAM (80 cycles)
                if currentState.ScanlineCycle >= OAM_SCAN_CYCLES then
                    currentState <- { currentState with Mode = VramScan }
            | VramScan -> // Mode 3: Drawing pixels (172 cycles, varies)
                if currentState.ScanlineCycle >= OAM_SCAN_CYCLES + VRAM_SCAN_CY
CLES then // Approx 80 + 172 = 252 cycles
                    // 📌 Key Idea: Render the current scanline when VRAM scan
is complete.
                    let tileData = Mmu.readVramTileData currentMmu
                    let spriteData = Mmu.readOamSpriteData currentMmu
```

```
                    currentState <- Renderer.renderScanline currentState curren
tState.Scanline tileData spriteData currentMmu.Memory.IoRegisters
                    currentState <- { currentState with Mode = HBlank }
            | HBlank -> // Mode 0: Horizontal Blank (204 cycles, varies)
                if currentState.ScanlineCycle >= PPU_CYCLES_PER_SCANLINE then
                    // Scanline complete, move to next scanline
                    currentState <- { currentState with Scanline =
currentState.Scanline + 1; ScanlineCycle = 0 }

                    // Check for LYC=LY coincidence interrupt
                    let lyc = currentMmu.Memory.IoRegisters.[0xFF45 - 0xFF00] /
/ LYC register
                    if currentState.Scanline = int lyc then
                        currentState <- { currentState with LycInterruptRequest
ed = true }

                    if currentState.Scanline >= SCREEN_HEIGHT then // 144
scanlines
                        // 🧠 Important: Enter VBlank after rendering all
visible scanlines.
                        // This is when the VBlank interrupt (bit 0 of IF
register) is triggered.
                        currentState <- { currentState with Mode = VBlank; VBla
nkInterruptRequested = true }
                    else
                        currentState <- { currentState with Mode =
OamScan } // Back to OAM scan for next scanline
            | VBlank -> // Mode 1: Vertical Blank (10 scanlines, 4560 cycles)
                if currentState.ScanlineCycle >= PPU_CYCLES_PER_SCANLINE then
                    currentState <- { currentState with Scanline =
currentState.Scanline + 1; ScanlineCycle = 0 }
                    if currentState.Scanline >= TOTAL_SCANLINES then // 144 +
10 = 154 scanlines
                        // End of VBlank period, reset for next frame
                        currentState <- { currentState with Scanline = 0; Mode
= OamScan }

            processCycles nextRemainingCycles // Continue processing remaining
cycles

    processCycles cpuCycles
```

Explanation:

- `PpuState` now directly stores `FrameBuffer`, `Scanline`, `ScanlineCycle`, `Mode`, and flags for `VBlankInterruptRequested` and `LycInterruptRequested`.
- The `update` function takes `cpuCycles` and `mmu` (to read VRAM/OAM/I/O registers).
- It iteratively processes cycles, updating `ScanlineCycle` and transitioning `Mode` (OAM Scan, VRAM Scan, HBlank, VBlank) based on cycle counts, as described in the Pan Docs.
- `Renderer.renderScanline` (which you would have implemented in a previous chapter) is called when the VRAM scan completes for a scanline.

- **Interrupts:** `VBlankInterruptRequested` and `LycInterruptRequested` flags are set when their respective conditions are met. These will be picked up by the `Interrupts` module.
- The PPU now internally manages its timing entirely based on the `cpuCycles` it receives.

CPU Instruction Cycle Return

Modify `Cpu.fs` so that `executeInstruction` returns the number of cycles consumed. Each opcode has a specific cycle count. You'll need to consult the Game Boy CPU manual (SM83) or Pan Docs for these values.

```
// src/Gameboy/Cpu.fs

module Cpu

open Mmu
open Registers

// ... existing types and functions ...

// 🧠 Important: Each instruction consumes a specific number of CPU cycles.
// This is crucial for accurate synchronization with other components.
let executeInstruction (state: CpuState) (mmu: MmuState) : CpuState * MmuState
* int =
    let pc = state.PC
    let opcode = Mmu.readByte mmu pc // Fetch opcode
    let nextPc = pc + 1

    let (nextState, nextMmu, cycles) = // All instructions should return cycles
        match opcode with
        // Example opcodes (you'll have many more):
        | 0x00uy -> // NOP
            { state with PC = nextPc }, mmu, 4 // NOP takes 4 cycles
        | 0x01uy -> // LD BC, d16
            let value = Mmu.readWord mmu nextPc
            let nextNextPc = nextPc + 2
            let nextRegs = { state.Registers with BC = value }
            { state with PC = nextNextPc; Registers = nextRegs }, mmu,
12 // LD BC,d16 takes 12 cycles
        | 0xAFuy -> // XOR A
            let nextRegs = { state.Registers with A = 0x00uy; Flags = { Z = tru
e; N = false; H = false; C = false } }
            { state with PC = nextPc; Registers = nextRegs }, mmu, 4
        // ... many more opcodes ...
        | _ ->
            // ⚠️ What can go wrong: Unimplemented opcodes will cause
unexpected behavior or crashes.
            // This is a common point of failure for emulator development.
            Debug.log (sprintf "Unimplemented opcode: 0x%02X at PC: 0x%04X" opc
ode pc)
            failwithf "Unimplemented opcode: 0x%02X at PC: 0x%04X" opcode pc

    // ⚡ Real-world insight: Interrupt handling is often checked *after* an
instruction,
    // but the actual jump to the interrupt handler happens before the *next*
instruction.
    // Our Interrupts module handles the actual jumping.
    nextState, nextMmu, cycles
```

Explanation:

- The `executeInstruction` function's return type is now `CpuState * MmuState * int`, where the `int` is the number of CPU cycles consumed by the instruction.
- You must populate the `cycles` value for every opcode you implement. This will involve careful reference to the Game Boy CPU documentation (e.g., Pan Docs, SM83 manual).

3. Implementing a Debugging Module

Create a new file `Debug.fs` in your `src/Gameboy` directory.

```
// src/Gameboy/Debug.fs

module Debug

open System
open System.IO

// 🧠 Important: Conditional compilation allows us to remove debugging code
// from release builds for performance.
#if DEBUG
let mutable private enableLogging = true
let mutable private logToFile = false
let mutable private logFilePath = "emulator_log.txt"

let init (enabled: bool) (toFile: bool) (path: string) =
    enableLogging <- enabled
    logToFile <- toFile
    logFilePath <- path
    if logToFile then
        // Clear log file on startup
        try
            File.WriteAllText(logFilePath, "")
        with ex ->
            Console.WriteLine($"Error clearing log file: {ex.Message}")

let log (message: string) =
    if enableLogging then
        let formattedMessage = sprintf "[%s] %s" (DateTime.Now.ToString("HH:mm:
ss.fff")) message
        if logToFile then
            try
                File.AppendAllText(logFilePath, formattedMessage +
Environment.NewLine)
            with ex ->
                Console.WriteLine($"Error appending to log file: {ex.Message}")
        else
            Console.WriteLine(formattedMessage)

// 🔥 Optimization / Pro tip: Use this for verbose logging that can be
toggled.
let logCpuState (state: Cpu.CpuState) =
    if enableLogging then
        log (sprintf "CPU: PC=0x%04X AF=0x%04X BC=0x%04X DE=0x%04X HL=0x%04X
SP=0x%04X Flags:%A"
            state.PC
            state.Registers.AF
            state.Registers.BC
            state.Registers.DE
            state.Registers.HL
            state.Registers.SP
            state.Registers.Flags)

let logMmuRead (addr: int) (value: byte) =
    if enableLogging then
        log (sprintf "MMU Read: 0x%04X -> 0x%02X" addr value)

let logMmuWrite (addr: int) (value: byte) =
    if enableLogging then
        log (sprintf "MMU Write: 0x%04X <- 0x%02X" addr value)

#else // If not in DEBUG configuration
```

```
// Provide no-op implementations for release builds
let init (_: bool) (_: bool) (_: string) = ()
let log (_: string) = ()
let logCpuState (_: Cpu.CpuState) = ()
let logMmuRead (_: int) (_: byte) = ()
let logMmuWrite (_: int) (_: byte) = ()
#endif
```

Explanation:

- **Conditional Compilation (`#if DEBUG`):** This is a powerful feature. When you compile your project in `Debug` configuration, the logging code is included. When you compile in `Release` configuration, all the code within the `#if DEBUG` block is completely removed, ensuring zero performance overhead for logging in production builds.
- `enableLogging` and `logToFile`: Mutable flags to control logging behavior at runtime.
- `init`: Sets up the logger, optionally clearing the log file.
- `log`: The core logging function.
- `logCpuState`, `logMmuRead`, `logMmuWrite`: Helper functions for common debugging scenarios.

Integrating Debug Logging

Now, integrate `Debug.log` calls into your CPU and MMU logic.

In `Cpu.fs`:

```
// src/Gameboy/Cpu.fs

module Cpu

open Mmu
open Registers
open Debug // Add this line

// ... existing code ...

let executeInstruction (state: CpuState) (mmu: MmuState) : CpuState * MmuState
* int =
    let pc = state.PC
    let opcode = Mmu.readByte mmu pc
    let nextPc = pc + 1

    Debug.logCpuState state // Log CPU state before instruction
    Debug.log (sprintf "Executing PC=0x%04X Opcode=0x%02X" pc opcode)

    let (nextState, nextMmu, cycles) =
        match opcode with
        // ... your opcode implementations ...
        | _ ->
            Debug.log (sprintf "Unimplemented opcode: 0x%02X at PC: 0x%04X" opc
ode pc)
            failwithf "Unimplemented opcode: 0x%02X at PC: 0x%04X" opcode pc

    nextState, nextMmu, cycles
```

In `Mmu.fs`:

```
// src/Gameboy/Mmu.fs

module Mmu

open Debug // Add this line

// ... existing code ...

let readByte (state: MmuState) (addr: int) =
    let value =
        // ... existing read logic ...
    Debug.logMmuRead addr value
    value

let writeByte (state: MmuState) (addr: int) (value: byte) =
    // ... existing write logic ...
    Debug.logMmuWrite addr value
    nextState
```

Enabling Debugging in `Program.fs` (or your UI layer):

```
// src/GameboyEmulator/Program.fs (or wherever your main application entry
point is)

[<EntryPoint>]
let main argv =
    Debug.init true true "emulator_trace.log" // Enable logging, write to file

    // ... your existing UI setup and emulator loop ...
    0
```

4. Test ROM Loading

Loading a test ROM is no different from loading a regular Game Boy ROM. Ensure your `loadRom` function (from an earlier chapter) is robust.

You can find Blargg's test ROMs at various emulator development resources, often bundled with other projects or directly from his GitHub. A common set includes `cpu_instrs.gb`, `instr_timing.gb`, `mem_timing.gb`, `ppu_timing.gb`.

Example:

When running your emulator, instead of loading a game ROM, load `cpu_instrs.gb`. If your CPU emulation is correct, you should see a series of messages on the screen, eventually ending with a "Passed" message.

Testing & Verification

1. **Run `cpu_instrs.gb`:**
    - Download `cpu_instrs.gb` (e.g., from Blargg's GB tests repo).
    - Load it into your emulator.

- **Expected Behavior:** The screen should go through several tests, displaying "Passed" or "Failed" for each. If all CPU instructions are correctly emulated, you should eventually see a final "Passed" message on the screen.
- **Verification:** If it fails or hangs, inspect your `emulator_trace.log` file. Look for the last executed opcode before the failure, or memory accesses that seem incorrect.

1. **Run `dmg_acid2.gb` (PPU Test):**
    - Download `dmg_acid2.gb` (e.g., from GBDEV Wiki).
    - Load it into your emulator.

- **Expected Behavior:** A smiling face should appear, with specific pixel patterns. This ROM is very sensitive to PPU timing and rendering accuracy.

- **Verification:** Compare your emulator's output against reference screenshots. If it's incorrect, use your PPU mode and scanline cycle logging (if you add it) to pinpoint rendering issues.

Production Considerations

- **Performance:** The conditional compilation (`#if DEBUG`) for the `Debug` module is key. In a release build, all logging calls compile away, ensuring maximum performance. For actual performance profiling, use .NET's built-in profilers (e.g., Visual Studio Profiler, `dotnet-trace`). The PPU rendering loop and CPU instruction execution are the most critical hotspots.
- **Maintainability:** Keeping synchronization logic centralized in the main `Emulator.update` loop and distinct within each component (PPU managing its own internal cycles) improves clarity. A well-defined `Debug` module with clear logging categories aids in long-term maintenance.
- **Error Handling:** Unimplemented opcodes should `failwith` in debug builds, but in a production emulator, you might want a more graceful error message or a "halt" state.
- **Cross-Platform:** The core F# logic is cross-platform. Your UI layer (e.g., SDL.NET) handles the platform-specific rendering.

Common Issues & Solutions

- **Incorrect Timing/Synchronization:**
- **Problem:** The most common issue. Game Boy screens might flicker, graphics might be corrupted, or test ROMs might hang. This usually means the PPU isn't advancing correctly relative to the CPU, or interrupts are triggered at the wrong time.
- **Solution:** 1. **Verify CPU Cycle Counts:** Double-check every single opcode's cycle count against the Pan Docs. Even one incorrect count can throw off the entire system. 2. **PPU Mode Transitions:** Carefully review the PPU's `update` logic. Ensure it transitions between OAM Scan, VRAM Scan, HBlank, and VBlank at the exact cycle counts (e.g., 80 cycles for OAM, 252 for VRAM+OAM, 456 for a full scanline). 3. **Interrupt Flags:** Confirm that the VBlank interrupt (bit 0 of IF register) is set precisely when the PPU enters VBlank mode, and reset when the CPU acknowledges it. The LYC=LY interrupt also needs precise timing.

- **Debugging:** Use the `Debug` module to log `Ppu.Scanline`, `Ppu.ScanlineCycle`, and `Ppu.Mode` at every PPU update. Compare this trace against the expected PPU behavior described in Pan Docs.
- **Test ROMs Hang or Fail Immediately:**
- **Problem:** Often indicates a fundamental error in CPU instruction implementation (e.g., stack operations, jumps, flag updates) or MMU address mapping.
- **Solution:** 1. **Start Simple:** Begin with the simplest Blargg's test ROMs, like `01-special.gb` or `02-interrupts.gb`, which test basic CPU features. 2. **Instruction Tracing:** Enable verbose CPU logging. Trace the execution of the ROM instruction by instruction. When it hangs, the last few instructions in the log will be your culprits. Compare the register states (PC, SP, AF, BC, DE, HL) with a known good emulator trace or step through manually with the Pan Docs. 3. **MMU Mapping:** Ensure your MMU correctly maps ROM, VRAM, WRAM, OAM, and I/O registers to their precise addresses. Test ROMs often try to write to specific I/O registers to signal success or failure.
- **Visual Glitches on PPU Test ROMs:**
- **Problem:** Sprites might be misplaced, background scrolling might be off, or colors might be wrong.
- **Solution:** 1. **LCDC Register:** The `0xFF40` (LCDC) register controls most PPU features. Log its value and ensure your PPU interprets all its bits correctly (e.g., background display enable, window enable, sprite enable, tile map selection). 2. **Scroll Registers (SCX, SCY):** Ensure `0xFF42` (SCY) and `0xFF43` (SCX) are read and applied correctly during background rendering. 3. **Palette Registers:** Verify that `0xFF47` (BGP), `0xFF48` (OBP0), `0xFF49` (OBP1) are correctly used to map tile/sprite pixel values to actual colors.
- **Debugging:** Create specific logging for PPU register reads when rendering a scanline. Dump VRAM tile data or OAM sprite data at critical moments to see if the raw data is what you expect.

Summary & Next Step

In this chapter, we've brought our Game Boy emulator closer to reality by tackling synchronization, a cornerstone of accurate emulation. We established a master clock model, allowing the CPU, PPU, and APU to advance in harmony. We also built a robust debugging framework, leveraging F#'s capabilities and conditional compilation, to peer into the emulator's inner workings. Finally, we introduced the

critical practice of using test ROMs to verify our emulation against the actual hardware behavior.

Your emulator should now be capable of loading and running simple Game Boy ROMs, including verification test suites like Blargg's CPU tests. This marks a significant milestone in ensuring the foundational accuracy of your Game Boy emulation.

The next step would be to refine the APU implementation, add more sophisticated Memory Bank Controllers (MBCs) for larger and more complex ROMs, and further polish the user interface.

🧠 Check Your Understanding

- What is the primary role of the `cyclesConsumed` value returned by the CPU in the main emulator loop?
- Why is conditional compilation (`#if DEBUG`) important for the `Debug` module in an emulator project?
- Name two common Game Boy test ROMs and explain what aspects of the emulator they help verify.

⚡ Mini Task

Modify your PPU's `update` function to include a `Debug.log` call whenever the PPU's `Mode` changes (e.g., from `OamScan` to `VramScan`). Observe the output when running a simple ROM.

🚀 Scenario

Your emulator runs `cpu_instrs.gb` and passes most tests, but consistently fails on a specific `LD (HL+), A` instruction test. You've confirmed the opcode's cycle count is correct. What specific debugging steps would you take, beyond just logging the CPU state, to pinpoint the exact issue? Consider both CPU and MMU interactions.

📌 TL;DR

- **Synchronization** is achieved by having the CPU report consumed cycles, which then drive PPU and APU updates.

- **Debugging** involves conditional logging of CPU state, memory access, and PPU events, crucial for diagnosing issues.
- **Test ROMs**, particularly Blargg's suite, are essential for verifying emulator accuracy against real hardware behavior.

🧠 Core Flow

1. CPU executes an instruction and returns the number of cycles it consumed.
2. Main emulator loop advances PPU and APU by the same number of cycles.
3. PPU updates its internal state (scanline, mode, pixels) and triggers interrupts based on received cycles.
4. Interrupts are checked and handled by the CPU before the next instruction.
5. Debugging logs provide insight into CPU, MMU, and PPU states for verification.

🚀 Key Takeaway

Accurate synchronization and robust debugging tools are paramount for emulator development; without them, even small timing discrepancies can lead to significant emulation failures that are nearly impossible to diagnose.

References

- F# Language Reference: https://learn.microsoft.com/en-us/dotnet/fsharp/
- .NET Documentation: https://learn.microsoft.com/en-us/dotnet/
- SDL Documentation: https://wiki.libsdl.org/
- Pan Docs (Game Boy Technical Manual): https://gbdev.io/pandocs/
- Blargg's Game Boy Test ROMs: https://github.com/retrio/gb-test-roms

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 13

Building a Game Boy Emulator with F#

Building a Game Boy emulator from the ground up is a deeply rewarding project that takes you into the heart of computer architecture and low-level system design. This guide will walk you through constructing a functional Game Boy emulator using F#, focusing on a modular, functional approach to replicate the original hardware's behavior.

Why Build an Emulator?

Emulators are more than just software; they are digital time capsules that preserve computing history and provide a unique window into how hardware and software truly interact. By building one, you'll gain an unparalleled understanding of:

- **CPU Architecture:** How instructions are fetched, decoded, and executed, and how registers and flags manage state.
- **Memory Management:** The intricate dance between ROM, RAM, and I/O registers, and how memory bank controllers extend addressable space.
- **Graphics Pipelines:** The process of rendering pixels from tile data, sprites, and background maps to form a visual display.
- **Timing and Synchronization:** The critical importance of precise timing between different hardware components to ensure correct operation.
- **Functional Programming in Systems:** How F#'s strong typing, immutability, and expressive power can be leveraged for complex, stateful systems while maintaining clarity and correctness.

This project is not just about writing code; it's about reverse-engineering a classic system and bringing it back to life, pixel by pixel, cycle by cycle.

What You'll Build

By the end of this guide, you will have developed a desktop application capable of loading and running simple Game Boy ROMs. Your emulator will feature:

- A cycle-accurate CPU core for the Game Boy's custom 8-bit processor (SM83).

- A Memory Management Unit (MMU) handling ROM, RAM, VRAM, OAM, and I/O registers.
- A Picture Processing Unit (PPU) for rendering graphics to a display window.
- Input handling to map keyboard presses to Game Boy buttons.
- Basic Audio Processing Unit (APU) emulation for core sound channels.
- Support for common Memory Bank Controllers (MBCs) to run a wider range of games.

We'll prioritize correctness and clarity, leveraging F#'s strengths to model hardware components as immutable data structures and state transitions as pure functions where possible.

Prerequisites

To get the most out of this guide, you should have:

- **F# Proficiency:** A solid understanding of F# syntax, types (records, discriminated unions), pattern matching, and functional programming concepts.
- **C# Background (Optional but helpful):** Familiarity with .NET ecosystem concepts.
- **Basic Computer Architecture Knowledge:** Understanding of CPUs, memory, and I/O.
- **A Desire to Learn:** This project involves diving deep into technical documentation and problem-solving.

Development Environment Setup

We'll use the .NET SDK for F# development and SDL.NET for cross-platform graphics.

1. .NET SDK and F# Tooling

The F# compiler and tools are included with the .NET SDK.

- **.NET SDK:** As of 2026-05-05, please check the official .NET website for the latest stable release. We recommend installing the latest stable version available for your operating system.
- **Verification:** After installation, open a terminal or command prompt and run: `bash dotnet --version` This should output the installed SDK version.

- **Code Editor:**
- **Visual Studio Code:** Recommended for its cross-platform support and excellent F# extension, Ionide.
- **Visual Studio (Windows):** Full-featured IDE with integrated F# support.

2. SDL.NET for Graphics

SDL.NET is a .NET binding for the popular Simple DirectMedia Layer (SDL) library, which provides cross-platform access to graphics, audio, and input devices.

- **SDL Development Libraries:** You will first need to install the native SDL2 development libraries for your operating system.
- **Windows:** Download the development libraries from the SDL website (e.g., `SDL2-devel-2.x.x-vc.zip` for MSVC). Extract them and ensure the `bin` directory (containing `SDL2.dll`) is in your system's PATH or copied to your project's output directory.
- **macOS:** Install via Homebrew: `brew install sdl2`
- **Linux (Debian/Ubuntu):** `sudo apt-get install libsdl2-dev`
- **SDL.NET NuGet Package:** We will add the `SDL.NET` NuGet package to our F# project. As of 2026-05-05, please check NuGet Gallery for the latest stable version of `SDL.NET`.
- **Verification:** We'll verify this in the first chapter by creating a basic window.

Architectural Approach

Our emulator will follow a modular design, treating each Game Boy hardware component as a distinct F# module or type.

- **Immutable State:** Where feasible, we'll represent hardware components (like CPU registers, PPU state) as immutable F# records. This enhances predictability and makes debugging easier.
- **Explicit State Transitions:** Functions will take an old state and return a new state, making state changes explicit and traceable.
- **Discriminated Unions for Instructions:** CPU opcodes and their parameters will be modeled using F# discriminated unions, allowing for powerful pattern matching in the CPU's execution loop.
- **Performance Considerations:** While F# promotes immutability, we'll strategically use mutable arrays for large memory blocks (like WRAM or

VRAM) where performance is critical, managing these mutable regions carefully within a functional context.

- **Separation of Concerns:** The CPU, MMU, PPU, and APU will interact through well-defined interfaces, minimizing coupling.

Production Awareness

While building an emulator is often seen as a hobby project, we'll approach it with a production mindset:

- **Testing:** Unit tests will be crucial for verifying individual CPU opcodes, memory operations, and PPU logic. We'll also use well-known Game Boy test ROMs (like Blargg's tests) for integration testing.
- **Maintainability:** Clear code, good documentation, and a modular structure will ensure the codebase remains understandable and extensible.
- **Performance:** Emulators are performance-sensitive applications. We'll discuss profiling and optimization techniques, especially for the CPU and PPU loops, to achieve acceptable frame rates.
- **Debugging:** We'll integrate basic logging and debugging capabilities to inspect the emulator's internal state, which is invaluable when tracking down subtle hardware emulation bugs.

Learning Path

This guide is structured into incremental milestones, allowing you to build and verify your emulator piece by piece.

Setting Up Your Emulator Development Environment

Configure the .NET SDK, F# tooling, SDL.NET, and gain an overview of the Game Boy's core architecture and essential technical documentation.

The CPU Core: Registers, Flags, and Basic Instructions

Model the Game Boy CPU's registers and flags using F# records and discriminated unions, then implement initial data manipulation opcodes like LD, INC, and DEC.

Memory Management Unit (MMU) and Basic Memory Access

Design the MMU structure, define the Game Boy's memory map, and implement fundamental byte read/write operations for Work RAM and High RAM.

Loading ROMs and Initial Boot Sequence

Implement cartridge loading from a file, parse ROM headers, and simulate the Game Boy's boot ROM to begin executing game code.

CPU Control Flow: Jumps, Calls, and Conditional Logic

Expand the CPU with instructions for program flow control, including Jumps (JP), Calls (CALL), Returns (RET), and their conditional variants.

Interrupts and the Main CPU Execution Loop

Implement the Game Boy's interrupt system, including enabling/disabling and handling, and establish the CPU's cycle-accurate main execution loop.

Picture Processing Unit (PPU) Part 1: VRAM and Background Rendering

Model the PPU's Video RAM and control registers, then implement the initial rendering pipeline for displaying static background tiles on screen.

Picture Processing Unit (PPU) Part 2: Sprites, Scrolling, and LCD Control

Enhance the PPU to render sprites, handle background and window scrolling, and manage the LCD control register for display modes.

Input Handling: Connecting Keyboard to Game Boy Buttons

Implement the Game Boy's input registers and map keyboard presses to simulate button inputs for player interaction within the emulator.

Advanced MMU: Memory Bank Controllers (MBCs)

Implement support for various Memory Bank Controllers (MBCs) to correctly handle memory switching for larger and more complex Game Boy cartridges.

Audio Processing Unit (APU) Basics: Square Wave Channels

Model the APU's basic registers and generate simple square wave audio output for the Game Boy's first two sound channels.

Synchronization, Debugging, and Verifying with Test ROMs

Integrate all components with proper timing synchronization, add basic debugging capabilities, and validate the emulator's accuracy using Blargg's test ROMs.

🧠 Check Your Understanding

- What are the primary benefits of using F# for an emulator project, especially concerning state management?

- Why is accurate timing and synchronization between CPU, PPU, and APU critical for an emulator's correctness?
- What are some key challenges you anticipate when emulating a system like the Game Boy, particularly regarding performance?

⚡ Mini Task

- Research the "Pan Docs" and the "Game Boy CPU Manual (SM83)". Understand their purpose and why they are indispensable resources for this project.

🚀 Scenario

You're debugging an issue where a game's graphics appear corrupted after a few seconds of play. Based on the architectural overview, which components would you primarily suspect, and what initial steps would you take to narrow down the problem? Consider the interaction between the CPU, MMU, and PPU.

References

- F# Language Reference: https://learn.microsoft.com/en-us/dotnet/fsharp/
- .NET Documentation: https://learn.microsoft.com/en-us/dotnet/
- SDL Documentation: https://wiki.libsdl.org/
- Pan Docs (Game Boy Technical Reference): https://gbdev.io/pandocs/
- Game Boy CPU Manual (SM83): https://gbdev.io/docs/cpu/sm83_cpu.pdf
- NuGet Gallery (SDL.NET): https://www.nuget.org/packages/SDL.NET/

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

📌 TL;DR

- Build a Game Boy emulator in F# to understand low-level system design.
- Focus on CPU, MMU, PPU, APU, and input, using F#'s functional strengths.
- Emphasize modularity, testing, performance, and clear state management.

🧠 Core Flow

1. Set up F# and SDL.NET development environment.
2. Implement CPU core, starting with registers and basic instructions.
3. Develop the MMU for memory access and ROM loading.
4. Integrate PPU for graphics rendering.
5. Add input, advanced MMU (MBCs), and basic APU.
6. Synchronize components and verify with test ROMs.

🚀 Key Takeaway

Emulating legacy hardware provides a deep, hands-on understanding of computer architecture, forcing you to confront and solve complex timing, state, and performance challenges that are invaluable for any system designer.