



UNIVERSITY OF
LEICESTER

CO3015 Computer Science Project Dissertation Software Emulator for the Nintendo Game Boy Color

B.Sc. in Computer Science

University of Leicester
Department of Informatics

Sean Dewar

Submitted: May, 2018

DECLARATION

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s).

Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s).

I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Sean Dewar

Signed: 

Date: 03/05/2018

Contents

1 Abstract	6
2 Introduction	7
2.1 Brief Description of the Game Boy Color.....	7
2.2 Terminology	8
2.3 Aims and Objectives.....	8
3 Hardware Background	10
3.1 Memory and System Start-up	11
3.1.1 System Memory	11
3.1.2 Program Cartridges.....	12
3.1.3 Memory Mapping.....	13
3.1.4 Boot Procedure.....	14
3.2 CPU	15
3.2.1 CPU Registers	15
3.2.2 ALU	16
3.2.3 Instructions	16
3.2.4 Program Stack	19
3.2.5 Clock	20
3.2.6 Interrupts.....	21
3.3 System Timer	23
3.4 Joypad	24
3.5 LCD Video Controller (PPU).....	26
3.5.1 General Operation.....	26
3.5.2 Tiles.....	28
3.5.3 Layers.....	30
3.5.4 DMA.....	32
3.6 Audio Controller (APU)	35
3.6.1 Channel Overview	35
3.6.2 Volume Envelope and Frequency Sweep	36
3.6.3 DAC, Frequency Timers, Length Counters and Triggers.....	37

3.6.4 Frame Sequencer	38
4 Software Overview	39
4.1 Main GUI Mode	39
4.1.1 Loading Programs	40
4.1.2 Battery-Packed RAM Snapshots	41
4.1.3 Emulation Controls	42
4.1.4 Miscellaneous Controls	42
4.2 Command-Line Debugger Mode	43
5 Implementation	45
5.1 Emulation Back-End	46
5.1.1 Class Diagram	46
5.1.2 Emulation	47
5.1.3 Memory	48
5.1.4 CPU	49
5.1.5 Joypad	52
5.1.6 System Timer	53
5.1.7 PPU	54
5.1.8 APU	55
5.2 GUI Front-End	57
5.2.1 Class Diagram	57
5.2.2 General Implementation	58
6 Testing	60
6.1 Critical Hardware Emulation Tests	60
6.2 Miscellaneous Hardware Emulation Tests	61
6.3 Manual Program Compatibility Tests	62
7 Critical Appraisal	63
7.1 Critical Analysis	63
7.1.1 Key Objectives	63
7.1.2 Optional Objectives	64
7.2 Social and Academic Context	65

7.3 Personal Development	66
8 Conclusion	67
9 Bibliography	68
10 Appendix.....	74
10.1 Popular Cartridge Configurations	74
10.2 Important Cartridge Header Fields	74
10.3 Detailed System Memory Map.....	76
10.4 Additional System Timer I/O Register Details.....	77
10.5 Block Diagrams of the Z80 and 8080.....	79
10.6 More Complete List of CPU Instructions	80
10.6.1 8-Bit Loads	80
10.6.2 Stack Operations and 16-Bit Loads	80
10.6.3 8-Bit General ALU, Bit Testing, Setting and Resetting Operations	80
10.6.4 16-Bit ALU Operations	81
10.6.5 8-Bit ALU Bitwise Rotation and Shift Operations	82
10.6.6 Jumps, Restarts and Subroutine Operations	82
10.6.7 Miscellaneous Operations	83
10.6.8 CGB Background Layer Tile Map Attributes	84
10.6.9 Object Layer Tile Attributes	84
10.7 Generated Class Diagram.....	86
10.8 Test Results	87
10.8.1 Blargg's cpu_instrs Tests	87
10.8.2 Miscellaneous Blargg CPU Tests	87
10.8.3 Blargg's cgb_sound Tests	87
10.8.4 Mooneye GB PPU Object Priority Test	89
10.8.5 Manual Program Compatibility Tests	89

1 Abstract

The Game Boy Color was a highly successful handheld video game console released by Nintendo in 1998 to the tune of over 100 million units sold worldwide [1]. The Game Boy Color directly contributed to the creation of some of today's most popular media franchises such as *Pokémon* [2], which saw its first few games released on the system [3].

For this project, I have developed an application that emulates most of the functionality of the Game Boy Color using a Low-Level Emulation (LLE) approach, whereby the internal operation of the system's hardware is replicated using software.

The emulation software allows a user to run most programs developed for the Game Boy Color to a good level of accuracy within a GUI environment. Users can interact with the emulated system joypad via the use of keyboard keypresses as a way of supplying input to programs. Menu options for controlling certain aspects of emulation (such as speed) are also available to the user. Additionally, the application implements the Game Boy Color's backwards-compatibility functionality with its predecessor, the Game Boy, which also enables the seamless emulation of Game Boy programs.

To test the correctness of emulation, a simple command-line debugger was implemented that allows developers to find the source of compatibility issues by manually stepping through the control flow of emulated programs. Furthermore, a range of automated third-party emulator test programs and real games were utilised during development. Simple functionality for displaying the results of completed automated tests was also added to the debugger.

2 Introduction

For this project, I have developed software that aims to emulate most of the functionality of the Nintendo Game Boy Color handheld video game console. This is achieved using a Low-Level Emulation (LLE) approach, in which the operation of the Game Boy Color's hardware and internal design is replicated using software [4].

The emulator allows a user to run programs assembled for both the Game Boy and Game Boy Color on modern computer systems, which otherwise would not be natively possible due to differences in modern computer architecture.

Supported programs include games licensed and released by Nintendo, and unlicensed software such as third-party emulator test programs [5] and homebrew games.

2.1 Brief Description of the Game Boy Color



Figure 1 – An Atomic Purple Game Boy Color [6].

The Game Boy Color was first introduced in Japan by Nintendo on 21st October 1998 as a successor to the Game Boy [7].

Unlike its predecessor, which most notably features a monochrome dot-matrix LCD [8, p. 12] and an 8-bit CPU clocked at ~ 4.2 MHz, the Game Boy Color uses a colour LCD screen with support for up to 32,768 different colours [9], and a CPU clocked at a maximum frequency of ~ 8.4 MHz [10, p. Game Boy Technical Data].

Both systems feature a physical joypad (called the *Control Pad* [11, p. 279]), consisting of a directional pad and four additional buttons for providing user input to programs. A single front-facing speaker is also present below the joypad, along with a stereo headphone jack located on the bottom side of the unit [9].

A Nintendo *Game Link* serial communications port was built-in to each unit, allowing two systems (or more via the use of special adapters [12]) to exchange data with one another using a *Link Cable* [13]. Additionally, the Game Boy Color featured an infrared communications port, allowing for wireless data transfer between two Game Boy Color systems at a maximum range of less than 2 metres [9]; however, this feature was rarely used [14].

Unlike modern personal computers, programs for the Game Boy and Game Boy Color were distributed in physical ROM cartridges (called *Game Paks* [15]). These cartridges were sometimes internally bundled with additional hardware such as battery-packed RAM [10, p. MBC1], which was used for persisting data while the console was powered off.

Importantly, the Game Boy Color's hardware was designed to be backwards compatible with the Game Boy, enabling Game Boy programs to also run on the Game Boy Color; however, programs specifically targeting the Game Boy Color would not necessarily also run on the Game Boy [16].

Over their lifetimes, the Game Boy and Game Boy Color sold a combined total of ~118.7 million units worldwide [1]. Both consoles played large roles in the success of best-selling media franchises such as *Pokémon* [2], which saw its first few games released on both systems [3].

2.2 Terminology

From this point onward, the abbreviations DMG and CGB will be used in most places within this document to refer to the Game Boy and Game Boy Color respectively. These abbreviations are derived from the codenames used by Nintendo to internally distinguish between these two systems [11, p. 3].

The dollar sign (\$) will be used as a prefix for denoting quantities in hexadecimal (base 16); for example, the hexadecimal number \$1F is equivalent to the denary (base 10) number 31.

2.3 Aims and Objectives

The final software aims to emulate the key hardware components of the CGB necessary for enabling the execution and user interactivity of most CGB programs; including hardware such as the CPU, the LCD video and sound controllers, the various memory modules and memory addressing circuitry, certain internal cartridge components, and the physical joystick.

Due to project time constraints, non-essential features such as the DMG and CGB's serial and infrared I/O functionality were not considered for a full implementation. Details on the design and operation of the hardware components considered for emulation can be found in Section 3.

In addition to the emulation-specific aims described, the finished software also aims to implement a basic GUI to enable interactions between the user and the emulated CGB system. The GUI aims to provide a screen area for presenting the LCD's video output, while also providing joypad input for the system via the detection of keystrokes from the user's keyboard. Furthermore, the GUI aims to provide functionality for selecting programs to execute from program ROM images stored within the user's file system.

Finally, an optional set of aims were considered that were subject to available development time. Extra features such as the addition of tools for debugging the emulated CGB hardware, for saving and restoring emulation state (save states [17]), for replaying emulated joypad inputs (movie playback [18]), and the ability to remap joypad button functions to different keys on the user's keyboard were considered for possible implementation.

3 Hardware Background

The CGB contains a variety of hardware components that operate in parallel to one another while the system is powered on. This section aims to explain the operation of these components at a level necessary for understanding the LLE implementation of the software in Section 5.

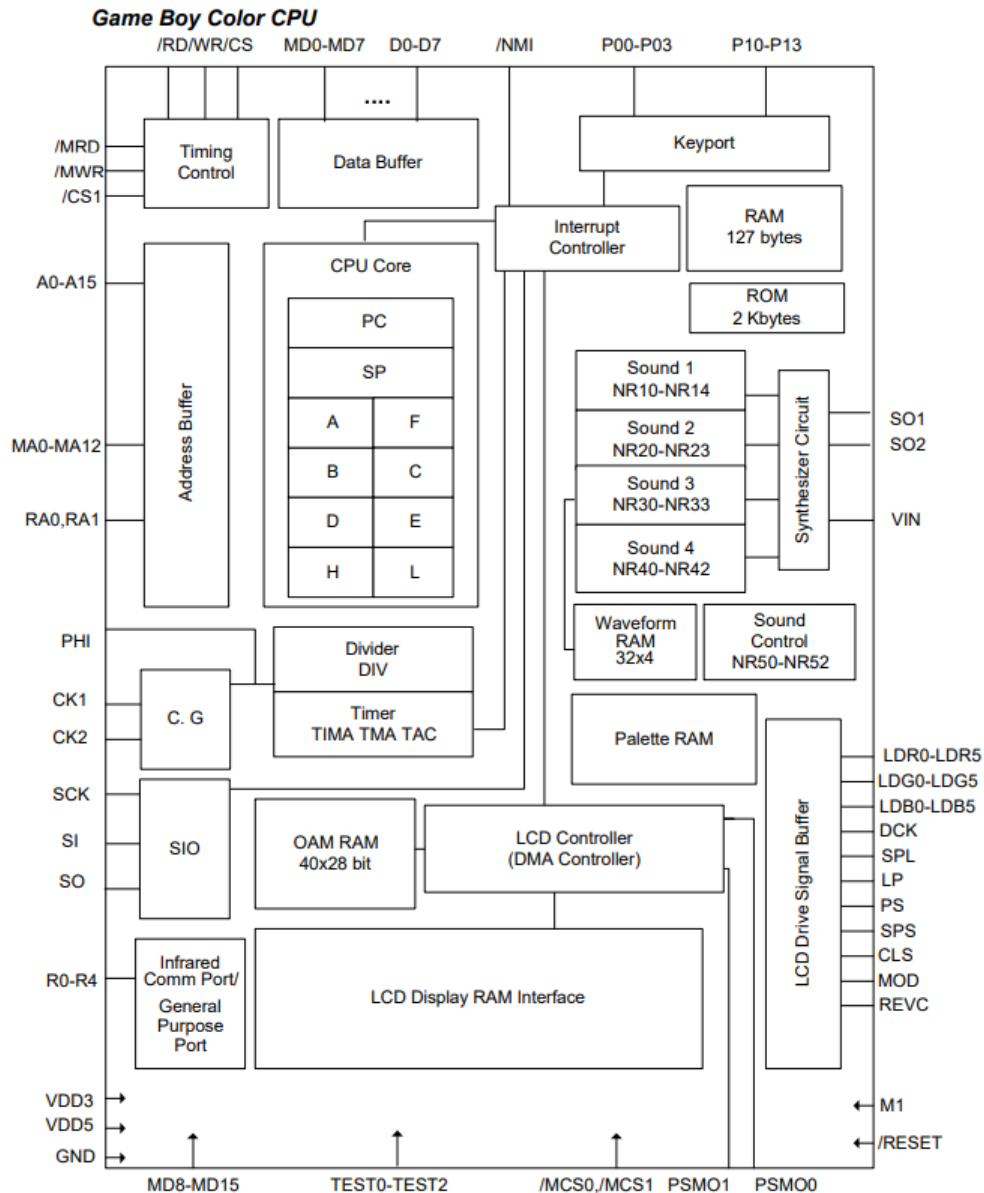


Figure 2 – Block diagram of the CGB CPU chip [11, p. 20].

It should be noted that majority of the CGB's important hardware components such as the CPU core, video and audio controllers are all integrated within the same *Sharp LR35902* CPU chip [11, p. 20] [19]. In this document, these components are generally referred to as separate entities to aid understanding.

3.1 Memory and System Start-up

The DMG and CGB processors utilise a 16-bit address bus that is connected to the system's various memory modules, allowing the access of up to 2^{16} (65,536) different memory locations at any one time. Each memory location is 8 bits (a byte) in size [10, p. Memory Map].

3.1.1 System Memory

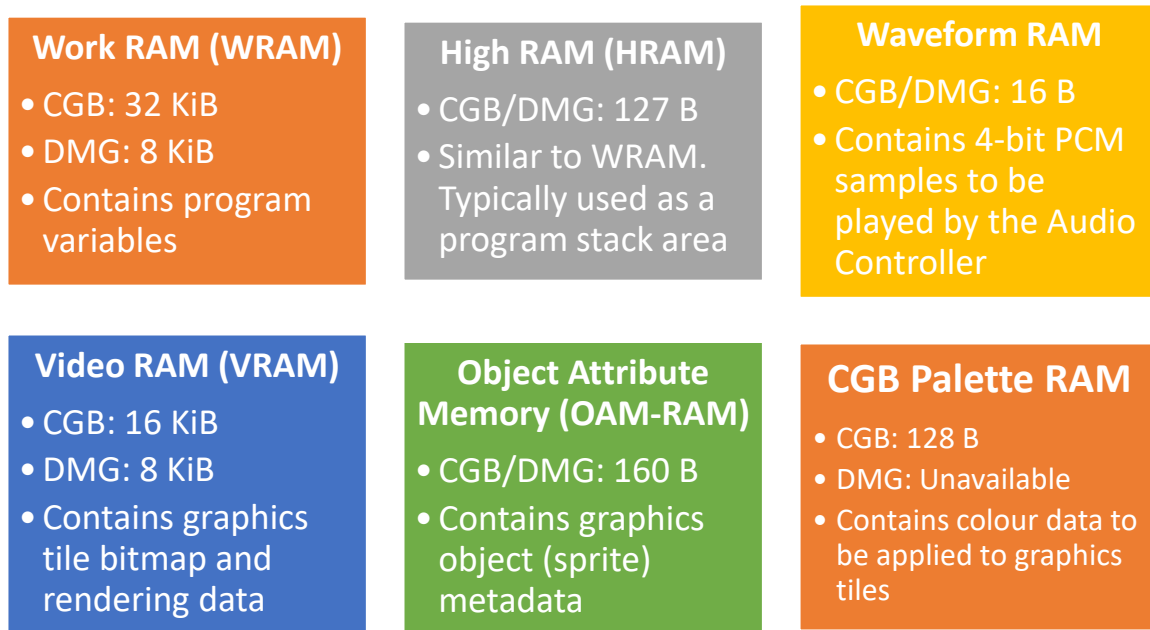


Figure 3 – A simplified diagram of the Game Boy's six main system memory modules [10, p. Memory Map].

The CGB contains six main types of system memory that is utilised during program runtime: work RAM (WRAM), High RAM (HRAM), waveform RAM, video RAM (VRAM), Object Attribute Memory (OAM or OAM-RAM) and CGB Palette RAM. Unlike the CGB, the DMG does not contain Palette RAM, including only five modules in total.

Eight 4 KiB work RAM (WRAM) banks, 32 KiB in total, are available for use by the CGB CPU for maintaining the state of executing programs. This memory contains the data of variables in-use by the running program (such as player position and speed in games) and is frequently read from and written to during runtime. In contrast to the CGB, the DMG contains only two banks of WRAM [10, p. Memory Map].

The DMG and CGB both contain a 127-byte area of memory called High RAM (HRAM) that functions like WRAM. HRAM is intended to be an area to store the program stack (see Section 3.2.4 for more details).

A 16-byte area known as waveform RAM is present in both the DMG and CGB. The most and least significant 4-bits (nibbles) of each byte within waveform RAM are used to encode two 4-bit PCM audio samples, allowing for 32 samples to be stored at once [20]. Waveform RAM was exclusively used by programs to output digital sound such as custom instruments and voices using the processor's third sound channel.

Two 8 KiB video RAM (VRAM) banks, 16 KiB in total, are usable by both the CGB CPU and LCD video hardware during program runtime for storing graphics tile bitmap and rendering data. The DMG has only one bank of VRAM [10, p. Memory Map].

In addition to VRAM, both systems also contain a 160-byte memory area called Object Attribute Memory (OAM or OAM-RAM), which is used to store graphics object metadata (for sprites) [10, p. Memory Map].

Finally, the CGB contains a 128-byte memory area called Palette RAM, which is used to store RGB intensity values for colours to be applied to graphics tile patterns stored within VRAM. This memory area is not accessible in DMG compatibility mode [10, p. LCD Color Palettes (CGB only)].

More information regarding the data stored within VRAM, OAM-RAM and Palette RAM is contained within Section 3.5.

3.1.2 Program Cartridges



Figure 4 – A Game Boy game cartridge for Tetris (1989) [21].

Programs for the DMG and CGB were distributed on physical ROM cartridges that could be inserted into the back of the system to be loaded.

Cartridges were distributed with varying amounts of program ROM, with each ROM bank being 16 KiB in size. Under normal circumstances, a DMG or CGB can only address two ROM banks, 32 KiB of ROM in total, at any given time [10, p. Memory Map].

To support programs larger than 32 KiB, Memory Bank Controllers (MBCs) were integrated within cartridges, allowing actively mapped ROM banks to be swapped by the program during runtime [10, p. Memory Bank Controllers].

In addition to program ROM, some cartridges contained varying amounts of dedicated RAM, with each bank of RAM being 8 KiB in size [10, p. Memory Map]. In most cases, this additional RAM was used to store data to be retrieved later after the console was powered off (such as high scores for games); however, due to the volatile nature of RAM, an internal battery would also be included to supply power to the RAM even while the console was powered off. A list of popular cartridge hardware configurations can be found in the appendix: Section 10.1.

The first ROM bank (bank 0) of the cartridge contains 80 bytes of header information (starting at memory address \$0100 within the bank) that is read by the DMG and CGB boot procedure before program execution happens (see Section 3.1.4). A list of important fields within the cartridge header can be found in the appendix: Section 10.2.

3.1.3 Memory Mapping

Address Range	Memory Area
\$0000 to \$7FFF	Cartridge ROM
\$8000 to \$9FFF	VRAM
\$A000 to \$BFFF	Cartridge RAM (<i>if any</i>)
\$C000 to \$DFFF	WRAM
\$FE00 to \$FE9F	OAM-RAM
\$FF00 to \$FF2F	I/O Registers
\$FF30 to \$FF3F	Wave RAM
\$FF40 to \$FF7F	I/O Registers
\$FF80 to \$FFFE	HRAM
\$FFFF	I/O Registers

Figure 5 – A simplified layout of the Game Boy’s system memory map.

Like most video game consoles and computer systems at the time, the memory modules of the DMG and CGB were mapped to a single address space [22]. Due to the 16-bit wide address bus utilised by the DMG and CGB, memory addresses may range from \$0000 to \$FFFF (65,535, or $2^{16}-1$).

In addition to memory modules such as RAM and ROM, an area for accessing hardware I/O registers and flags is also mapped into the system’s address space. Accessing these registers allows the running program to influence the operation of

the system's different hardware components. The purpose and operation of some of these registers are explained in later subsections.

Specific details regarding the layout of the Game Boy memory map can be found in the appendix: Section 10.3.

3.1.4 Boot Procedure



Figure 6 – The screen displayed during the CGB's boot procedure [23].

When the console is powered on, before the execution of the loaded program begins, a small procedure is run from the boot ROM contained within the CPU chip [10, p. Power Up Sequence] [19]. This procedure displays a short animation of the Game Boy and Nintendo logos while playing an audible tone at the end.

More importantly, the boot procedure reads the header of the loaded program ROM; using this data, the boot procedure can lockout programs that fail certain data integrity and licensing checks, while also determining whether the program should be ran in DMG backwards compatibility mode.



Figure 7 – The screen shown by the CGB version of Donkey Kong Country (2000) when ran on a DMG.

After the procedure completes, the state of certain system memory locations and registers are left with leftover values from the boot procedure. The value of the CPU's Accumulator (A) register (see Section 3.2.1) can be read after booting to reliably determine the type of Game Boy in use [11, p. 120], allowing programs to display compatibility warnings for unsupported systems if necessary.

3.2 CPU

The DMG and CGB both utilise the *Sharp LR35902* CPU: a little-endian CPU with an 8-bit data bus that was designed to be a hybrid between the *Intel 8080* and *Zilog Z80* (block diagrams for these CPUs are available within the appendix: Section 10.5). The *Z80* was designed to be binary-code compatible with the *8080*, allowing both processors to execute the same machine code [24].

3.2.1 CPU Registers

The CPU features a total of 10 registers that are located locally to the CPU core [11, p. 20]. These registers are the fastest memory locations accessible to the CPU and are either 8 or 16 bits in size [10, p. CPU Registers and Flags].

The purpose of these registers are as follows: [10, p. CPU Registers and Flags] [25]

Register	Function	Size	Details
B, C, D, E, H, L	General-Purpose	8 bits each	Can be written to or read by the program for any purpose.
A	Accumulator	8 bits	Contains the result of the previously executed arithmetic or logical operation.
F	Status Flag Register	8 bits	<p>Contains read-only status information regarding the previously executed arithmetic or logical operation.</p> <p>This information is represented using bits 4 to 7 of the register as follows: (see Section 3.2.2)</p> <ul style="list-style-type: none">• Bit 4 – Carry Flag.• Bit 5 – Half-Carry Flag.• Bit 6 – Subtraction Flag.• Bit 7 – Zero Flag.
PC	Program Counter	16 bits	<p>Contains the memory address of the next program instruction to be executed by the CPU.</p> <p>After booting, the PC is set to the program entry at address \$0100 within the ROM header [10, p. The Cartridge Header].</p>
SP	Stack Pointer	16 bits	Contains the memory address of the 16-bit value currently at the top of the program stack.

Additionally, the 8-bit CPU registers are also accessible as the following 16-bit register pairs by some program instructions: AF, BC, DE and HL [10, p. CPU Registers and Flags].

3.2.2 ALU

The CPU utilises an Arithmetic Logic Unit (ALU) for performing 8-bit integer arithmetic and logical operations. Supported ALU functions include: [26]

- Addition and Subtraction
- Increment and Decrement
- Comparisons
- Bitwise Logical AND, OR and XOR
- Bitwise Arithmetic and Logical Shifts and Rotations
- Setting, Resetting and Testing Individual Bits

The ALU provides output lines for retrieving additional information regarding the previously invoked operation. These lines are used to define the following flags within the CPU's Status Flag (F) register: [27]

- **Carry Flag** – Set if the 8-bit result of the operation generated a carry from its most significant bit (bit 7); unset otherwise.
This carry can be used with a more significant byte to enable arithmetic calculations for integers larger than 8 bits in size; however, this cannot be done as a single operation.
- **Half-Carry Flag** – Set if the 8-bit result of the operation generated a carry from bits 3 to 4; unset otherwise.
- **Subtraction Flag** – Set if the previous operation was a subtraction-type operation; unset otherwise.
- **Zero Flag** – Set if the result of the operation was zero; unset otherwise.
Frequently used within program conditional branches.

3.2.3 Instructions

The CPU defines a set of commands, known as instructions, which are responsible for driving all algorithmic logic within programs (which themselves are simply collections of instructions and data).

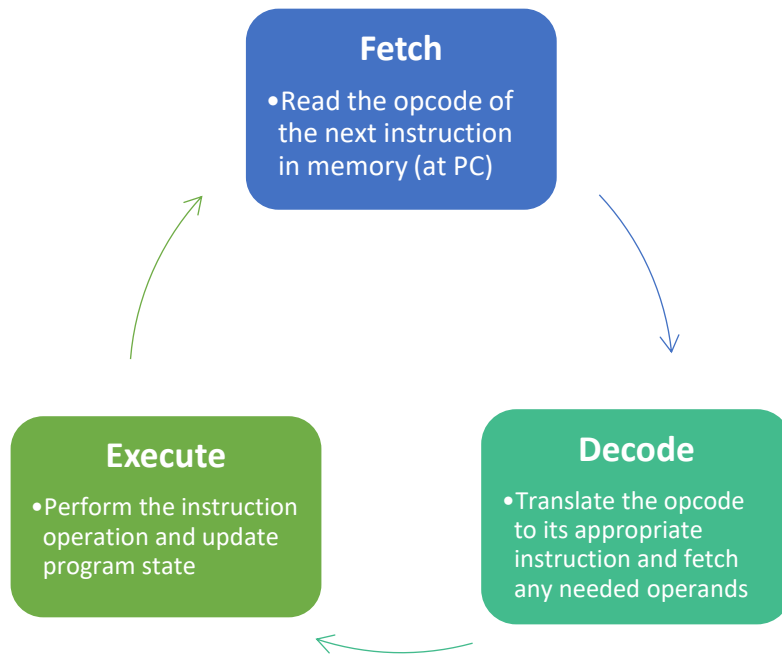


Figure 8 – A diagram explaining the simplified operational process of the CPU known as the Fetch-Decode-Execute (FDE) cycle.

Within memory, program instructions are typically encoded as 8-bit machine code values known as ‘opcodes’. These opcodes are continuously read from memory, decoded into the appropriate instruction and executed in what is known as the Fetch-Decode-Execute (FDE) cycle. The CPU’s Program Counter (PC) register is used to point to the memory location containing the next program opcode to be fetched for execution.

Some noteworthy instruction operations that the CPU can perform include: [26]

- **8 and 16-Bit Loads** – *Instructions: LD, LDI, LDD, LDH.*
These instructions are used to write data to system memory locations or to the CPU’s 8 or 16-bit registers and register pairs.
- **8 and 16-Bit ALU Operations** – *Some instructions include: ADD, SUB, AND, OR, XOR, CP, INC, DEC, BIT, SET, RES.*
Arithmetic and logical operations using the CPU’s ALU.
- **16-Bit Stack Operations** – *Instructions: PUSH, POP.*
Program stack pushes and pops as described in Section 3.2.4.
- **Jumps and Subroutine Calls** – *Instructions: JP, JR, CALL, RST.*
Operations for setting the value of the Program Counter (PC) register,

allowing programs to execute instructions in specific areas of memory.

- **Subroutine Returns** – *Instructions: RET, RETI.*
Operations for restoring the value of the Program Counter (PC) to the address of the instruction after the currently active subroutine call (as described in Section 3.2.4).
- **Master Interrupt Controls** – *Instructions: EI, DI.*
Enables or disables all interrupts from being handled as described in Section 3.2.6.
- **Execution Halt Controls** – *Instructions: STOP, HALT.*
Commands for temporarily halting the execution of program instructions. Generally used as a power saving mechanism to extend the battery life of the system by programs when there is currently no more work to do.

The Game Boy CPU defines a total of 500 instruction opcodes split into tables of 244 and 256 opcodes respectively [28]. Due to the typical 8-bit size limitation of instruction opcodes (allowing for only 256 unique opcode values in a byte), the opcode \$CB is reserved as a prefix for accessing opcodes within the larger 256-opcode table. This technicality effectively makes each opcode within the 256-opcode table 16 bits large, with \$CB being the value of the most-significant byte.

Many different opcodes can be used to refer to the same CPU operation, but with an assortment of different input or output operands in use. These configuration types are known as addressing modes and are responsible for specifying how the CPU locates and interprets instruction operands.

The addressing modes supported by the CPU are: [27, pp. 44-48]

- **Immediate** – The next 8 bits after the opcode is the operand.
- **Immediate Extended** – The next 16 bits after the opcode is the operand.
- **Relative** – The next 8 bits after the opcode is a signed two's complement offset value. This offset value is added to the memory address of the next instruction to produce the operand.

- **Extended** – The next 16 bits after the opcode is the memory address of the operand to use.
- **Register** – The operand is a CPU register.
- **Implied** – The operand is implied to be a CPU register by the opcode itself. For example, instructions for invoking ALU operations already imply that the CPU Accumulator register (A) is the output operand for results.
- **Register Indirect** – The operand is a 16-bit CPU register pair.
- **Bit** – The operand is a specific bit of a given value. This given value is supplied as another operand via the use of the Register, Register Indirect or Indexed addressing modes.

A more complete list of available instructions with additional details can be found within the appendix: Section 10.6.

3.2.4 Program Stack

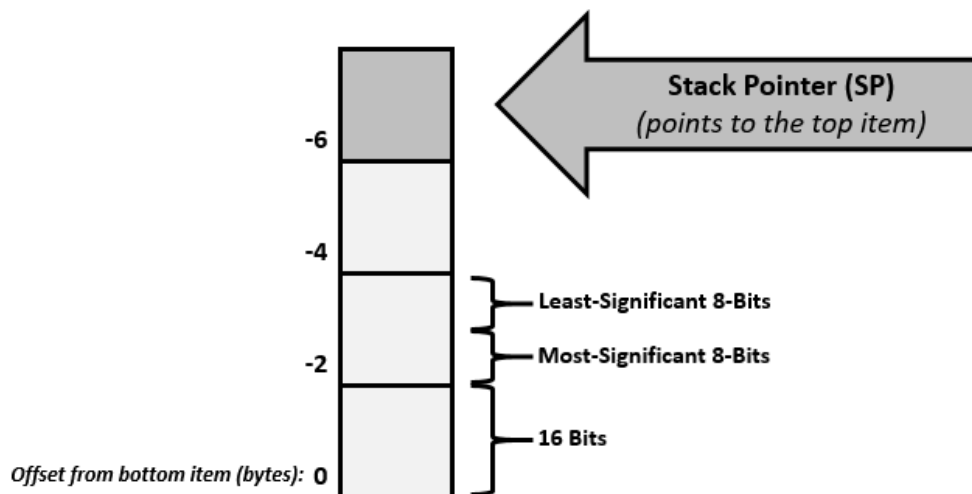


Figure 9 – A simple diagram of a program stack containing four elements (as an example).

The program stack is a stack (last-in, first-out) data structure mainly used by the CPU as a call stack for keeping track of the return addresses of called program subroutines. It is also frequently directly utilised by programs for other algorithmic purposes. Each individual item stored within the stack is 16 bits in size.

The CPU's Stack Pointer (SP) register is used to store the memory address of the current topmost item within the stack. By default, the system boot procedure sets the value of the Stack Pointer to the address \$FFFE (HRAM) [29]; however, this value may be changed during program execution via the use a specific LD instruction (LD SP, HL: opcode \$F9 [28]).

The program instructions PUSH and POP may be used to manually push to and pop from the top of the stack respectively. The program stack grows “downwards” in memory, meaning that the memory addresses of items near the top of the stack are lower than that of those near the bottom [11, pp. 90-91].

It should be noted that there are no explicit restrictions limiting the value of the Stack Pointer or the maximum size of the program stack. The responsibility is left to the programmers to ensure that the contents of the stack does not overflow into unrelated areas of memory. If the stack overflows, the contents of the overflowed memory addresses will be overwritten with stack data, likely leading to program bugs and crashes.

3.2.5 Clock

Once powered on, the Game Boy CPU runs at a 4.194304 MHz clock frequency. This clock frequency is responsible for synchronizing the operation of many system hardware components [10, p. Game Boy Technical Data]. All hardware operations take several clock cycles to complete, with all CPU operations aligned to 4 clock cycle intervals (known as machine cycles).

Unlike the DMG, the CGB implements a togglable double speed mode for the CPU clock, allowing running programs to optionally increase the clock frequency to 8.388608 MHz [10, p. Game Boy Technical Data]. The running program may request to enable or disable double speed mode by writing to bit 0 of the Prepare Speed Switch (KEY1: address \$FF4D) I/O register.

After KEY1 has been written to, the program must then execute the STOP instruction to start the speed switching process to or from double speed mode [10, p. CGB Registers]. The speed switch takes an additional 130,992 clock cycles to complete [30, p. 12].

When double speed mode is enabled, numerous hardware components and system features operate twice as fast: [10, p. CGB Registers]

Operate twice as fast in double speed mode:	Operate at the same speed in double speed mode:
<ul style="list-style-type: none"> • The CPU • The System Timer • OAM-RAM DMA (see Section 3.5.2) • Serial I/O 	<ul style="list-style-type: none"> • The LCD Video Controller • The Audio Controller • The CGB's H-Blank and General-Purpose DMA (see Section 3.5.2)

3.2.6 Interrupts

Interrupts are signals that are used to indicate that another event requires immediate attention by the CPU. Interrupts temporarily postpone the normal execution of the program to instead execute instructions contained within program-defined subroutines known as Interrupt Service Routines (ISRs).

Interrupt Type	IVT Entry Address <i>(within program ROM bank 0)</i>	IF/IE Mask Bit Number <i>(IF: IRQ requested if bit is set IE: Interrupt enabled if bit is set)</i>
LCD V-Blank	\$0040	0
LCD STAT	\$0048	1
System Timer	\$0050	2
Serial I/O	\$0058	3
Joyypad	\$0060	4

Figure 10 – The Game Boy CPU's five maskable interrupt types (listed in descending order of handling priority) [10, p. Interrupts].

The Game Boy CPU defines five different maskable interrupt types which can be requested by the system's various hardware components or by the program itself.

The ISRs for each interrupt type are defined within an Interrupt Vector Table (IVT) located at address \$0040 within bank 0 of the program ROM, where each IVT entry is 8 bytes in size [31]. Due to the small size of the IVT, the majority of program ISR machine code is usually stored elsewhere in ROM, with IVT entries typically containing jump instructions to these areas.

Before any interrupts can be triggered, the CPU's Interrupt Master Enable (IME) flag must be set: the program instructions EI and DI are used to set and unset the IME flag respectively. When an interrupt is about to be handled, the IME flag will be automatically unset by the CPU before the appropriate ISR is called. Normally,

returning from an ISR is done using the RETI instruction, which has the same functionality as a RET and EI instruction [10, p. Interrupts].

The CPU exposes two I/O registers for managing each individual interrupt type: The Interrupt Flag (IF: address \$FF0F) and Enable (IE: address \$FFFF) registers. The IF register can be used to perform an Interrupt Request (IRQ) for a specific type of interrupt, while the IE register can be used to enable or disable certain IRQs from being handled by the CPU (provided that the IME flag is set) [10, p. Interrupts]. Refer to Figure 10 to see which interrupt types are affected by each bit within IF and IE.

Whenever multiple IRQs are simultaneously active (more than one bit set within the IF register), the interrupt type with the highest handling priority will be processed first by the CPU. The handling priorities for IRQs are determined by the position of their corresponding bits within IF, with the least significant bits having priority over the most significant bits [10, p. Interrupts].

3.3 System Timer

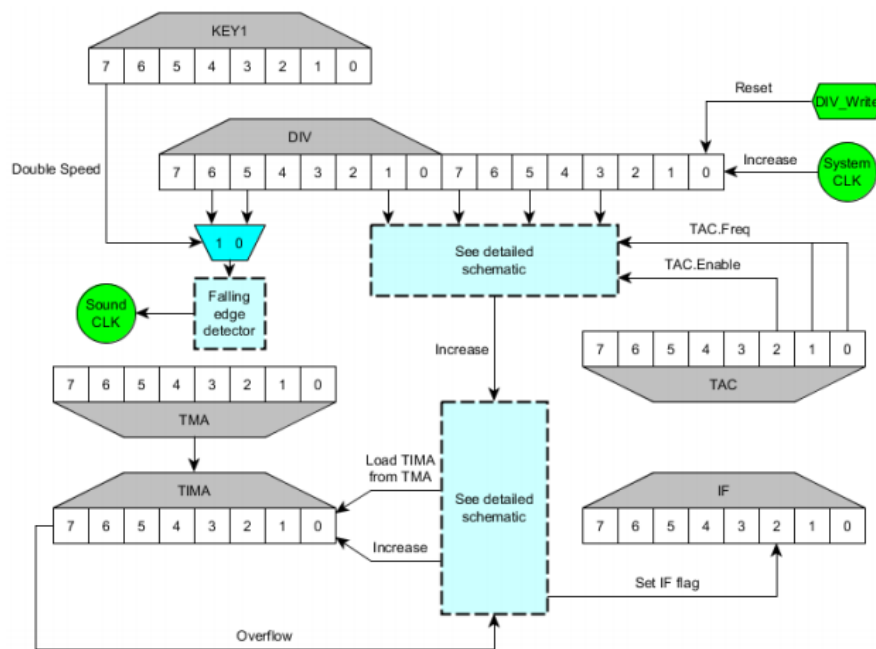


Figure 11 – A simplified schematic of the system timer's circuitry [30, p. 18].

The DMG and CGB contain an internal timer that allows programs to schedule tasks at periodic time intervals. This functionality was generally used by programs as a way of maintaining the speed of music playback and certain time-critical events. Unlike a Real-Time Clock (RTC), the internal timer does not provide functionality for keeping track of the current date or time of day.

The internal timer can tick at four different base frequencies, ranging from 4,096 Hz to 262,144 Hz. When the CPU clock is in double speed mode, the internal timer ticks twice as fast [10, p. Timer and Divider Registers]. The rate in which the timer ticks is configurable via writes to the Timer Control I/O register (TAC).

For every tick of the timer, the value of the Timer Counter I/O register (TIMA) is incremented. When this 8-bit value overflows, it is reset with the value contained within the Timer Modulo I/O register (TMA) and a timer interrupt is requested [10, p. Timer and Divider Registers].

In addition to the previously described registers, an 8-bit Divider I/O register (DIV) also exists, which increments its value every 256 CPU clock cycles (16,384 Hz if CPU double speed mode is disabled) [10, p. Timer and Divider Registers]. Video games such as *Tetris* (1989) utilise this register to seed their pseudorandom number generators [32].

Additional details regarding the system timer's I/O registers can be found in the appendix: Section 10.4.

3.4 Joypad



Figure 12 – An illustration of a Game Boy joypad [33].

The Game Boy joypad consists of a directional pad along with four additional keys for accepting user input for programs.

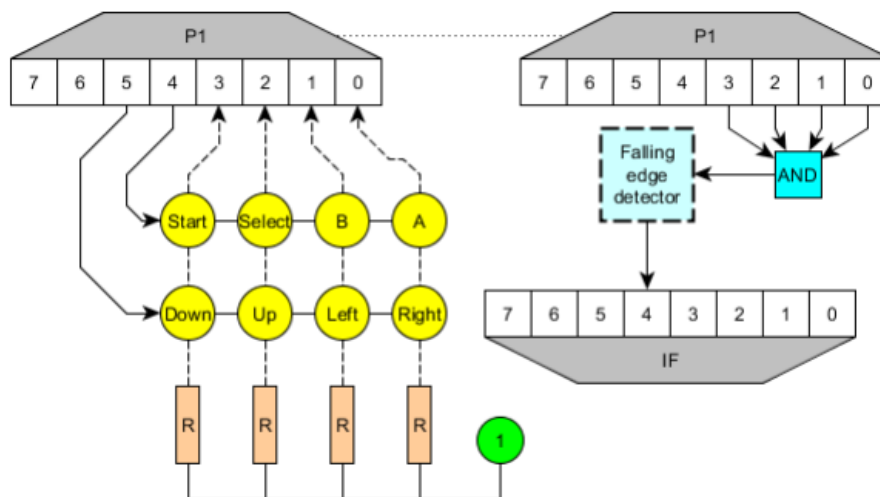


Figure 13 – A schematic of the joypad's I/O circuitry [30, p. 25].

Programs can query the status of the joypad by reading from bits 0 to 3 of the JOYP I/O register (address \$FF00). At any one time, only the status of either the joypad's directional pad or button keys can be selected by the program for reading. This selection can be changed by writing to bit 4 or 5 of the JOYP register [10, p. Joypad Input].

A special type of interrupt exists for joypad events that is requested when a selected key is pressed. However, due to a flaw in the design of the joypad keys in most Game Boy variants, multiple interrupts may be requested once a selected key is pressed due to switch bounce [34].

Due to the lack of an inverter within the joypad I/O circuitry, the selection and status bits of the JOYP register are unset (rather than set) to indicate key selection and active button presses [30, p. 25].

The meaning of the individual status bits of JOYP are as follows: [10, p. Joypad Input]

If the direction keys are selected: <i>(bit 4 of JOYP unset)</i>	If the button keys are selected: <i>(bit 5 of JOYP unset)</i>
The following bits are unset if the corresponding key is being pressed:	
<ul style="list-style-type: none"> • Bit 0 – A. • Bit 1 – B. • Bit 2 – Select. • Bit 3 – Start. 	<ul style="list-style-type: none"> • Bit 0 – Right. • Bit 1 – Left. • Bit 2 – Up. • Bit 3 – Down.

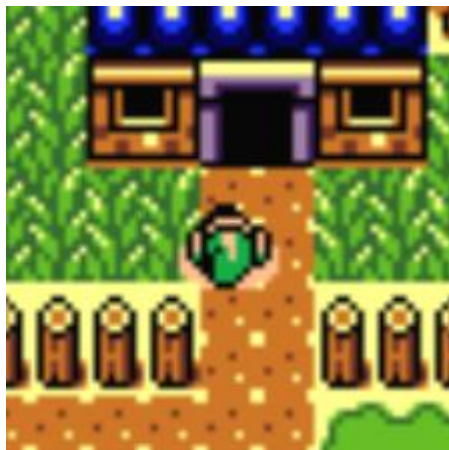


Figure 14 – A glitch in The Legend of Zelda: Link's Awakening DX (1998) showing the player character swimming inside of the ground. The glitch was performed by pressing opposite directional keys simultaneously in a certain pattern.

It should be noted that due to the physical layout of the Game Boy's directional pad, it is impossible to press either the left and right or up and down keys simultaneously. Some programs inhibit glitches if this restriction is violated as seen in Figure 14.

3.5 LCD Video Controller (PPU)

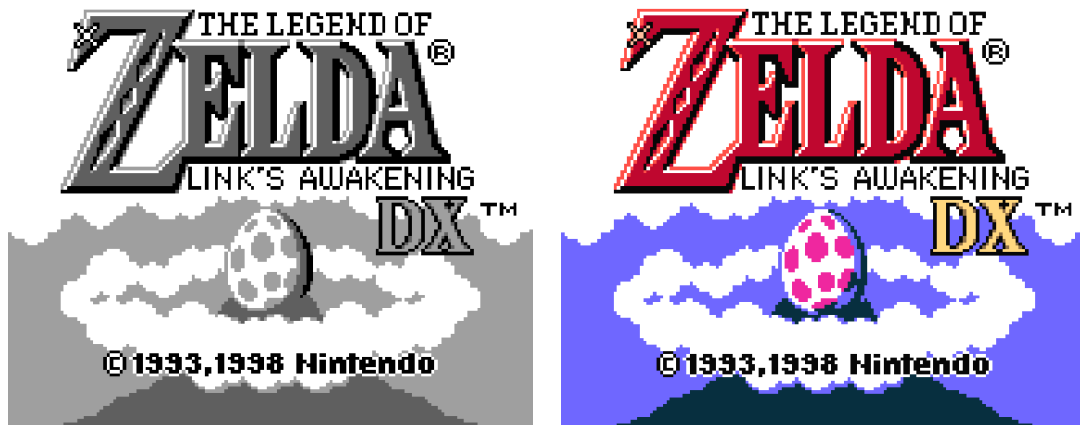


Figure 15 – The title screen from *The Legend of Zelda: Link's Awakening DX* (1998) in DMG compatibility mode (left) and CGB mode (right).

The LCD video controller, also known as the Picture (or Pixel) Processing Unit (PPU), is the hardware component responsible for rendering and presenting colour graphics (or monochrome graphics in DMG compatibility mode) onto the CGB's 160×144-pixel resolution colour LCD [10, p. Game Boy Technical Data].

3.5.1 General Operation

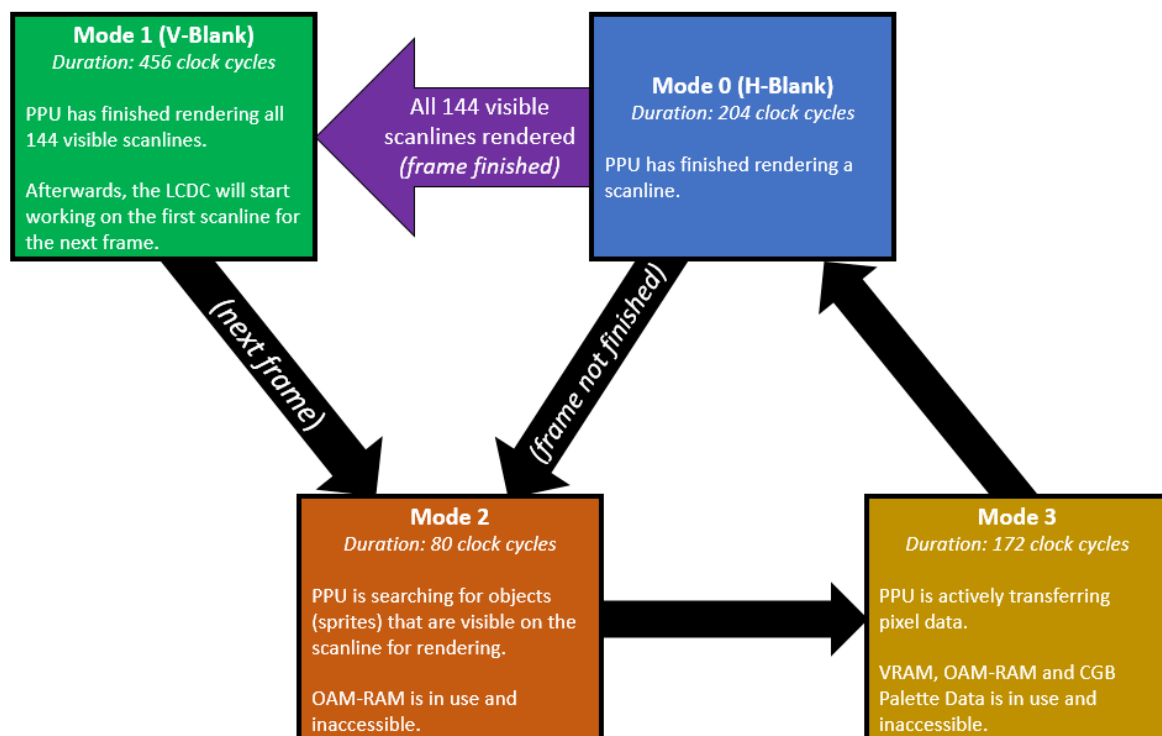


Figure 16 – A diagram detailing the operative cycle of the PPU [10, p. LCD Status Register] [35].

The CGB's PPU operates on a per-scanline basis, rendering each row of 160 pixels starting from the top of the LCD screen going downwards. For each scanline, the PPU continuously alternates between three different modes of operation until a full frame of video (144 scanlines long) has been rendered. Once an entire frame has been rendered, the PPU enters an intermediary mode known as the Vertical Blanking Interval (V-Blank: mode 1) and will immediately request a V-Blank interrupt for the program to handle: this occurs ~59.723 times per second [10, p. Game Boy Technical Data]. Figure 16 describes these modes and the operative cycle in more detail.

Throughout modes 2 and 3, the PPU starts reading from certain memory locations such as VRAM, OAM-RAM and CGB Palette RAM. During this period, these memory locations become inaccessible to the CPU, causing program reads to return \$FF and writes to have no effect on memory [10, p. LCD Status Register] [36]. The V-Blank interrupt is especially important for this reason, as it provides ways for programs to upload new data to VRAM, OAM-RAM and Palette RAM as soon as these memory locations are available again for use.

The LCD Status (STAT: address \$FF41) I/O register may be configured to allow the PPU to generate a STAT interrupt when one or more of the following events occur: [10, p. LCD Status Register]

- **Mode 0 (H-Blank) started (Bit 3 set)** – like V-Blank, memory such as VRAM, OAM-RAM and CGB Palette RAM are available for use during this period. However, due to the shorter duration of H-Blank when compared to V-Blank (204 vs 456 clock cycles), programs using this interrupt typically only copy small amounts of data to PPU memory during this period.
- **Mode 1 (V-Blank) started (Bit 4 set)** – for all intents and purposes, interrupting on this event will replicate the functionality of the PPU V-Blank interrupt.
- **Mode 2 started (Bit 5 set)** – the transition to mode 2 indicates that the PPU is about to start working on the next scanline. Interrupting on this event allows programs to cease any active transfers to PPU memory (OAM-RAM specifically) that they may be performing, for example.
- **LY coincidence (Bit 6 set)** – this event indicates that the current scanline number, indicated by the read-only LY I/O register (address \$FF44), equals the value of the LYC I/O register (address \$FF45). Interrupting on this event

allows programs to easily perform graphical effects starting from a specific vertical position on the screen, for example.

3.5.2 Tiles

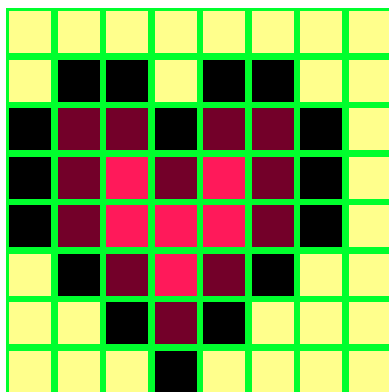


Figure 17 – An example of an 8×8-pixel tile with gridlines from The Legend of Zelda: Link's Awakening DX (1998).

All graphics rendered by the PPU comprise of 8×8 bitmap images known as tiles. Each tile is encoded using 16 bytes of pattern data, with each 8-pixel line consuming 2 bytes of memory [10, p. VRAM Tile Data].

Tile patterns are stored within Tile Pattern Tables located within either the first or second banks of VRAM, mapped to addresses \$8000 to \$8FFF and \$8800 to \$97FF in memory, allowing for the patterns of up to 384 tiles to be held in VRAM at any given time. In DMG compatibility mode, only bank 0 of VRAM is accessible to the program, restricting the maximum number of tile patterns within VRAM to 192 [10, p. VRAM Tile Data].

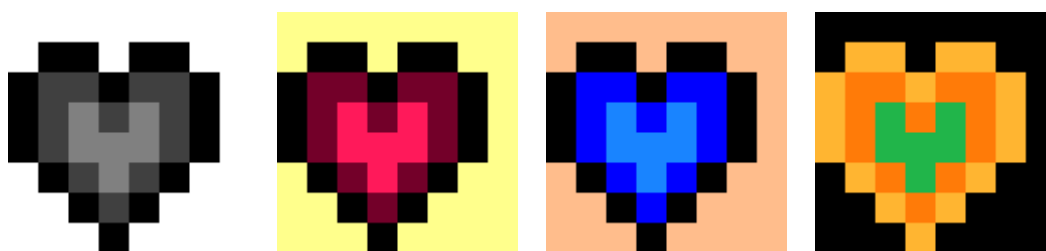


Figure 18 – An example of different CGB mode palette configurations being applied to the same tile graphic from Figure 17.

Tile pixel colours are not encoded as separate RGB components inside of tile pattern data as one may expect, but rather as 2-bit numbers referencing colours within palette entries in the PPU's palette colour table. The least and most-significant bytes of an encoded tile pattern row contain the least and most-significant bits of each pixel's palette colour number respectively, from left to right [10, p. VRAM Tile Data].

The location and structure of the palette colour table is dependent on whether the system is in DMG compatibility mode or not:

In CGB mode, Palette RAM is used to store colours as 15-bit RGB intensity values (each occupying 2 bytes of memory), with each intensity component being 5 bits in size (values ranging from \$00 to \$1F). Palette RAM has the capacity to store 64 colours: 32 colours for tiles on the background and object layers respectively (see Section 3.5.3) [10, p. LCD Color Palettes (CGB only)].

In DMG compatibility mode, the monochrome palette registers BGP, OBPO and OBP1 (addresses \$FF47, \$FF48 and \$FF49 respectively) are used instead. The BGP and OBP registers are used to assign one of four different shades of grey for each palette table index for tiles on the background and object layers respectively [10, p. LCD Monochrome Palettes].

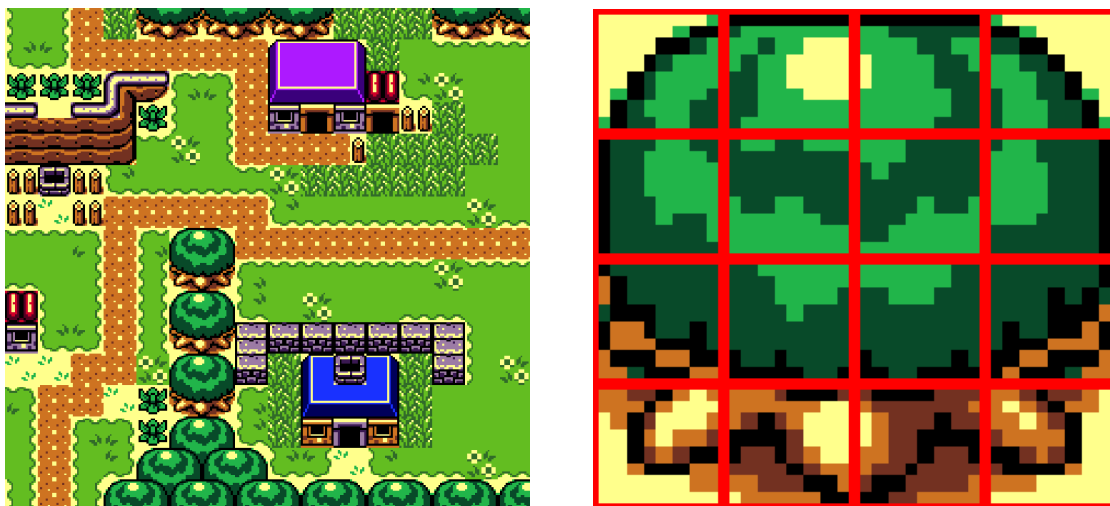


Figure 19 – An example of a background tile map from The Legend of Zelda: Link's Awakening DX (1998), with a 4×4-tile subsection with gridlines on the right.

All on-screen graphics to be rendered by the PPU are produced using an assortment of tiles called a tile map, which is 32×32 tiles in size. Two tile map data structures, each 32×32 bytes large, reside within VRAM bank 0: mapped to addresses \$9800 to \$9BFF and \$9C00 to \$9FFF in memory [10, p. VRAM Background Maps]. Each byte within the tile map is used as an offset value referring to a specific tile within the Tile Pattern Table. Due to the 160×144-pixel resolution of the LCD screen, only a 20×18-tile sized viewport of any tile map can be shown on-screen at once.

3.5.3 Layers

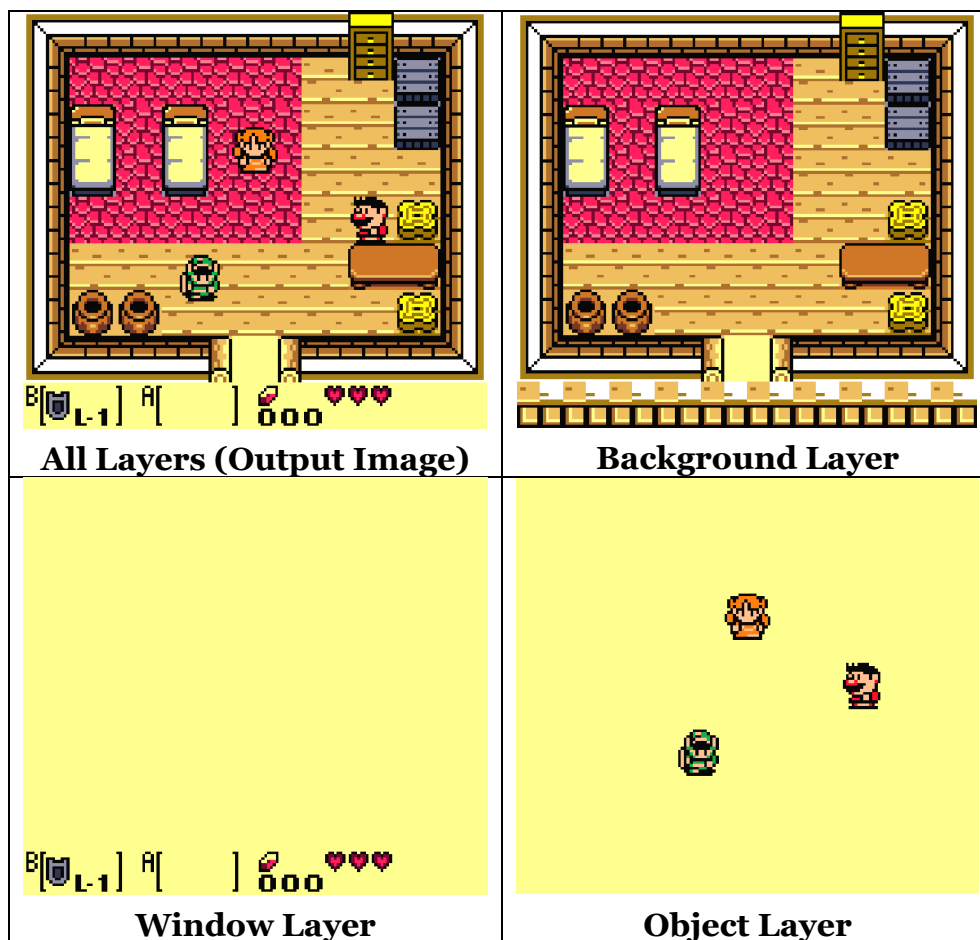


Figure 20 – The different graphics layers shown from within *The Legend of Zelda: Link's Awakening DX* (1998).

The PPU defines three graphics layers which are combined during the rendering process to create a full image: the background, window and object layers.

3.5.3.1 Background Layer

The background is the backmost graphics layer typically used for displaying large images which cover the entire screen area. It may be composed using either one of the PPU tile maps, selectable via bit 3 of the LCD Control (LCDC: address \$FF40) I/O register [10, p. LCD Control Register].

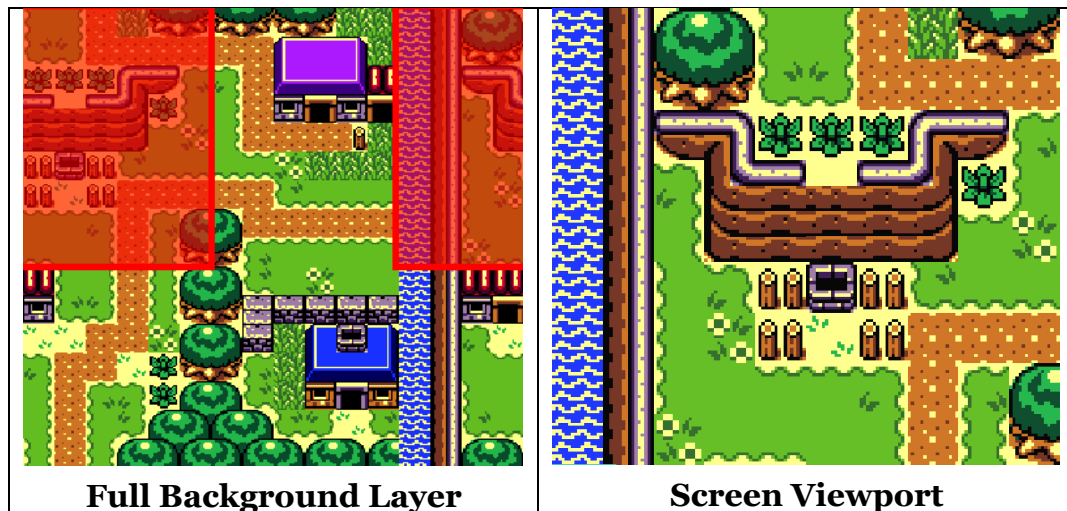


Figure 21 – An example of the screen viewport (in red) wrapping around the background (left), with the contents of the viewport (right) in *The Legend of Zelda: Link's Awakening DX* (1998).

The screen contents of the background tile map can be scrolled by writing pixel offset values from the top-left corner of the screen to the Scroll X (SCX: address \$FF43) and Scroll Y (SCY: address \$FF42) I/O registers. If this screen viewport is scrolled outside of the bounds of the background area, the PPU will automatically wrap the viewport around to the opposite side of the layer tile map as shown in Figure 21 [10, p. LCD Position and Scrolling].

In CGB mode, tiles within the background tile map have several editable attributes affecting how they are rendered. These attributes are stored within one of two 32×32-byte data structures known as background attribute tables. Like the two available tile maps, the attribute tables reside at either address \$9800 to \$9BFF or \$9C00 to \$9FFF in memory depending on which map is used for the background, but within VRAM bank 1 instead of 0. Attributes are indexed in the same order as their related tile pattern numbers in the attribute table's corresponding tile map [10, p. VRAM Background Maps].

Details regarding the attributes stored within the background attribute table can be found in the appendix: Section 10.6.8.

3.5.3.2 Window Layer

The window is the graphics layer above the background typically used to display text and UI elements. The window functions like a second movable background layer with mostly the same configuration options available but does not include the background's screen viewport wrapping functionality. The tile map used for the

window can be selected via bit 6 of the LCDC I/O register [10, p. LCD Control Register].

The screen viewport of the window tile map may be moved by writing pixel offset values from the top-left corner of the screen to the Window X (WX: address \$FF4B) and Window Y (WY: address \$FF4A) I/O registers. It should be noted that the WX register holds the viewport's horizontal offset value minus 7, allowing the left side of the window to be positioned partially off-screen [10, p. LCD Position and Scrolling].

3.5.3.3 Object Layer

The object layer is the frontmost graphics layer used to display objects (or sprites) which move independently from the other background layers. All objects are composed using tiles and are either 8×8 or 8×16 pixels in size, configurable via bit 2 of the LCDC I/O register [10, p. LCD Control Register].

The contents of the object layer are determined by the attribute data stored within OAM-RAM. Each object has 4 bytes of attribute data associated with it, allowing for the attributes of up to 40 objects to be stored within OAM-RAM (160 bytes in size) [10, p. VRAM Sprite Attribute Table (OAM)].

Due to a hardware limitation of the PPU, only the 10 objects with the highest rendering priority are displayed at once on any given scanline. In the case where two objects with the same horizontal screen positions overlap, the object whose attributes are stored in a lower OAM-RAM position will have priority. Furthermore, if two objects with different horizontal screen positions overlap in DMG compatibility mode, the object closer to the left of the screen will have priority [10, p. VRAM Sprite Attribute Table (OAM)].

Details regarding the attributes stored within OAM-RAM can be found in the appendix: Section 10.6.9.

3.5.4 DMA

Direct Memory Access (DMA) is a hardware feature that allows certain hardware components to directly access memory without utilising the CPU, generally resulting in faster access times.

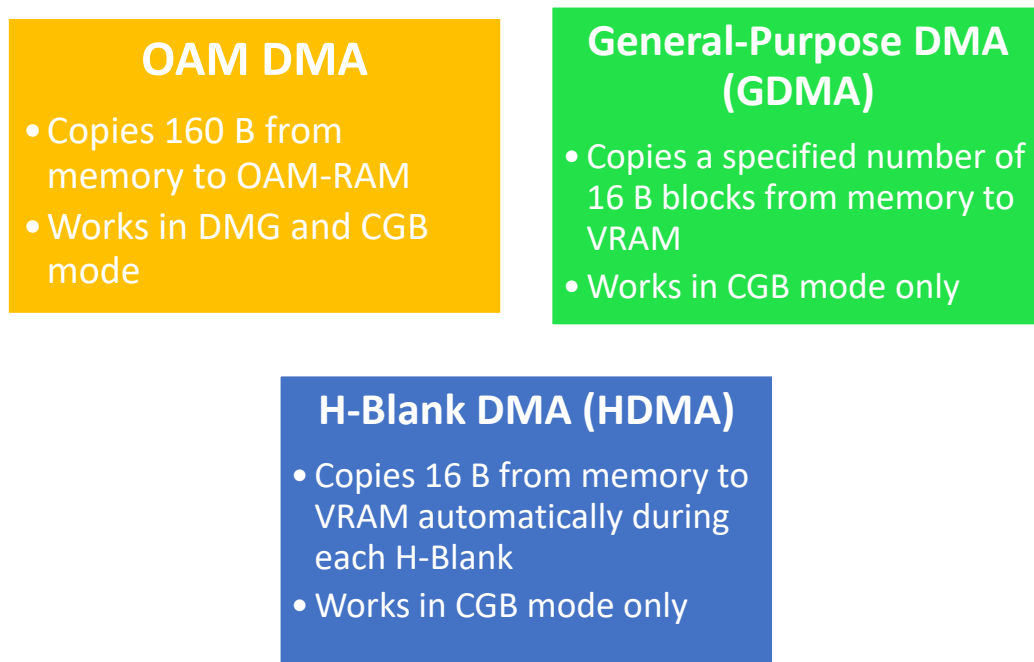


Figure 22 – A simplified diagram of the different CGB DMAs [11, p. 21].

The CGB utilises three types of DMA to enable programs to quickly copy data to VRAM or OAM-RAM from other areas of memory: OAM DMA, General-Purpose DMA (GDMA) and H-Blank DMA (HDMA).

3.5.4.1 OAM DMA

OAM DMA performs a 160-byte copy to OAM-RAM from elsewhere in memory, taking 644 clock cycles to complete [30, p. 45]. It is available for use within both CGB and DMG compatibility mode [11, p. 21].

The DMA is started by writing the most-significant 8 bits of the copy source address to the DMA I/O register (address \$FF46). This source address is restricted to increments of \$0100 for the address range \$0000 to \$DFF0 [30, p. 45].

3.5.4.2 GDMA

GDMA performs a copy of a program-specified number of 16-byte blocks of data to VRAM. The copy takes $4+32\times N$ or $4+64\times N$ clock cycles to complete with CPU double speed mode disabled and enabled respectively, where N is the number of blocks to transfer. GDMA is only available in CGB mode [11, p. 21] [30, pp. 46-47].

The HDMA1 to HDMA4 I/O registers (addresses \$FF51 to \$FF54) are used to specify the source (HDMA1-2) and destination (HDMA3-4) addresses of the transfer. The lower 4 bits of HDMA2 and HDMA4 are ignored, restricting the source and destination addresses to increments of \$0010 (16) [30, p. 46].

The source address must reside within program ROM (\$0000 to \$7FFF), cartridge RAM (\$A000 to \$BFFF) or WRAM (\$C000 to \$DFFF) [11, p. 21]. The destination address is a local address within the VRAM bank currently mapped to system memory, ranging from \$0000 to \$1FF0 [30, p. 46].

To begin the transfer, the number of 16-byte blocks to transfer is written to bits 0 to 6 of the HDMA5 I/O register (address \$FF55) with bit 7 left unset. During the copy, program execution and interrupt handling is suspended [10, p. LCD VRAM DMA Transfers (CGB only)] [30, p. 46].

3.5.4.3 HDMA

HDMA is a specialized version of GDMA that performs an automatic single 16-byte block copy to VRAM during H-Blank. The copy takes 36 or 68 clock cycles with CPU double speed mode disabled and enabled respectively. Like GDMA, HDMA is only available in CGB mode [11, p. 21] [30, p. 47].

The source and destination address for HDMA is configured exactly like GDMA with the same set of restrictions. To enable HDMA, a value must be written to the HDMA5 I/O register with bit 7 set. If HDMA is enabled, writing a value with bit 7 unset to HDMA5 will disable it [30, p. 47].

3.6 Audio Controller (APU)

The Audio Controller, also known as the Audio Processing Unit (APU), is the hardware component responsible for generating stereo sound via the Game Boy's left (SO2) and right (SO1) sound output terminals.

3.6.1 Channel Overview

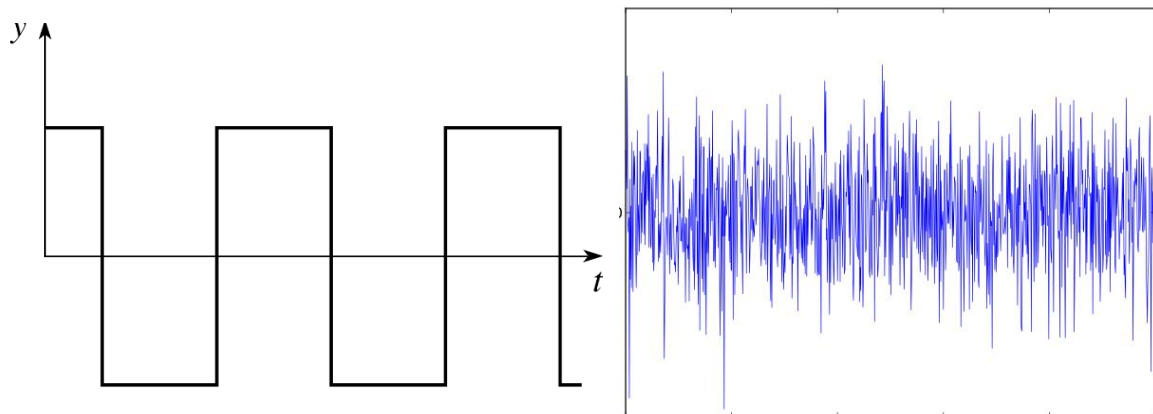


Figure 23 – A square wave [36] (left) and typical white noise wave [37] (right). The x and y-axis indicate time and amplitude respectively.

The APU can produce and mix together four different types of sound via its internal sound generation channels: [10, p. Sound Overview]

1. Square wave with envelope and sweep capabilities
2. Square wave with envelope capabilities
3. Program-defined 4-bit PCM wave samples from wave RAM
4. Pseudorandom white noise with envelope capabilities

The APU's NR50 I/O register (address \$FF24) may be used to control the output volumes of SO1 and SO2. Additionally, the NR51 I/O register (address \$FF25) may be used to individually mute each channel's DAC input to SO1 and SO2 [10, p. Sound Control Registers].

3.6.2 Volume Envelope and Frequency Sweep

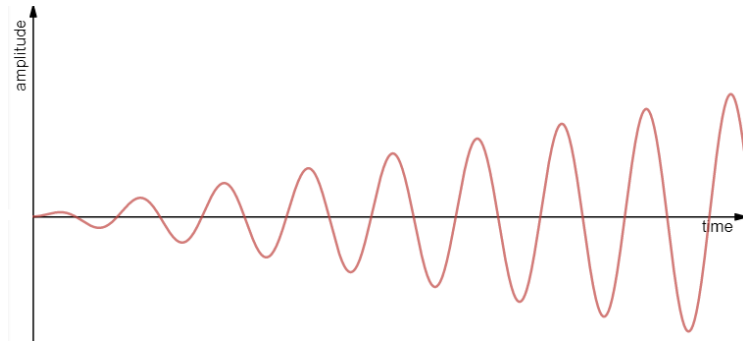


Figure 24 – An example of a sine wave with an increasing amplitude over time.

The square and noise channels (channels 1, 2 and 4) contain a volume envelope control unit that allows these channels to gradually increase or decrease their output volumes over time. This allows for fading sound effects to be handled automatically by the hardware [20, p. Overview]. The envelope effect is generally achieved by gradually increasing or decreasing the amplitude of the resulting sound as described by Figure 24.

Each channel with envelope support has an Envelope Control I/O register which may be used to change the period, initial volume and direction (increasing or decreasing) of the envelope, as well as enable or disable its operation. These registers are mapped as follows: [20, p. Registers]

- **Channel 1:** NR12 (address \$FF12)
- **Channel 2:** NR22 (address \$FF17)
- **Channel 4:** NR42 (address \$FF21)

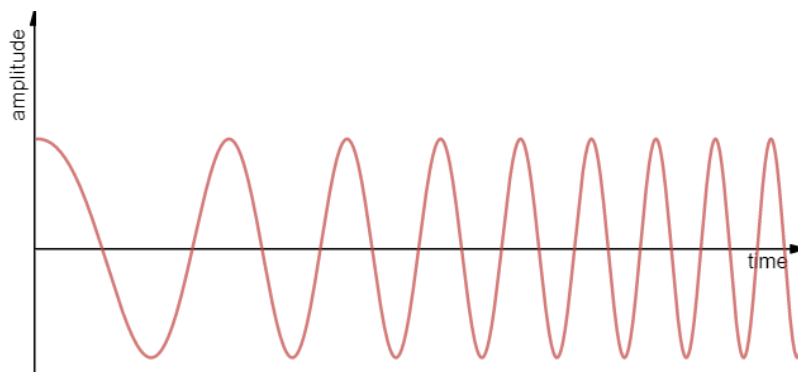


Figure 25 – An example of a cosine wave with an increasing frequency over time.

In addition to this, the first square channel (channel 1) contains a frequency sweep unit that allows the channel to gradually increase or decrease its output frequency as described by Figure 25 [20, p. Overview].

The Sweep Control (NR10: address \$FF10) I/O register may be used to change the period, duration, and direction of the sweep, as well as enable and disable it [10, pp. Sound Channel 1 - Tone & Sweep].

3.6.3 DAC, Frequency Timers, Length Counters and Triggers

All APU sound channels utilise a Digital-to-Analog Converter (DAC) to convert their outputs into a proportional voltage to be mixed and fed into the SO1 and SO2 sound output terminals.

Disabling a channel's DAC has the effect of silencing its sound output, which can be achieved by setting the initial envelope volume of the channel to zero. In the case of the wave channel (channel 3), which does not include an envelope control unit, DAC may be disabled by clearing bit 7 of the DAC Control I/O register (NR30: address \$FF1A) [20, p. Channel DAC].

Additionally, each channel includes a frequency timer with an internal counter that decrements with every CPU clock cycle until it reaches zero. When this happens, the channel updates its output waveform and reloads its frequency timer with a specified period [10, p. Timer].

Frequency timer period values for channels 1 to 3 are stored within their corresponding Frequency Period Least and Most-Significant Byte I/O registers. These are mapped as follows: [20, p. Registers]

- **Channel 1:** NR13 and NR14 (addresses \$FF13 to \$FF14)
- **Channel 2:** NR23 and NR24 (addresses \$FF18 to \$FF19)
- **Channel 3:** NR33 and NR34 (addresses \$FF1D to \$FF1E)

Every APU channel has a length counter, which may be used to automatically silence a playing channel after a certain duration [10, p. Sound Overview]. Each channel may have their length counters written to using their appropriate Length Load I/O registers. These registers are mapped as follows: [20, p. Registers]

- **Channel 1:** NR11 (address \$FF11)
- **Channel 2:** NR21 (address \$FF16)
- **Channel 3:** NR31 (address \$FF1B)
- **Channel 4:** NR41 (address \$FF20)

The length counter may be disabled by the running program at any time through clearing bit 6 of the corresponding channel's Frequency Period MSB. In the case of the noise channel (channel 4), the NR44 I/O register (address \$FF23) is used instead [20, p. Registers].

In a similar manner, channels may be restarted (triggered) by the program via setting bit 7 of the corresponding channel's Frequency Period MSB (or NR44) register [20, p. Registers]. Triggering a channel re-enables it, resets its length counter if it is zero, reloads its frequency timer and resets its current volume [20, p. Trigger Event].

3.6.4 Frame Sequencer

Clock #	0	1	2	3	4	5	6	7	Rate
Length	✓		✓		✓		✓		256 Hz
Sweep			✓				✓		128 Hz
Envelope								✓	64 Hz

Figure 26 – The operation of the frame sequencer. A tick (✓) represents the specified component being updated at that specific clock number [20, p. Frame Sequencer].

The APU frame sequencer is responsible for updating the state of the channel length counters, envelope and sweep units, and is clocked by a 512 Hz timer. These components are updated at an interval as described by Figure 26.

4 Software Overview

This section aims to provide an overview of the finished emulation software, explaining all program features available to the user. The software, called *sdgbc*, has two distinct modes of operation: each of which provides its own tailored user experience: the main GUI and command-line debugger modes. The software is available for both the Windows and Linux operating systems.

4.1 Main GUI Mode

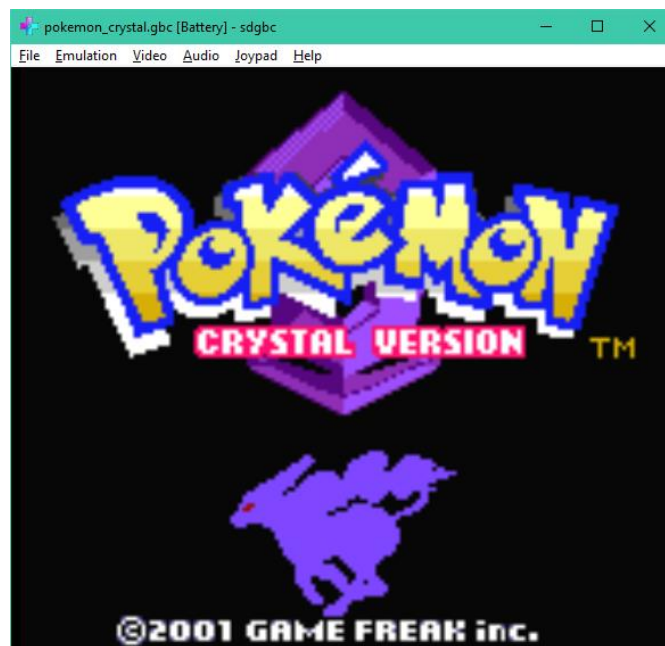


Figure 27 – A screenshot of Pokémon Crystal Version (2001) being emulated in the main GUI mode.

The main GUI mode is the default user experience provided by the software and is specifically designed for enabling user interaction with emulated programs. The GUI contains a large screen area dedicated to displaying the emulated system's live LCD video output. Additionally, input for the emulated joypad is mapped to the following keyboard keys:

Joypad Key	Keyboard Key	Joypad Key	Keyboard Key
Left	Left Arrow	A	X
Right	Right Arrow	B	Z
Up	Up Arrow	Select	Left/Right Shift
Down	Down Arrow	Start	Return

Figure 28 – A table showing the keyboard mappings for the emulated joypad keys.

The window title bar is used to convey some additional information about the emulating program, including the file name of the loaded program ROM file, whether the program utilises battery-packed cartridge RAM, whether the program is emulating in DMG compatibility mode and whether the emulation is currently paused.

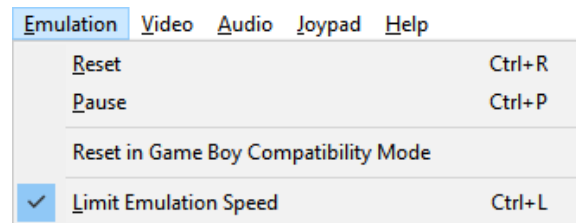


Figure 29 – The “Emulation” submenu on Windows. Mnemonics (underlined) may be accessed by pressing Alt + Letter keys. Accelerator shortcuts are shown to the right of each item (if any).

The GUI utilises its window menu bar for all additional available interactions, with support for keyboard shortcuts such as mnemonics and accelerators present as described by Figure 29. Available interactions are detailed in the following subsections.

4.1.1 Loading Programs

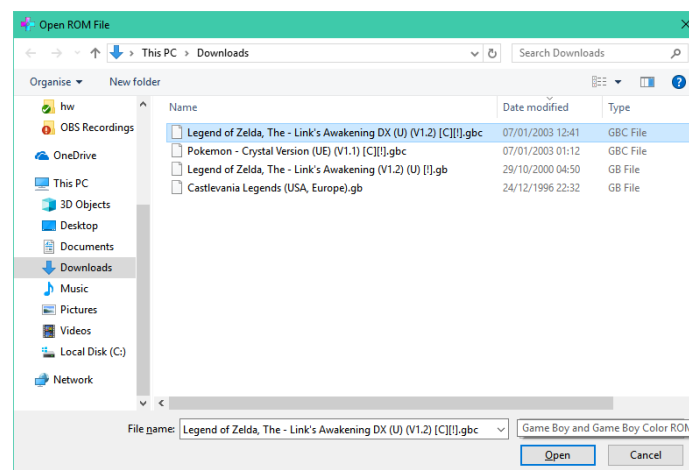


Figure 30 – The “Open ROM File” dialog on Windows.

CGB and DMG programs can be emulated from ROM files stored on the user’s file system. These files are expected to contain exact byte-for-byte dumped cartridge ROM data and will have some of their ROM header information checked for validity when loaded. These files are usually use a *.gb* or *.gbc* file extension.

In addition to being loadable via the “open file” dialog in Figure 30, ROM files may also be dragged and dropped into the main GUI window to be loaded. Successfully loading a new program for emulation will unload the loaded program (if any) and automatically reset the emulated system before its execution begins.

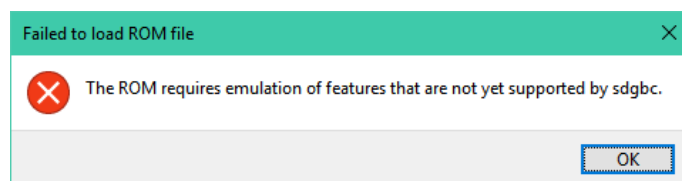


Figure 31 – The error message shown when trying to load a program requiring the emulation of an unsupported cartridge configuration.

Supported emulated cartridge configuration types include those referenced in the appendix: Section 10.1. Trying to load a program requiring the emulation of an unsupported cartridge configuration will yield the message shown in Figure 31.

4.1.2 Battery-Packed RAM Snapshots

The emulator will automatically save and load snapshots of battery-packed RAM to and from the user’s file system for programs that utilise it. This stops program data such as character saves and leader board scores from being lost after the emulating program or the emulator itself has been closed. These snapshots are exact dumps of the emulated cartridge RAM and are compatible with the *VisualBoyAdvance* .sav file format.

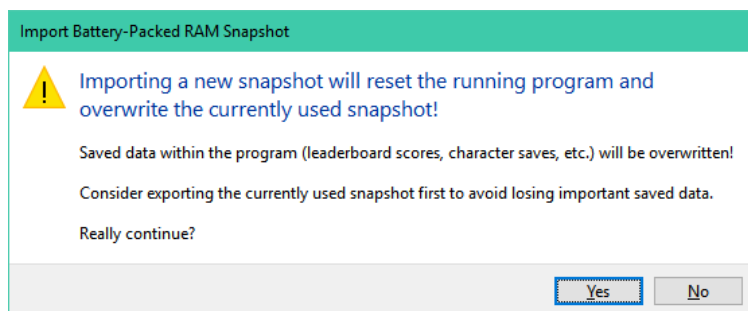


Figure 32 – The warning message shown when importing a new battery-packed RAM snapshot.

Additionally, the user may import or export battery-packed RAM snapshots from any file system location. Importing a snapshot will overwrite the snapshot currently used by the emulator for persisting data between sessions for the program and reset the emulated system.

Like loading program ROM files, snapshot files may be imported via an “open file” dialog or dragged and dropped into the main GUI window.

4.1.3 Emulation Controls

Options for controlling aspects of the emulated CGB system exist as follows (as shown in Figure 29):

- **Reset** – Performs a hardware reset on the emulated system: powering the system off and then on again. This has the effect of restarting the execution of the emulating program. The emulator will automatically decide if the program should be run in CGB or DMG compatibility mode.
- **Reset in DMG Compatibility Mode** – Performs a hardware reset and forces the emulated system to run the program in DMG compatibility mode (many CGB features will be unavailable). CGB-only programs will display a compatibility warning like in Figure 7 when executed in this mode.
- **Pause** – Pauses the state of the emulated system until emulation is subsequently un-paused again by the user. Disabled by default.
- **Limit Emulation Speed** – Limits the speed of emulation to that of the CGB (~4.2 or 8.4 MHz emulated clock speed). Enabled by default.

4.1.4 Miscellaneous Controls

In addition to the emulation controls described, the following miscellaneous controls also exist:

- **Show PPU Graphics Layers** – May be used to show or hide the emulated PPU’s background, window or object graphics layers. All layers are enabled by default.
- **Emulate PPU Scanline Object Limit** – Enables the emulation of the PPU’s 10 objects per scanline limitation. Enabled by default.
- **Maintain LCD Aspect Ratio** – Maintains the Game Boy’s 10:9 aspect ratio for the emulated LCD video output, regardless of the size of the window. Enabled by default.
- **Smoothen LCD Video Output** – Applies a smooth filter to the emulated LCD video output to make the individual pixels less noticeable. Enabled by

default.

- **Mute Audio** – Mutes all audio output when enabled. Disabled by default.
- **Mute APU Channels** – May be used to mute the individual DAC outputs of any of the four emulated APU channels. All channels are enabled by default.
- **Allow Impossible Joypad Inputs** – Allows impossible directional pad key combinations on the emulated joypad as described in Section 3.4. Disabled by default.
- **Show Joypad Controls** – Opens a dialog window showing the emulated joypad keyboard mappings as described in Figure 28.
- **Show About Dialog** – Opens a window showing general information about the program and the author.

4.2 Command-Line Debugger Mode

```
(running, PC=$3778, (PC)=[$1b $7a $b3]) > print
stat: running
cycl: 737820
step: 123456

regs: PC=$3778 SP=$dffc
      AF=$d700 BC=$0000 DE=$13c5 HL=$406b
flag: - - - -
inme: on

memory around PC:
=====
$3750 | 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f | 0123456789abcdef
-----
$3750 | cb 42 3e 10 20 02 3e 20 e2 3e 30 e2 cb 1a 1d 20 | .B>. .> .>0....
$3760 | ef 05 20 e8 3e 20 e2 3e 30 e2 c1 05 c8 cd 72 37 | .. .> .>0.....r7
$3770 | 18 d1 11 58 1b 00 00 00>1b 7a b3 20 f8 c9 f3 d5 | ...X.....Z. ....
$3780 | cd 0c 39 3e e4 e0 47 11 00 88 01 00 10 cd b9 de | ..9>..G.....
$3790 | 21 00 98 11 0c 00 3e 80 0e 0d 06 14 22 3c 05 20 | !.....>....."<.

memory around SP:
=====
$dfd0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
$dfe0 | 00 00 00 00 00 00 00 00 00 00 7f 0a 7f 0a a5 00 | .....
$dff0 | 73 00 6b 40 7a 17 00 00 00 7f 79 37>4e 40 a1 01 | s.k@z.....y7N@..
$e000 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
$e010 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
(running, PC=$3778, (PC)=[$1b $7a $b3]) > 
```

Figure 33 – The command-line debugger running the “print” command.

The command-line debugger mode is designed to aid in the testing process and can be accessed by providing the `--cpu-cmd` command-line argument to the program. It provides simple functionality for manually stepping the emulated CPU through the execution of the running program, while outputting information about its current

state. Furthermore, output produced by emulator test programs in this mode are automatically forwarded to the standard output stream.

The following commands are available for use in this mode:

- **help** – Prints a list of available commands (as shown here).
- **quit** – Exits the application.
- **load** – Loads a program ROM from a given file path. Resets the emulated system once loaded.
- **step** – Commands the emulated CPU to execute a specified amount of program instructions.
- **print** – Prints status information regarding the emulated CPU's state. This includes the values of registers, the amount of elapsed clock cycles and the contents of memory around the locations pointed to by the Program Counter and Stack Pointer.
- **memory** – Prints the contents of memory at the specified address. Values of memory locations are represented in both hexadecimal and ASCII.
- **reset** – Performs a hardware reset on the emulated system.
- **unhalt** – Forces the emulated CPU to return from a halted state (invoked via the HALT or STOP instructions) and resume executing instructions.
- **autoresume** – Toggles the debugger's ability to automatically resume the CPU from a halted state. This feature is disabled by default.
- **stepgbc** – Toggles the debugger's ability to update the state of the whole emulated CGB system while stepping or just the state of the CPU. By default, the state of the whole emulated system is updated.

5 Implementation

All the emulation software code was written in C++ using some features introduced in the C++11 and 14 language standards. A third-party tool called CMake was used as a build system for generating Visual Studio project files and Unix Makefiles for compiling the software on the Windows and Linux operating systems respectively.

This section will focus on the implementation of the most important aspects of the software's back-end emulation and the front-end GUI mode functionality, including a simplified class diagram for each section. A class diagram generated by Visual Studio including all classes (showing only class inheritance relationships) can be found in the appendix: Section 10.7.

5.1.1 Class Diagram

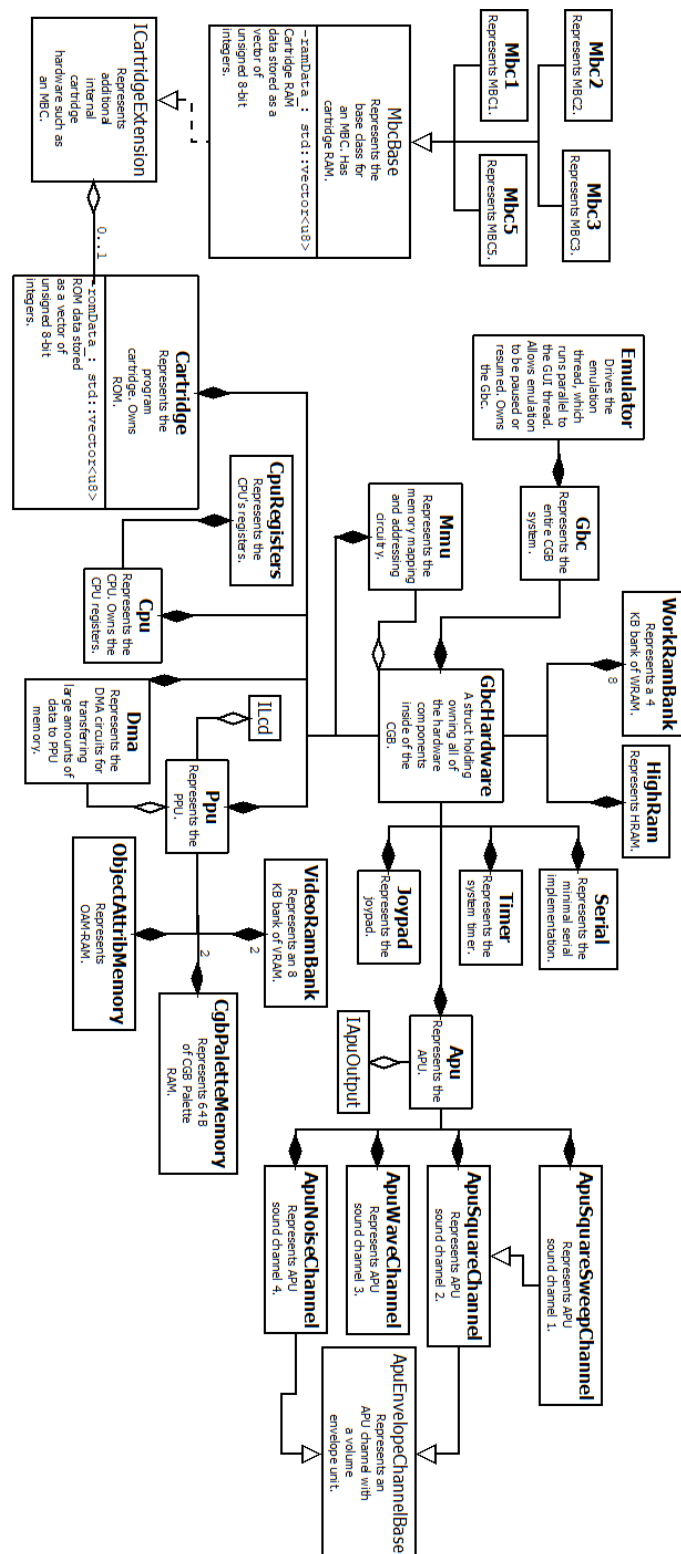


Figure 34 – A simplified class diagram of the software’s emulation back-end. Less important classes and associations are omitted for readability. The purpose of *IApuOutput* and *ILcd* is explained in Figure 48.

5.1.2 Emulation

Each emulated system hardware component is represented as its own class, typically defining public methods for updating its state and for performing hardware resets, which is done using its `Update` and `Reset` methods respectively. Additionally, accessor and mutator methods are frequently defined by these classes to expose emulated I/O registers, which commonly utilise bitmasks to ensure that unused or unmapped register bits cannot be modified. Because some hardware components behave differently in DMG compatibility mode, many classes maintain a `cgbMode` Boolean flag that is typically received as a parameter to the `Reset` method of the class.

The `Gbc` class, used to represent the whole emulated CGB system, creates and exposes an instance of the `GbcSystem` structure, which is used as a container for holding the object instances of the emulated hardware classes. In a similar manner to other emulation classes, the `Gbc` class also provides public methods for updating and resetting all of the system's emulated hardware components.

```
void Emulator::SetJoypadKeyState(JoypadKey key, bool pressed) {  
    std::unique_lock<std::mutex> lock(emulationMutex_);  
    gbc_.GetHardware().joypad.SetKeyState(key, pressed);  
}
```

Figure 35 – Code within the Emulator's SetJoypadKeyState method, which utilises a scoped mutex lock.

The `Emulator` class owns an instance of the `Gbc` that it calls `Update` on continuously from a separate emulation thread. Numerous public methods for interacting with the underlying `Gbc` in a thread-safe manner are provided by the `Emulator`, utilising C++ `std::mutex`s and `std::unique_lock`s to create scoped mutex locks that are active for the lifetimes of the critical sections within these functions.

To emulate the clock speed of the CGB CPU, the `Emulator` continuously updates the state of the `Gbc` for 70,224 clock cycles (or 140,448 cycles in CPU double speed mode) in what is referred to as a single frame of emulation. The emulation frame rate is capped at 59.7275 frames per second as to synchronize with the PPU's V-Blank refresh rate, which is achieved by sleeping the emulation thread for a short period after each frame. Occasionally, the emulation thread will “frame skip”, whereby it will process multiple frames before sleeping to meet the target frame rate if the emulation is running too slowly.

5.1.3 Memory

Each emulated memory module is represented as an array of unsigned 8-bit integers in the form of a C++ `std::array` for fixed-size arrays or `std::vector` for dynamically sized arrays. Accesses to memory are simply achieved by writing to or reading from a specified index within the array. Using this approach, classes such as `WorkRamBank` are simply type aliases for `std::array`s of different sizes, with `std::vector` mainly used for storing program ROM and cartridge RAM (whose size is only known at runtime after the appropriate ROM file has been loaded).

```
void Mmu::Write8(u16 loc, u8 val) {
    if (loc < 0x4000) {
        // cartridge ROM bank 0
        hw_.cartridge.RomBank0Write8(loc, val);
    } else if (loc < 0x8000) {
        // cartridge switchable ROM bank 0-N
        hw_.cartridge.RomBankXWrite8(loc - 0x4000, val);
    } else if (loc < 0xa000) {
        // VRAM switchable bank 0-1
        hw_.ppu.VramWrite8(loc - 0x8000, val);
    } else if (loc < 0xc000) {
        // external cartridge RAM
        hw_.cartridge.RamWrite8(loc - 0xa000, val);
    } else if (loc < 0xd000) {
        // WRAM fixed bank 0
        hw_.wramBanks[0][loc & 0xffff] = val;
    } else if (loc < 0xe000) {
        // WRAM switchable bank 1-7
        hw_.wramBanks[GetWramBankIndex()][loc & 0xffff] = val;
    } else if (loc < 0xfe00) {
```

Figure 36 – Code within the Mmu’s Write8 method, which is used to write an 8-bit value to a specified memory location.

The `Mmu` class contains methods for writing to and reading from the emulated system memory, automatically handling the translation of mapped memory addresses as described in Figure 5. Determining the mapped memory module to access from its address is done using a simple chain of if-else statements which tests against each possible mapped memory range as shown in Figure 36.

The `Cartridge` class is responsible for loading program ROM files, parsing their headers and making the loaded program available to the emulated system. If the loaded program requires the emulation of additional cartridge hardware such as an MBC or cartridge RAM, the `Cartridge` will automatically create the appropriate object implementing the `ICartridgeExtension` interface for emulating the needed components.


```

void Mbc2::ExtRomBank0Write8(u16 loc, u8 val) {
    if (loc < 0x2000) {
        if (!(util::GetHi8(loc) & 1)) {
            ramEnabled_ = !ramEnabled_;
        }
    } else {
        romBankNum_ = std::max<u16>(val & 0xf, 1);
    }
}

```

Figure 37 – Code intercepting writes to the cartridge ROM area. MBC2 uses such writes to configure the mapped ROM bank and enable accesses to cartridge RAM [38].

Classes implementing ICartridgeExtension define numerous methods for intercepting reads and writes to the cartridge ROM and RAM areas, which in the case of MBCs, allows these classes to easily implement their appropriate bank switching functionalities. The emulator implements Mbc1, Mbc2, Mbc3 and Mbc5 classes for supporting the most popular MBC cartridge configurations.

5.1.4 CPU

```

unsigned int Gbc::Update() {
    const auto cycles = hw_.cpu.Update();

    hw_.dma.Update(cycles);
    hw_.apu.Update(cycles);
    hw_.ppu.Update(cycles);
    hw_.timer.Update(cycles);
    hw_.serial.Update(cycles);

    return cycles;
}

```

Figure 38 – Code from the Gbc's Update method, which shows how the Cpu is responsible for telling other components how long to Update for.

The Cpu class handles the emulation of the CPU and is also responsible for determining how many clock cycles other emulated components should run for with each call to the Gbc's Update method.

The Cpu's Update method is responsible for driving the instruction FDE cycle and for handling any raised interrupts. It is implemented using the principles outlined in the following algorithm:

<u>Variables:</u>
pc = 16-bit unsigned integer value of the Program Counter (PC).
<u>Pseudocode:</u>

```

function Cpu::Update() {
    if (CPU is not hung) {
        HandleInterrupts()

        if (CPU is not halted or stopped) {
            // The postfix increment operator (++) increments the PC, but
            // returns the previous value.
            ExecuteOp(MmuRead8(pc++))
            return clock cycles spent executing the instruction
        }
    }

    // a NOP takes 4 clock cycles to execute on the Game Boy [28].
    return clock cycles spent performing a no-operation (NOP)
}

```

Figure 39 – Pseudocode of a simplified version of the Cpu's Update method.

<pre> bool Cpu::ExecuteOp(u8 op) { // standard instructions table switch (op) { NOP(0x00); // NOP OP(0x01, ExecLoad(reg_.bc, IoPcReadNext16())); // LD BC,nn OP(0x02, ExecLoad(reg_.bc.Get(), reg_.a.Get())); // LD (BC),A OP(0x03, ExecInc16(reg_.bc)); // INC BC OP(0x04, ExecInc(reg_.b)); // INC B OP(0x05, ExecDec(reg_.b)); // DEC B OP(0x06, ExecLoad(reg_.b, IoPcReadNext8())); // LD B,n OP(0x07, ExecOpRlca0x07()); // RLCA } } </pre>
Opcode Decoder and Executor
<pre> template <typename H, typename L> void Cpu::ExecAdd16(CpuRegister8Pair<H, L>& destRegPair, u16 val) { const uint_fast32_t result = destRegPair.Get() + val; InternalDelay(); reg_.f.SetNFlag(false); reg_.f.SetCFlag(result > 0xffff); reg_.f.SetHFlag((destRegPair.Get() & 0xffff) + (val & 0xffff) > 0xffff); destRegPair.Set(result & 0xffff); } </pre>
Example Instruction Handling Method (16-bit ADD)

Figure 40 – Code from the Cpu's ExecuteOp (top) and ExecAdd16 (bottom) methods.

The Cpu's ExecuteOp method is used to decode the next opcode in memory, fetch its operands and execute the associated program instruction. It is primarily implemented using a switch statement, in which each possible opcode value has a C pre-processor macro defining a case label mapping it to its appropriate handler method at compile-time.

```

42      // standard instructions table
43      switch (op) {
0x000000000004114f2 <+18>:    movzbl  -0x1c(%rbp),%eax
0x000000000004114f6 <+22>:    cmp      $0xff,%eax
0x000000000004114fb <+27>:    ja       0x413ad6 <Cpu::ExecuteOp(unsigned char)+9718>
0x00000000000411501 <+33>:    mov      %eax,%eax
0x00000000000411503 <+35>:    mov      0x4278e0(,%rax,8),%rax
0x0000000000041150b <+43>:    jmpq     *%rax

```

Figure 41 – Screenshot of the disassembly of the Cpu’s ExecuteOp method, which shows an x86 relative jump instruction (jmpq) being used.

It is important to note that the implementation of the opcode decoder differs from that of the real Game Boy CPU’s; for example, the Game Boy CPU usually inspects the value of the three least-significant bits of the opcode to determine the Register addressing mode to use for the BIT, SET and RES instructions [11, pp. 103-104]. By using a switch statement, a modern C++ compiler is easily able to generate a jump-table that is very performant to execute on modern hardware as shown in Figure 41 [39].

```

template <typename C, typename T>
class CpuRegister {
    static_assert(std::is_integral<T>::value, "T must be an integral type");

public:
    void Set(T val);
    T Get() const;

    // prefix inc/dec operators
    C& operator++();
    C& operator--();

    // postfix inc/dec operators
    T operator++(int);
    T operator--(int);

private:
    C& AsC();
    const C& AsC() const;
};

template <typename T>
class CpuBasicRegister : public CpuRegister<CpuBasicRegister<T>, T> {
    static_assert(std::is_integral<T>::value, "T must be an integral type");

public:
    explicit CpuBasicRegister(T val = 0);

    void SetImpl(T val);
    T GetImpl() const;

private:
    T val_;
};

```

Figure 42 – Code showing the base class definitions for the emulated CPU registers, which utilise the CRTP idiom.

Due to the mutability restrictions of the bits within the CPU’s status flag register compared to other CPU registers, polymorphism was incorporated in the

CpuRegister classes to provide the convenience of allowing calling code to access all CpuRegisters in the same manner, regardless of differences in behaviour.

Specifically, the CpuRegister classes implement a C++ idiom known as the Curiously Recurring Template Pattern (CRTP) to implement static polymorphism [40]. In the emulator's codebase, the concrete type of each CpuRegister is known by the calling code at compile-time; because of this, static polymorphism was used over the (standard) dynamic polymorphism approach, as there was no need to utilise any of the runtime virtual method table lookup features C++ incorporates when using dynamic polymorphism [41].

5.1.5 Joypad

The Joypad class is responsible for the emulation of the system joypad and contains methods for updating the state of joypad keypresses. The state of all joypad keys is internally stored as an 8-bit value, with each bit representing whether a specific key is currently pressed. The GetJoyp method is called for every read to the JOYP I/O register and is responsible for converting the internal representation used for storing the joypad key states into the format expected by the emulated program as described in Section 3.4.

```
enum class JoypadKey : u8 {  
    A      = 0x01,  
    B      = 0x02,  
    Select = 0x04,  
    Start  = 0x08,  
    Right  = 0x10,  
    Left   = 0x20,  
    Up     = 0x40,  
    Down   = 0x80  
};
```

Figure 43 – The enumeration used by the Joypad for uniquely identifying keypresses.

The JoypadKey enumeration defines a named set of bitmasks for all joypad keys to be given as an argument to the Joypad's SetKeyState method, which buffers the state of the keypresses until the CommitKeyStates method is called.

Once every frame, the Emulator calls CommitKeyStates, which updates the key states of the joypad from their buffered states and handles the request of the joypad interrupt if any selected keys were newly pressed since the last call to CommitKeyStates.

5.1.6 System Timer

The Timer class is responsible for emulating the system timer. The Timer is driven by calls to its Update method, which is used to refresh the state of the timer for however many clock cycles have elapsed from the last call to the Cpu's Update method.

```
// calculated as (non-double speed frequency in Hz) / (timer frequency in Hz)
enum TimerFreqInCycles : unsigned int {
    kTimerFreq262144HzCycles = 16,
    kTimerFreq65536HzCycles  = 64,
    kTimerFreq16384HzCycles  = 256,
    kTimerFreq4096HzCycles   = 1024
};
```

Figure 44 – The constants used by the Timer for maintaining its different ticking frequencies.

The four base timer ticking frequencies described in Section 3.3 are represented as named constants specifying the amount of clock cycles to wait between each tick. These constants are used within the Update method to regulate the increment ticking frequency of the DIV and TIMA I/O registers as follows:

Variables:

cycles = unsigned integer amount of clock cycles elapsed.
div, tima, tma = 8-bit unsigned integer values of DIV, TIMA and TMA.
tac = 8-bit integer value of TAC.

divCycles, timaCycles = unsigned integers counting up the number of clock cycles until the next increment tick for DIV and TIMA.

timaFrequencyCycles = the amount of clock cycles delay between each TIMA increment tick as configured by bits 0 to 1 of TAC.

kTimerFreq16384HzCycles = 256 clock cycles as described in Figure 44.

Pseudocode:

```
function Timer::Update(cycles) {
    divCycles += cycles

    div      += divCycles / kTimerFreq16384HzCycles
    divCycles = divCycles mod kTimerFreq16384HzCycles

    if (bit 2 set in tac) {
        timaCycles += cycles
    }
}
```

```

newTima = tima + (timaCycles / timaFrequencyCycles)
if (newTima > 255) {
    // Value of TIMA is about to rollover, reset it to TMA and
    // interrupt. The use of mod will add any remaining cycles
    // after the rollover reset happens (if any).
    tima = newTima mod (256 - tma)
    CpuRequestTimerInterrupt()
} else {
    tima = newTima
}

timaCycles = timaCycles mod timaFrequencyCycles
}
}

```

Figure 45 – Pseudocode of the algorithm implemented by the Timer's Update method.

5.1.7 PPU

The Ppu class is responsible for the emulation of the CGB PPU. It owns instances of the VideoRamBank, ObjectAttribMemory and CgbPaletteMemory classes for representing VRAM, OAM-RAM and CGB Palette RAM respectively. The Ppu maintains a pointer to an object implementing the ILcd interface received by the Ppu's SetLcd method, which allows the Ppu to interact with the software's front-end systems to display the emulated LCD video output.

The Ppu's UpdateScreenMode method, frequently invoked by the Ppu's Update method, automatically handles the transitions between the emulated PPU's four different rendering modes. To reduce the complexity of the Ppu's implementation, scanlines are rendered (using the RenderScanline method) during the Ppu's transition from mode 3 to H-Blank (mode 0); because of this, graphical effects relying on mid-scanline palette modifications (during modes 2 and 3) will not render as intended.

When invoked, the RenderScanline method calls the RenderBufferBgScanline, RenderBufferWindowBgScanline and RenderBufferSpriteScanline methods, which buffer intermediary scanline pixel data necessary for rendering the PPU's background, window and object layers respectively. From this buffered data, the RenderScanline method translates the PPU palette colour values of each scanline pixel into RGB colours to be rendered, sending them to the graphics driver via invocations of the ILcd LcdPutPixel method.

DMA transfers to PPU memory are emulated by the Dma class, which holds a reference to the Ppu class for accessing the Ppu's VideoRamBanks and ObjectAttribMemory. To reduce the complexity of the Dma's implementation,

emulated DMA copies implement an artificial transfer delay when requested, performing the actual transfer after the delay has finished.

5.1.8 APU

The `Apu` class handles the emulation of the APU and contains instances of the `ApuSquareSweepChannel`, `ApuSquareChannel`, `ApuWaveChannel` and `ApuNoiseChannel` classes for emulating the APU's four sound generation channels. The `Apu` maintains a pointer to an object implementing the `IApuOutput` interface given to the `Apu`'s `SetApuOutput` method, which enables the emulated APU's sound output to be played by the software's front-end systems.

The `Apu`'s `Update` method is responsible for calling `UpdateFrameSequencerCycle` to emulate the APU's frame sequencer and for ticking channel frequency timers by calling each channel's `UpdateFrequencyTimer` method. The `UpdateFrameSequencerCycle` method periodically calls the appropriate `UpdateLengthCounter`, `UpdateEnvelope` and `UpdateSweep` methods of each channel, updating the state of their emulated length counters, volume envelope and frequency sweep units.

Additionally, the emulated DAC outputs of the four channels, retrieved using each channel's `CalculateDacOutputVolume` method, are periodically mixed together using the `Apu`'s `MixChannels` method, which produces 16-bit signed PCM samples for the left and right stereo channels to be sent to the audio driver using the `IApuOutput` `AudioBufferSamples` method. These stereo samples are produced at an interval necessary for producing audio at a 44,100 Hz sample rate, which is widely supported by modern sound cards.

```
constexpr std::array<u8, 4> kSquareDutyWaveforms {
    0b00000001,
    0b10000001,
    0b10000111,
    0b01111110
};
```

Figure 46 – An array of 8-bit values used by the `ApuSquareChannel` to produce different types of square waves [20, p. Square Wave].

The `ApuSquareChannels` generate square waves from waveforms encoded as 8-bit values as shown in Figure 46, producing silence whenever an unset bit of the waveform is selected: this selection is moved to the next bit of higher significance (the next bit to the left) when the value of the channel's emulated frequency timer reaches zero.

The `ApuNoiseChannel` generates white noise output primarily through the emulation of a 15-bit Linear-Feedback Shift Register (LFSR), producing silence whenever bit 0 of the LFSR is set. The value of the LFSR is updated when the value of the `ApuNoiseChannel`'s emulated frequency timer reaches zero using the following algorithm:

<p style="text-align: center;"><u>Variables:</u></p> <p>nr43 = the 8-bit unsigned integer value in NR43. xorBit = an 8-bit unsigned integer.</p> <p>lfsr = a 15-bit unsigned integer (the value of the linear-feedback shift register).</p>
<p style="text-align: center;"><u>Pseudocode:</u></p> <pre>function ApuNoiseChannel::UpdateLfsr() { // >> and << are right and left bitshifts respectively. xorBit = (lfsr and 1) xor ((lfsr >> 1) and 1) lfsr = (lfsr >> 1) or (xorBit << 14) if (bit 3 set in nr43) { lfsr = (lfsr and 191) or (xorBit << 6) } }</pre>

Figure 47 – Pseudocode for calculating the value of the APU Noise Channel's LFSR used in pseudorandom noise generation. [20, p. Noise Channel]

5.2 GUI Front-End

5.2.1 Class Diagram

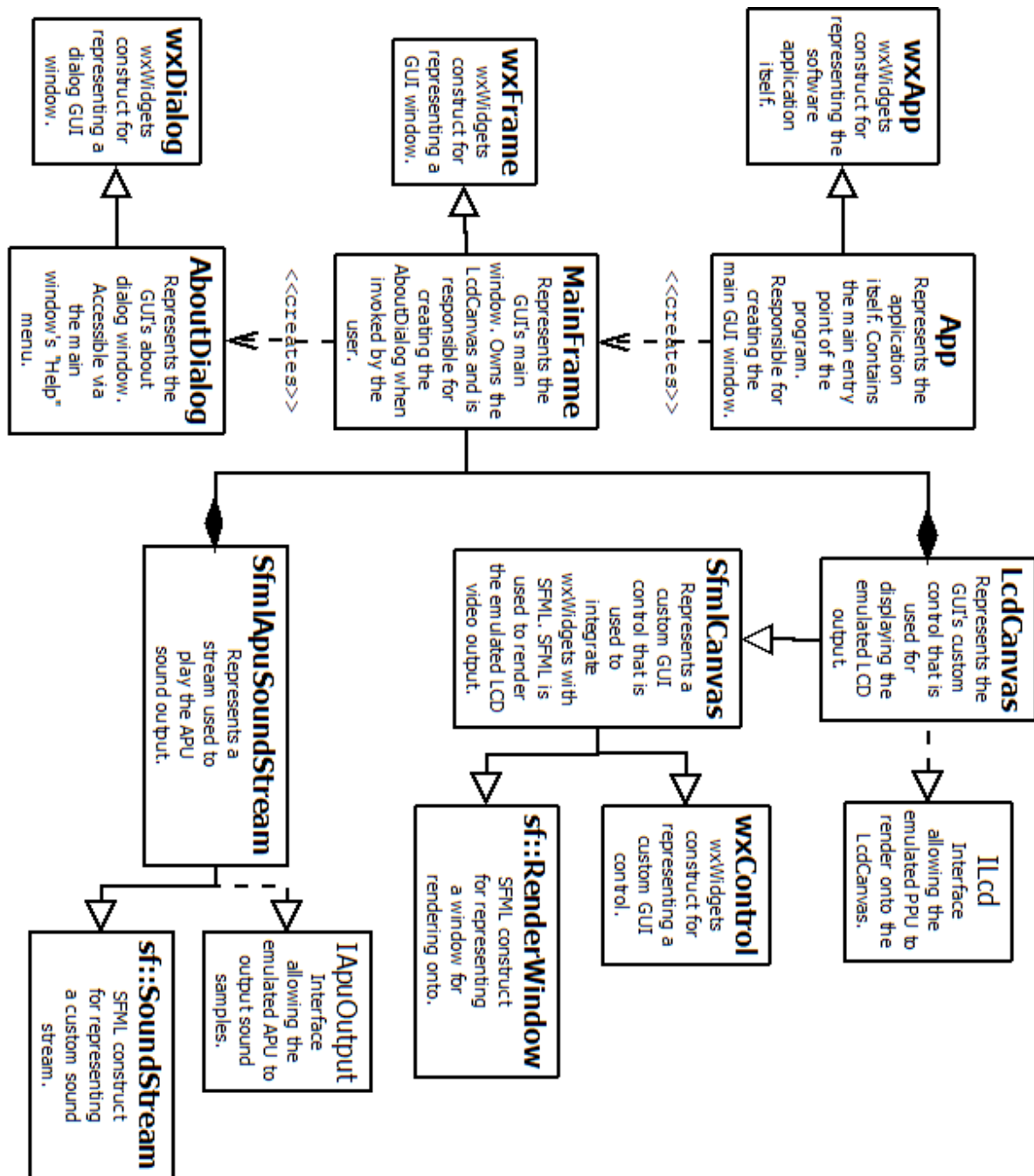


Figure 48 – A simplified class diagram of the software's GUI front-end. Less important classes and associations are omitted for readability.

5.2.2 General Implementation

To implement the software's GUI, a cross-platform third-party library called wxWidgets was used, which provides C++ classes such as `wxFrame` for creating windows, `wxControl` for creating custom window controls and `wxDialog` (which inherits from `wxFrame`) for creating dialog boxes. In addition to wxWidgets, a cross-platform third-party library called SFML was used for playing audio and performing hardware-accelerated graphics rendering.

The `MainFrame` class, which derives from `wxFrame`, is used to implement the main GUI window shown in Figure 27. Upon its instantiation, the `MainFrame` creates `wxMenuBar` and `wxMenu` objects for defining the contents of the window's menu bar and creates an instance of the `LcdCanvas wxControl` and `SfmlApuSoundStream` classes, which are respectively used to display the live video contents of the emulated LCD and play the emulated APU's live sound output.

```
wxBEGIN_EVENT_TABLE(SfmlCanvas, wxControl)
    EVT_IDLE(SfmlCanvas::OnIdle)
    EVT_PAINT(SfmlCanvas::OnPaint)
    EVT_SIZE(SfmlCanvas::OnSize)
    EVT_ERASE_BACKGROUND(SfmlCanvas::OnEraseBackground)
    EVT_SET_FOCUS(SfmlCanvas::OnSetFocus)
    EVT_KILL_FOCUS(SfmlCanvas::OnKillFocus)
wxEND_EVENT_TABLE()
```

Figure 49 – Code within the `SfmlCanvas` class used to define its wxWidgets event table.

Classes that derive from `wxWindow`, such as `wxFrame` and `wxControl`, may define event tables like in Figure 49 that can be used to listen for certain GUI events such as mouse clicks and keyboard keystrokes. This feature is most notably used in the implementation of emulated joypad interactions using the keyboard, which is achieved by defining event listeners that call the `HandleJoypadKeyEvent` method defined within the `MainFrame` class when a key is pressed or released.

Variables:

frontBuffer = pointer to the buffer currently being presented.
backBuffer = pointer to the other buffer to now present.
temp = a temporary pointer to a buffer.

Pseudocode:

```
function SwapBuffers() {
    temp = frontBuffer
    frontBuffer = backBuffer
    backBuffer = temp
}
```

Figure 50 – Pseudocode for swapping the front and back buffers in a double buffering implementation.

The `SfmlApuSoundStream` and `LcdCanvas` classes implement double buffering, which allows the emulator to update the state of audio and video while audio and video is currently being actively presented to the user. For example, this prevents partially rendered frames of video from being shown on the `LcdCanvas` by rendering instead to a hidden image that is then swapped with the shown image once the frame is finished.

6 Testing

Testing the emulation software involved the frequent use of automated third-party emulator test programs and the software's own command-line debugger mode during development. Additionally, many real games released for the CGB and DMG were also tested for compatibility issues.

Testing was performed throughout the duration of the project, with periods of extensive testing (usually lasting for around a week) performed before important project deadlines such as the prototype demonstration, interview and final software submission.

All tests were grouped into three categories: critical hardware emulation tests, miscellaneous hardware emulation tests and manual program compatibility tests.

6.1 Critical Hardware Emulation Tests

```
cpu_instrs
01:ok    02:ok    03:ok
04:ok    05:ok    06:ok
07:ok    08:ok    09:ok
10:ok    11:ok
Passed all tests
```

Figure 51 – Example LCD output from Blargg's cpu_instrs test program [5].

Critical hardware emulation tests are automated unit tests that verify the correctness of several aspects of the software's hardware emulation. All tests within this category are expected to unconditionally pass; a failure in any of these tests indicates an issue that will greatly lower emulated program compatibility. As a result, many of these tests specifically verify the functionality of the emulated CPU.

Tests within this category include some of the following:

- **Blargg's cpu_instrs tests** – Tests the emulation of CPU instructions and interrupts. This is composed using the following 11 important unit tests:
 - **01-special** – Tests some aspects of the JR, LD PC, HL, POP AF and DAA instructions.
 - **02-interrupts** – Tests the operation of the EI and DI instructions. Also verifies that the CPU can resume from a HALT via the use of a

raised system timer interrupt.

- **03-op sp,hl** – Tests the operation of instructions which modify the Stack Pointer or HL 16-bit CPU register pair.
- **04-op r,imm** – Tests the operation of instructions that modify CPU registers via an Immediate operand.
- **05-op rp** – Tests the operation of instructions that perform arithmetic operations on 16-bit CPU register pairs.
- **06-ld r,r** – Tests the operation of the 8-bit LD instructions.
- **07-jr,jp,call,ret,rst** – Tests the operation of the branch instructions and subroutine calls and returns.
- **08-misc instrs** – Tests the operation of certain LDH, LD, PUSH and POP instructions.
- **09-op r,r** – Tests the operation of most instructions which act on 8-bit CPU registers. Also tests the NOP instruction.
- **10-bit ops** – Tests the operation of the BIT, RES and SET instructions.
- **11-op a,(hl)** – Tests the operation of instructions which read from or write to memory at locations pointed to by 16-bit CPU register pairs.
- **Blargg's instr_timing test** – Tests the amount of clock cycles that each CPU instruction takes to execute.
- **Blargg's interrupt_time test** – Tests the amount of clock cycles used in the interrupt handling process.

The results of these tests can be found in the appendix: Sections 10.8.1 and 10.8.2.

6.2 Miscellaneous Hardware Emulation Tests

Miscellaneous hardware emulation tests are mostly automated tests that verify the emulation of non-critical or obscure hardware features. Passing tests within this

category was not a huge priority during development, as these features typically have little effect on emulated program compatibility.

Tests within this category include some of the following:

- **Blargg's cgb_sound tests** – Tests the emulation of the CGB's APU, including many obscure hardware interactions and bugs. This is composed as 12 individual unit tests.
- **Mooneye GB sprite_priority test** [42]– Tests the emulation of PPU object rendering priorities. This test requires some manual inspection of the output, as the system itself is unable to verify that the test passed.

The results of these tests can be found in the appendix: Sections 10.8.3 and 10.8.4.

6.3 Manual Program Compatibility Tests

In addition to these tests, many real games released for both the CGB and DMG were manually tested for compatibility issues. The program ROMs for these games were previously dumped directly from their respective physical game cartridges using a *Gen3 Reader/Writer* [43].

The full list of games tested include:

- | | |
|----------------------------------|--|
| • <i>Cannon Fodder</i> | • <i>Super Mario Land 2 – 6 Golden Coins</i> |
| • <i>Castlevania Legends</i> | • <i>Tetris</i> |
| • <i>Donkey Kong Country</i> | • <i>Tetris DX</i> |
| • <i>Donkey Kong Land III</i> | • <i>The Legend of Zelda – Link's Awakening</i> |
| • <i>Ghosts 'n Goblins</i> | • <i>The Legend of Zelda – Link's Awakening DX</i> |
| • <i>Jurassic Park</i> | • <i>The Legend of Zelda – Oracle of Seasons</i> |
| • <i>Kirby's Dream Land</i> | • <i>Toy Story Racer</i> |
| • <i>Kirby's Dream Land 2</i> | |
| • <i>Mega Man Xtreme</i> | |
| • <i>Nintendo Baseball</i> | |
| • <i>Perfect Dark</i> | |
| • <i>Pokémon Crystal Version</i> | |
| • <i>Pokémon Yellow Version</i> | |
| • <i>Prehistorik Man</i> | |
| • <i>Super Mario Land</i> | |

The results of these tests can be found in the appendix: Section 10.8.5.

7 Critical Appraisal

7.1 Critical Analysis

In this section, the final state of the project will be critically compared against the original list of objectives defined during the project’s planning phase. These objectives were organized into two different categories: key objectives and optional objectives.

7.1.1 Key Objectives

The term “key objectives” is used to refer to the project’s most important objectives. All project requirements derived from these objectives were assigned the highest possible priority for completion.

7.1.1.1 Emulation of Key Hardware Components

This objective refers to the successful emulation of the CGB’s hardware components necessary in the execution of most programs developed for the system. This includes the emulation of the CGB’s CPU, PPU, APU, joypad and various memory modules.

The final emulation software can emulate all these key hardware components to a good level of accuracy as evidenced by the test results outlined in the appendix: Section 10.8. Most importantly, the emulator can play all 22 of the CGB and DMG games I own with only minor compatibility issues in 4 of them.

Furthermore, although not considered a key hardware component in the original project aims, the CGB’s serial I/O functionality was implemented to a small extent. This was done to stop certain games that rely on serial I/O features from exhibiting issues during emulation, allowing them to gracefully timeout any requested serial data transfers instead. A good example of a game that benefits from this implementation is *Nintendo Baseball* (1989), which would previously hang on start-up when no serial I/O emulation was present.

An obvious course of action for any future changes to the software would involve extending the scope of this objective to allow for the full implementation of the CGB’s serial I/O and infrared functionality. Additionally, further time could be spent improving the accuracy of emulation for obscure hardware interactions (as tested for by some miscellaneous hardware emulation tests described in Section 6.2).

7.1.1.2 Implementation of a Basic GUI

The software’s main GUI window presents a simple screen area for showing the live emulated LCD video output, with all other program interactions contained within the

window's menu bar as described in Section 4.1. This functionality meets the original requirements associated with this objective.

In the future, additional features such as a full screen mode for the emulated LCD video output and the automatic pausing of the emulation when the GUI window loses focus could be implemented. Additionally, a method for seamlessly switching between the main GUI and command-line debugger modes via the menu bar without needing to restart the application would be a nice convenience.

7.1.1.3 Extensive Testing of the Software

Testing was continuously performed throughout the development process, with periods of extensive testing carried out before certain project deadlines as described in Section 6. On top of this, a dedicated mode for debugging the state of the emulated CPU was implemented to aid in the testing process.

A useful improvement that could be made is to write a build script to automatically runs all tests every time the software executable is rebuilt. Doing so, however, will require some form of automation for verifying the result of tests which require some human input such as the Mooneye GB PPU Object Priority test.

7.1.2 Optional Objectives

The term “optional objectives” is used to refer to the project's low priority objectives. Any work done towards these objectives should ideally be done after the completion of all the key objectives previously described.

7.1.2.1 Implementation of Debugging Tools

Although an optional objective, it was eventually deemed necessary during development to create the emulator's command-line debugger mode before the completion of the key objectives. In addition to aiding the testing process, the addition of the debugger provided a method for demonstrating the software's features for the project prototype demonstration, which at the time consisted of only the emulated CPU and memory modules.

The command-line debugger's feature set has much room for expansion. For example, there were considerations to implement a basic disassembler for the CPU's instruction set, which would allow the software to translate program machine code into a more human-readable instruction mnemonic format. Additionally, the implementation of breakpoints for emulated programs would greatly aid in the testing process, allowing for the emulated CPU to be paused before the execution of certain program instructions suspected of causing emulation issues.

The implementation of extra debugging tools such as a GUI viewer for graphics tiles stored within VRAM and graphics objects within OAM-RAM would be a suitable goal for the future.

7.1.2.2 Implementation of Save States

Save states are snapshots of emulated memory typically used to restore the state of emulation to some point in the past [17]. Due to time constraints, save states were not implemented.

The added ability to import and export battery-packed cartridge RAM snapshots does to an extent allow some programs to restore aspects of their state in a similar fashion to that of save states. Unlike save states however, the effectiveness of this feature depends on the implementation of the programs themselves, and so is not a universal replacement for them.

7.1.2.3 Implementation of Key Remapping and Input Playback

Due to time constraints, neither keyboard key remapping or input playback for the emulated joypad was implemented. Meeting these objectives was a stretch goal for the project, and so was assigned the lowest priority for completion.

While key remapping serves mostly as a user experience improvement, input playback, on the other hand, is a useful feature to have available during the testing process. For example, input playback can be used to replay the exact program inputs necessary in replicating a hard to reproduce emulation issue. For this reason, the objective to implement input playback should maybe have been assigned a higher completion priority.

7.2 Social and Academic Context

In a social context, the emulation of discontinued computer systems such as the CGB is especially important in the preservation of their history [44]. As time goes on, existing Game Boy hardware will either be lost or begin to fail, making emulation a necessity in running programs made for these systems in the foreseeable future.

However, the emulation of computer systems, especially video game consoles, has frequently raised questions regarding the legality of the practice in the past. While emulation itself does not violate any laws, the method in which program ROMs are obtained can potentially breach copyright regulations. To avoid such issues in this project, the program ROMs used for compatibility testing were obtained from their respective physical cartridges as described in Section 6.3, which is widely regarded as fair use under archival or preservation purposes [45] [46].



Figure 52 – The hexadecimal output of a custom-made Game Boy program captured from the command-line debugger. The program calculates the first 10 Fibonacci numbers (from right to left).

In an academic context, the emulation software may be of interest to students studying computer systems courses, as the software’s command-line debugger can be used to effectively demonstrate how a real CPU behaves during program execution on an instruction-by-instruction basis. The debugger was utilised in the project prototype demonstration for a similar purpose, whereby simple custom-made program ROMs written in Assembly were ran to demonstrate the capabilities of the emulated CPU.

7.3 Personal Development

The research done for this project has refined my knowledge regarding how real computer systems work in practice, and how programs take advantage of hardware features to produce graphics and sound output and retrieve user input on the Game Boy. Importantly, this project has been an exercise in time management and planning, in which I have had to break down objectives into smaller, more manageable tasks to be completed throughout the year.

From a technical standpoint, by writing the emulation software in C++, I have had the opportunity to expand my knowledge regarding the standard template library, threading and language-specific features such as template metaprogramming. Additionally, the use of idioms such as the CRTP described in Section 5.1.4 has expanded my knowledge of software design patterns and the effects they have on both code maintainability and performance. Furthermore, I have learned how to use the CMake build system together with the wxWidgets and SFML libraries to deploy cross-platform code for graphics and audio, which has allowed the emulation software to run on both Windows and Linux.

8 Conclusion

In conclusion, I have developed emulation software for the CGB using an LLE approach that is able to run and debug the execution of many CGB and DMG programs with a good level of compatibility as evidenced by the test results detailed in Section 6. I have managed to complete all the key project objectives and some additional optional objectives. With all this in mind, I believe the project was a success overall.

Although I am pleased with the outcome of the software, there is still some room for future expansion, as some optional objectives were not met regarding the implementation of save states, key remapping and input playback for the emulated joypad, for example. There are also obvious improvements to be made concerning program compatibility; however, as of now, even the most accurate Game Boy emulators available such as *Gambatte* and *BGB* do not yet have perfect program compatibility, as CGB and DMG emulation is still an ongoing area of research [47] [48].

9 Bibliography

- [1] BusinessWeek, “A Brief History of Game Console Warfare: Game Boy,” [Online]. Available: https://web.archive.org/web/20070509094404/http://images.businessweek.com/ss/06/10/game_consoles/source/7.htm. [Accessed 12 March 2018].
- [2] The Pokémon Company, “Business Summary | The Pokémon Company,” [Online]. Available: <http://www.pokemon.co.jp/corporate/en/services/>. [Accessed 12 March 2018].
- [3] A. Frank, “A chronological history of Pokémon games | Polygon,” 26 February 2016. [Online]. Available: <https://www.polygon.com/pokemon/2016/2/26/11120098/pokemon-games-list-history-timeline-release-dates>. [Accessed 12 March 2018].
- [4] “High/Low level emulation - Emulation General Wiki,” [Online]. Available: http://emulation.gametechniki.com/index.php/High/Low_level_emulation. [Accessed 11 March 2018].
- [5] “Test ROMs - GbdevWiki,” [Online]. Available: http://gbdev.gg8.se/wiki/articles/Test_ROMs. [Accessed 12 March 2018].
- [6] E. Amos, “File:Nintendo-Game-Boy-Color-FL.jpg,” 13 March 2015. [Online]. Available: <https://commons.wikimedia.org/wiki/File:Nintendo-Game-Boy-Color-FL.jpg>. [Accessed 12 March 2018].
- [7] Nintendo, “ゲームボーイカラー (Game Boy Color),” [Online]. Available: <https://www.nintendo.co.jp/no2/dmg/hardware/color/>. [Accessed 12 March 2018].
- [8] Nintendo, “Game Boy Owner's Manual,” 1989. [Online]. Available: http://www.videogameconsolelibrary.com/images/handhelds/manuals/89_nintendo-game-boy-usa.pdf. [Accessed 22 March 2018].
- [9] Base Media, “Nintendo Game Boy Color Console Information,” [Online]. Available: <https://www.consoledatabase.com/consoleinfo/nintendogameboycolor/>. [Accessed 12 March 2018].

- [10] M. Fayzullin, P. Felber, P. Robson, M. Korth, n. and k. , “Pan Docs,” October 2001. [Online]. Available: <http://bgb.bircd.org/pandocs.htm>. [Accessed 12 March 2018].
- [11] Nintendo, “Game Boy Programming Manual Version 1.0,” 11 September 1999. [Online]. Available: <http://www.chrisantonellis.com/files/gameboy/gb-programming-manual.pdf>. [Accessed 11 March 2018].
- [12] Nintendo, “Four Player Adapter | Game Boy / Pocket / Color,” [Online]. Available: <https://www.nintendo.co.uk/Support/Game-Boy-Pocket-Color/FAQ/Four-Player-Adapter/Four-Player-Adapter-619640.html>. [Accessed 12 March 2018].
- [13] “Game Link Cable - Bulbapedia,” [Online]. Available: https://bulbapedia.bulbagarden.net/wiki/Game_Link_Cable. [Accessed 15 March 2018].
- [14] “List of Game Boy Color games with IR support - Nintendo Wiki,” [Online]. Available: http://nintendo.wikia.com/wiki/List_of_Game_Boy_Color_games_with_IR_support. [Accessed 15 March 2018].
- [15] “Game Pak - Nintendo Wiki,” [Online]. Available: http://nintendo.wikia.com/wiki/Game_Pak. [Accessed 12 March 2018].
- [16] Nintendo, “Game Boy - Compatibility Chart,” [Online]. Available: <https://www.nintendo.com/consumer/systems/gameboy/compatibilitychart.jsp>. [Accessed 12 March 2018].
- [17] “User guide: Save states - OpenEmu/OpenEmu Wiki - GitHub,” 19 December 2017. [Online]. Available: <https://github.com/OpenEmu/OpenEmu/wiki/User-guide:-Save-states>. [Accessed 12 March 2018].
- [18] “FCEUX: Movies,” [Online]. Available: <http://www.fceux.com/web/movies.html>. [Accessed 12 March 2018].
- [19] J. Javanainen, “Reverse Engineering fine details of Game Boy hardware,” in *Disobey*, Helsinki, 2018.

- [20] "Gameboy sound hardware - GbdevWiki," [Online]. Available: http://gbdev.gg8.se/wiki/articles/Gameboy_sound_hardware. [Accessed 13 March 2018].
- [21] "GameBoy Tetris kaufen | 9037715 | Konsolenkost," [Online]. Available: <https://www.konsolenkost.de/gameboy-tetris-modul-mit-anl-gebraucht/a-9037715/>. [Accessed 16 March 2018].
- [22] "The Nintendo® Game Boy™, Part 3: The Rest of the Hardware | RealBoy," 2 January 2013. [Online]. Available: <https://realboyemulator.wordpress.com/2013/01/02/the-nintendo-game-boy-part-3/>. [Accessed 21 March 2018].
- [23] "Game Boy Color Bootstrap ROM - The Cutting Room Floor," [Online]. Available: https://tcrf.net/Game_Boy_Color_Bootstrap_ROM. [Accessed 19 March 2018].
- [24] "The Nintendo® Game Boy™, Part 1: The Intel 8080 and the Zilog Z80. | RealBoy," 1 January 2013. [Online]. Available: <https://realboyemulator.wordpress.com/2013/01/01/the-nintendo-game-boy-1/>. [Accessed 7 April 2018].
- [25] "The Nintendo® Game Boy™, Part 2: The Game Boy's CPU | RealBoy," 2 January 2013. [Online]. Available: <https://realboyemulator.wordpress.com/2013/01/02/the-nintendo-game-boy-part-2/>. [Accessed 11 April 2018].
- [26] "GameBoy Opcode Summary," [Online]. Available: <http://www.devrs.com/gb/files/opcodes.html>. [Accessed 11 April 2018].
- [27] ZiLOG, "Z80 Family CPU User Manual (UM008004-1204)," December 2004. [Online]. Available: http://www.z80.info/zip/z80cpu_um.pdf. [Accessed 11 April 2018].
- [28] "Gameboy (LR35902) OPCODES," [Online]. Available: http://www.pastraiser.com/cpu/gameboy/gameboy_opcodes.html. [Accessed 16 April 2018].
- [29] J. Javanainen, "mooneye-gb/boot_regs-cgb.s at master - Gekkio/mooneye-gb," 31 January 2018. [Online]. Available:

https://github.com/Gekkio/mooneye-gb/blob/master/tests/misc/boot_regscgb.s. [Accessed 16 April 2018].

- [30] A. N. Díaz, “The Cycle-Accurate Game Boy Docs Version 0.0.X,” 2015. [Online]. Available: <https://github.com/AntonioND/giibiiadvance/blob/master/docs/TCAGBD.pdf>. [Accessed 22 March 2018].
- [31] “GameBoy Memory Map,” [Online]. Available: <http://gameboy.mongenel.com/dmg/asmmemmap.html>. [Accessed 13 March 2018].
- [32] “Randomizer Theory | Page 9 | Tetrisconcept,” 10 May 2013. [Online]. Available: <https://tetrisconcept.net/threads/randomizer-theory.512/page-9#post-51036>. [Accessed 22 March 2018].
- [33] “Create a Replica of the Legendary Game Boy Pocket in Photoshop | Naldz Graphics,” [Online]. Available: <https://naldzgraphics.net/game-boy-pocket-in-photoshop/>. [Accessed 22 March 2018].
- [34] “Joypad Input - GbdevWiki,” [Online]. Available: http://gbdev.gg8.se/wiki/articles/Joypad_Input. [Accessed 22 March 2018].
- [35] M. Steil, “The Ultimate Game Boy Talk,” in *33c3*, 2016.
- [36] “MATH 6.4: Waves and partial differential equations,” 1996. [Online]. Available: http://www.physics.brocku.ca/PPLATO/h-flap/math6_4.html. [Accessed 20 April 2018].
- [37] Morn, “File:White noise.svg,” 22 January 2013. [Online]. Available: https://commons.wikimedia.org/wiki/File:White_noise.svg. [Accessed 21 April 2018].
- [38] “Memory Bank Controllers - GbdevWiki,” [Online]. Available: http://gbdev.gg8.se/wiki/articles/Memory_Bank_Controllers. [Accessed 18 March 2018].
- [39] R. Pate, “c++ - Jump Table Switch Case question - Stack Overflow,” 3 December 2009. [Online]. Available: <https://stackoverflow.com/a/1837986>. [Accessed 2 May 2018].

- [40] “Curiously recurring template pattern - Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern. [Accessed 2 May 2018].
- [41] “Virtual method table - Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Virtual_method_table. [Accessed 2 May 2018].
- [42] J. Javanainen, “mooneye-gb/sprite_priority.s at master - Gekkio/mooneye-gb,” 31 January 2018. [Online]. Available: https://github.com/Gekkio/mooneye-gb/blob/master/tests/manual-only/sprite_priority.s. [Accessed 25 April 2018].
- [43] “Reader/Writer Gen3 - The joey-joebags - BennVenn's Shop,” [Online]. Available: <https://bennvenn.myshopify.com/products/reader-writer-gen2>. [Accessed 24 April 2018].
- [44] “Nintendo Game Boy Color (1998 - 2003) - Museum Of Obsolete Media,” [Online]. Available: <http://www.obsoletemedia.org/nintendo-game-boy-color/>. [Accessed 27 April 2018].
- [45] user56, “What exactly does the law state about emulation and ROMs? - Arqade Meta,” 6 August 2010. [Online]. Available: <https://gaming.meta.stackexchange.com/a/796>. [Accessed 30 April 2018].
- [46] GameCentral, “The Legality of Emulation, Part 2 - Reader’s Feature | Metro News,” 17 February 2013. [Online]. Available: <http://metro.co.uk/2013/02/17/the-legality-of-emulation-part-2-readers-feature-3480905/>. [Accessed 30 April 2018].
- [47] “Nintendo Game Boy (Color) Core Compatibility - Libretro Wiki,” [Online]. Available: [https://wiki.libretro.com/index.php?title=Nintendo_Game_Boy_\(Color\)_Core_Compatibility](https://wiki.libretro.com/index.php?title=Nintendo_Game_Boy_(Color)_Core_Compatibility). [Accessed 30 April 2018].
- [48] “bgb readme,” [Online]. Available: <http://bgb.bircd.org/manual.html#knownproblems>. [Accessed 30 April 2018].
- [49] “Reverse Engineering the Gameboy Boot Screen,” 19 December 2015. [Online]. Available: <https://catskull.net/gameboy-boot-screen-logo.html>. [Accessed 18 March 2018].

- [50] “Memory Map - GbdevWiki,” [Online]. Available:
http://gbdev.gg8.se/wiki/articles/Memory_Map. [Accessed 20 March 2018].
- [51] Appaloosa, “File:Z80 arch.svg,” 17 October 2007. [Online]. Available:
https://commons.wikimedia.org/wiki/File:Z80_arch.svg. [Accessed 12 April 2018].
- [52] Appaloosa, “File:Intel 8080 arch.svg,” 14 November 2008. [Online]. Available:
https://commons.wikimedia.org/wiki/File:Intel_8080_arch.svg. [Accessed 12 April 2018].

10 Appendix

10.1 Popular Cartridge Configurations

Some of the most popular cartridge MBC hardware configurations include: [38]

Configuration	Details
No MBC	<ul style="list-style-type: none">• At most two banks of program ROM (32 KiB total).• Supports up to one bank of RAM (8 KiB total).
MBC1	<ul style="list-style-type: none">• At most 125 banks of program ROM (2 MiB total).• Supports up to four banks of RAM (32 KiB total).
MBC2	<ul style="list-style-type: none">• At most 16 banks of program ROM (256 KiB total).• 512×4 bits of RAM built-into the MBC chip. Because each RAM location is only 4 bits in size, only the lower nibble of each location is addressable by the DMG/CGB.
MBC3	<ul style="list-style-type: none">• At most 128 banks of program ROM (2 MiB total).• Supports up to four banks of RAM (32 KiB total).• A Real-Time Clock (RTC) is built-into the MBC chip, allowing programs to optionally keep track of the current date and time of day. Requires an internal battery to allow the clock to tick while the DMG/CGB is powered off.
MBC5	<ul style="list-style-type: none">• At most 512 banks of program ROM (8 MiB total).• Supports up to 16 banks of RAM (128 KiB total).• Supports CGB double speed mode (see Section 3.2.4 for details).

10.2 Important Cartridge Header Fields

Notable data fields contained within the header include: [10, p. The Cartridge Header]

Address	Field Name	Details
\$0100 to \$0103	Program Entry Point	Contains three bytes of program code to be executed by the CPU once the boot procedure ends. Due to the limited capacity of the field, a single jump instruction is usually encoded here, causing the CPU to

		start executing program code in another area of ROM (see Section 3.2.2 for details).
\$0104 to \$0133	Nintendo Logo	<p>Contains the 48-byte bitmap data of the Nintendo logo that is displayed briefly by the boot procedure.</p> <p>The boot procedure does not continue further if the contents of this field differs from the bitmap stored within the boot ROM. This was used as a rudimentary form of protection to stop unlicensed games from booting [49].</p>
\$0143	CGB Compatibility Flag	<p>Contains \$80 if the program supports CGB functionalities (such as non-monochrome colours) and is also backwards compatible with the DMG.</p> <p>Contains \$C0 if the program only supports CGB functionalities and is not backwards compatible with the DMG.</p> <p>Contains any other value if the program specifically supports the DMG. The CGB will run the program in backwards compatibility mode in this case.</p>
\$0147	MBC Configuration Type	<p>Indicates the type of MBC hardware configuration used by the cartridge.</p> <p>Some typical values include:</p> <ul style="list-style-type: none"> • \$00 – Program ROM only. • \$01 to \$03 – MBC1 variants. • \$05 to \$06 – MBC2 variants. • \$0F to \$13 – MBC3 variants. • \$19 to \$1E – MBC5 variants.
\$0148	ROM Size	<p>Indicates the number of program ROM banks available within the cartridge.</p> <p>Bank numbers are typically powers of two, where 2^N is encoded as a value of $N-1$; therefore, values generally range from \$00 (two banks: 2^1) to \$07 (256 banks: 2^8).</p>

		An exception to the general encoding rule exists for values \$52, \$53 and \$54, which indicate respective bank numbers of 72, 80 and 96, that are not powers of two.
\$0149	RAM Size	<p>Indicates the number of RAM banks present inside of the cartridge.</p> <p>All possible values are:</p> <ul style="list-style-type: none"> • \$00 – No RAM. • \$01 – One bank of RAM, with only 2 KiB usable. • \$02 – One bank of RAM (8 KiB). • \$03 – Four banks of RAM (32 KiB). <p>A value of \$00 is also used to indicate that an MBC2 configuration is used, as MBC2's 512×4 bits of RAM is integrated within the chip itself.</p>
\$014D	Header Checksum	<p>Contains a single-byte checksum of the header data (from addresses \$0134 to \$014C) to be verified by the boot procedure. This checksum is used to verify the integrity of the header data stored within the program ROM.</p> <p>If the checksum calculated by the boot procedure differs from this value, the loaded program will not be executed.</p>

10.3 Detailed System Memory Map

Memory areas are mapped to the CGB's address space as follows: [10, p. Memory Map] [11, p. 13] [31]

Address Range	Memory Area	Additional Comments
\$0000 to \$3FFF	Cartridge ROM Bank	Fixed to bank 0.
\$4000 to \$7FFF	Cartridge ROM Bank	Swappable if the cartridge uses an MBC; otherwise, fixed to bank 1.
\$8000 to \$9FFF	VRAM Bank	Swappable (using the VBK I/O register: address \$FF4F) if not in

		DMG compatibility mode; otherwise, fixed to bank 0.
\$A000 to \$BFFF	Cartridge RAM Bank (if any)	Swappable if the cartridge uses an MBC.
\$C000 to \$CFFF	WRAM Bank	Fixed to bank 0.
\$D000 to \$DFFF	WRAM Bank	Swappable (using the SVBK I/O register: address \$FF70) if not in DMG compatibility mode; otherwise, fixed to bank 1.
\$E000 to \$FDFF	Echo WRAM	Accesses to this area map to the WRAM at \$C000 to \$DDFF. Programs rarely access this area, as Nintendo explicitly prohibits its use.
\$FE00 to \$FE9F	OAM-RAM	
\$FEA0 to \$FEFF	Unusable	The behaviour of accesses to this area are not well documented. Reading from this area may return different values depending on the type of Game Boy in use [50].
\$FF00 to \$FF2F	I/O Registers	
\$FF30 to \$FF3F	Wave RAM	
\$FF40 to \$FF7F	I/O Registers	
\$FF80 to \$FFFE	HRAM	
\$FFFF	I/O Registers	Contains the CPU's IE flag.

10.4 Additional System Timer I/O Register Details

The internal timer exposes the following I/O registers: [10, p. Timer and Divider Registers]

Address	I/O Register	Additional Comments
\$FF04	Divider (DIV)	Writing to this register sets its value to \$00.
\$FF05	Timer Counter (TIMA)	
\$FF06	Timer Modulo (TMA)	
\$FF07	Timer Control (TAC)	<p>The value of bits 1-0 of this register control the timer's ticking frequency as follows:</p> <ul style="list-style-type: none"> 00 – Ticks every 1,024 CPU clock cycles (4,096 Hz, or 8,192 Hz in CPU

		<p>double speed mode).</p> <ul style="list-style-type: none"> • 01 – Ticks every 16 CPU clock cycles (262,144 Hz, or 524,288 Hz in CPU double speed mode). • 10 – Ticks every 64 CPU clock cycles (65,536 Hz, or 131,072 Hz in CPU double speed mode). • 11 – Ticks every 256 CPU clock cycles (16,384 Hz, or 32,768 Hz in CPU double speed mode). <p>The ticking of the timer is disabled if bit 2 of this register is unset; however, this does not stop the DIV register automatically incrementing.</p>
--	--	--

10.5 Block Diagrams of the Z80 and 8080

Z80 Architecture

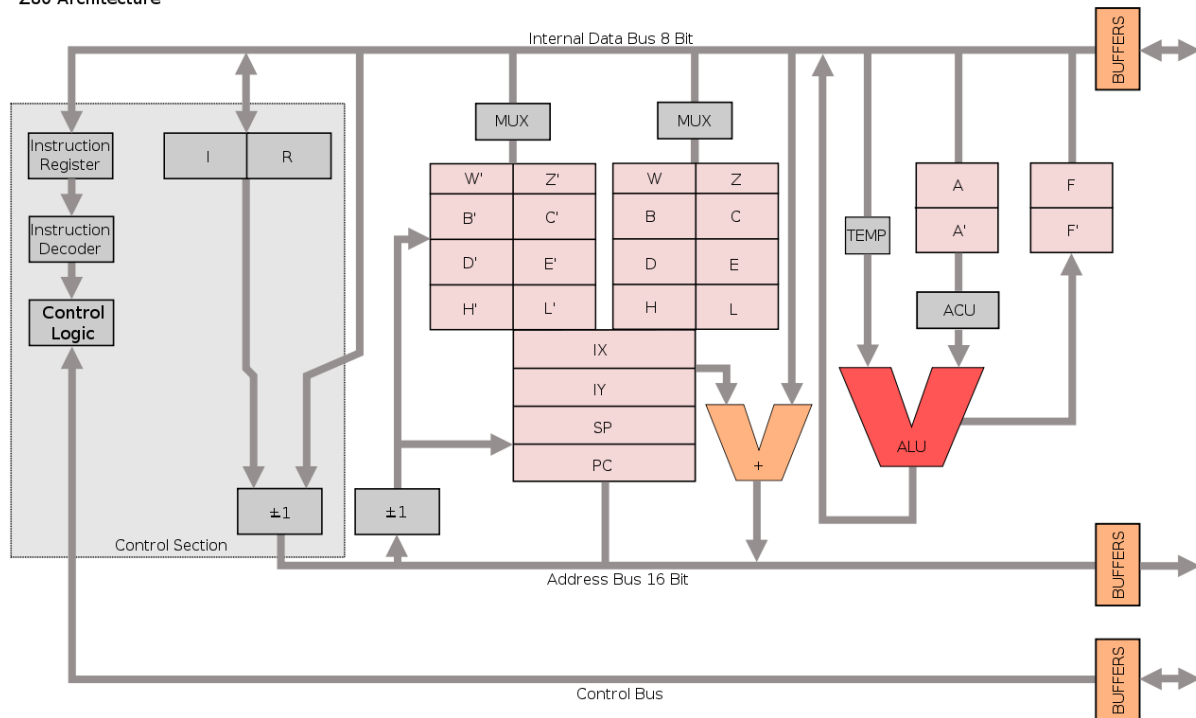


Figure 53 – An approximate block diagram of the Zilog Z80 [51].

Intel 8080 Architecture

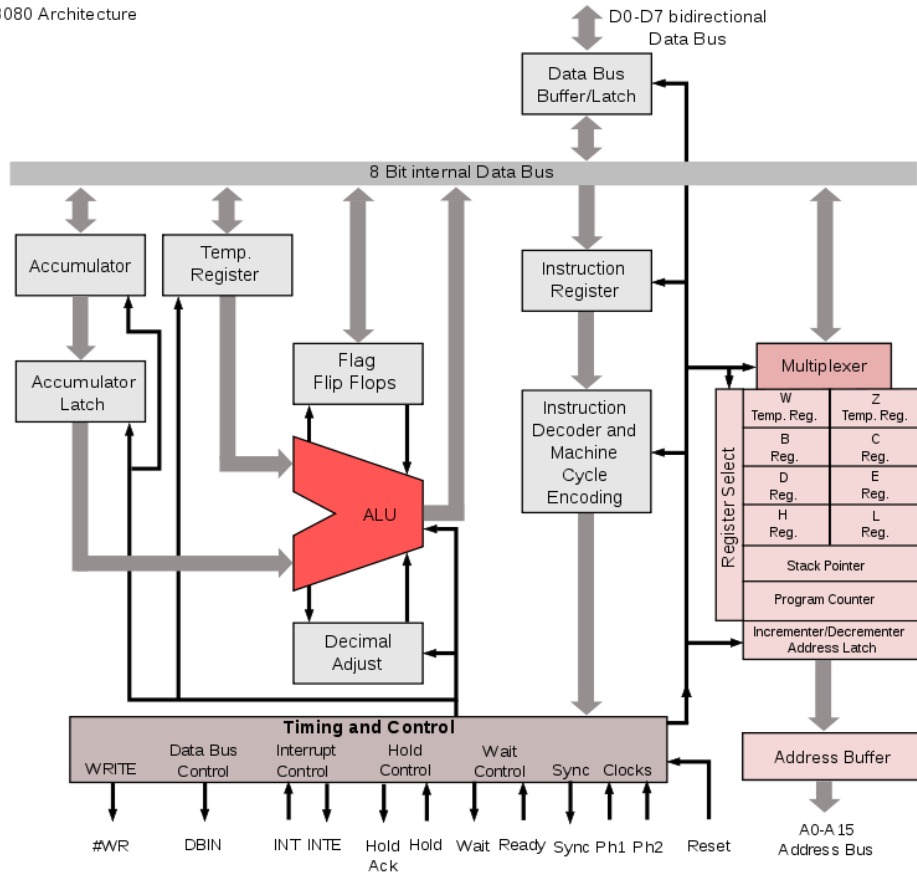


Figure 54 – An approximate block diagram of the Intel 8080 [52].

10.6 More Complete List of CPU Instructions

The following is a more complete (but simplified) list of CPU instructions to compliment the information within Section 3.2.3: [11, pp. 84-112] [26]

10.6.1 8-Bit Loads

Instruction	Details
LD (<i>8-bit variants</i>)	Write to an 8-bit CPU register or memory location with 8-bit data.
LDI	Either write the contents of the Accumulator to the memory location pointed to by HL or vice versa. Increment the value of HL afterwards.
LDD	Like LDI, but the value of HL is instead decremented afterwards.
LDH	Like LD, but operands are either the Accumulator or a memory location within the last memory page (\$FF00+N, where N is an 8-bit value operand).

10.6.2 Stack Operations and 16-Bit Loads

Instruction	Details
PUSH	Push a 16-bit value from a CPU register pair onto the top of the program stack.
POP	Pop a 16-bit value from the top of the program stack into a CPU register pair.
LD (<i>16-bit variants</i>)	Write to a CPU register pair, the Stack Pointer or memory location with 16-bit data. A variant of this instruction, LD HL, (SP+N) exists for loading a 16-bit value into HL from the memory address pointed to by the Stack Pointer + N (where N is an 8-bit two's complement offset operand).

10.6.3 8-Bit General ALU, Bit Testing, Setting and Resetting Operations

Instruction	Details
ADD (<i>8-bit variants</i>)	Add the value of an 8-bit CPU register or memory location to the Accumulator.
ADC	Like ADD, but the value of the Accumulator is incremented afterwards if the Status Flag's Carry bit is set.

SUB	Subtract the value of an 8-bit CPU register or memory location from the Accumulator.
SBC	Like SUB, but the value of the Accumulator is decremented afterwards if the Status Flag's Carry bit is set.
AND	Perform a bitwise AND on the Accumulator and an 8-bit CPU register or memory location. Write the result to the Accumulator.
OR	Perform a bitwise inclusive OR on the Accumulator and an 8-bit CPU register or memory location. Write the result to the Accumulator.
XOR	Perform a bitwise exclusive OR (XOR) on the Accumulator and an 8-bit CPU register or memory location. Write the result to the Accumulator.
CP	Like SUB, but the result of the operation is not written to the Accumulator. This instruction can be used to compare two values for equality by testing the value of the Status Flag's Zero bit afterwards.
INC (8-bit variants)	Increment the value of an 8-bit CPU register or memory location pointed to by HL.
DEC (8-bit variants)	Decrement the value of an 8-bit CPU register or memory location pointed to by HL.
BIT	Test if a specific bit within an 8-bit CPU register or memory location pointed to by HL is set. After the operation, the Status Flag's Zero bit is set if the specified bit was set, unset otherwise.
SET	Set a specific bit within an 8-bit CPU register or memory location pointed to by HL.
RES	Reset a specific bit within an 8-bit CPU register or memory location pointed to by HL.

10.6.4 16-Bit ALU Operations

Instruction	Details
ADD (16-bit variants)	Add the value of a 16-bit CPU register pair to HL or an 8-bit signed two's complement Immediate value to the Stack Pointer.
INC (16-bit variants)	Increment the value of the Stack Pointer or a 16-bit CPU register pair.
DEC (16-bit variants)	Decrement the value of the Stack Pointer or a 16-bit CPU register pair.

10.6.5 8-Bit ALU Bitwise Rotation and Shift Operations

Instruction	Details
RLC	Perform a bitwise left rotate on the value of an 8-bit CPU register or memory location pointed to by HL.
RL	Perform a bitwise left rotate through carry (using the Carry bit of the Status Flag register) on the value of an 8-bit CPU register or memory location pointed to by HL.
RRC	Perform a bitwise right rotate on the value of an 8-bit CPU register or memory location pointed to by HL.
RR	Perform a bitwise right rotate through carry (using the Carry bit of the Status Flag register) on the value of an 8-bit CPU register or memory location pointed to by HL.
RLCA	Like RLC, except the source and destination register is always the Accumulator.
RLA	Like RL, except the source and destination register is always the Accumulator.
RRCA	Like RRC, except the source and destination register is always the Accumulator.
RRA	Like RR, except the source and destination register is always the Accumulator.
SLA	Perform a bitwise left shift on the value of an 8-bit CPU register or memory location pointed to by HL.
SRA	Perform a bitwise arithmetic (signed) right shift on the value of an 8-bit CPU register or memory location pointed to by HL.
SRL	Perform a bitwise logical right shift on the value of an 8-bit CPU register or memory location pointed to by HL.

10.6.6 Jumps, Restarts and Subroutine Operations

Instruction	Details
JP*	Set the value of the Program Counter to the value of the 16-bit memory location pointed to by HL or a 16-bit Immediate Extended value.
JR*	Add an 8-bit signed two's complement Immediate value to the Program Counter.

CALL*	Push the 16-bit value of the Program Counter (which points to the next instruction) onto the top of the program stack, then set the value of the Program Counter to a 16-bit Immediate Extended value.
RST**	Like CALL, except that the value of the Program Counter is set to one of the following addresses within the first page of memory: \$0000, \$0010, \$0020, \$0030.
RET*	Pop a 16-bit value from the top of the program stack and write it to the Program Counter.
RETI	Same functionality as the RET and EI instruction.

* Variations of these instruction exist for conditional branching, where the specified operation will only be performed if the Zero or Carry bits within the Status Flag register are set or unset.

** Although like a CALL instruction, no conditional branching variations of this instruction exist.

10.6.7 Miscellaneous Operations

Instruction	Details
EI	Set the CPU's IME flag. This enables interrupt handling.
DI	Unset the CPU's IME flag. This disables interrupt handling.
HALT	Halts the further execution of program instructions until an interrupt is both enabled and requested via the IE and IF I/O registers. This happens regardless of the IME flag's value (if IME is unset however, no actual interrupt will be triggered as expected).
STOP	Like HALT, except that execution is only resumed if a joystick button selected by the JOYP I/O register is pressed (see Section 3.4).
SWAP	Swap the position of the most and least-significant nibbles of an 8-bit CPU register or memory location pointed to by HL.
DAA	Adjusts the value of the Accumulator to allow for Binary-Coded Decimal (BCD) arithmetic following an ADD, ADC, SUB or SBC operation. The exact process is described within page 110 of the <i>Game Boy Programming Manual</i> .
CPL	Perform a bitwise NOT (complement) on the 8-bit value of the Accumulator. Overwrite the value of the Accumulator with this result.
CCF	Perform a bitwise NOT (complement) on the Carry bit of the Status Flag register.
SCF	Sets the Carry bit of the Status Flag register.
NOP	No operation.

10.6.8 CGB Background Layer Tile Map Attributes

The following bits within each attribute byte have the following effects: [10, p. VRAM Background Maps]

Attribute	Bit Number	Details
Tile Palette Number	0 to 2	A 3-bit number used to refer to a palette of colours within Palette RAM to apply to the tile.
Tile Pattern VRAM Bank Number	3	This bit configures the VRAM bank number of the Tile Pattern Table to use.
Horizontal Flip	5	If set, the tile is mirrored horizontally when rendered.
Vertical Flip	6	If set, the tile is mirrored vertically when rendered.
Priority Above Object Layer	7	If set, this tile appears above any tiles on the object layer, regardless of any other priority settings.

10.6.9 Object Layer Tile Attributes

The attributes for each object stored within OAM-RAM are as follows: [10, p. VRAM Sprite Attribute Table (OAM)]

Attribute	Byte Number	Details
Object Y Position	0	The vertical screen position of the top of the object in pixels minus 16. An off-screen value of 0 or more than 159 hides the object. Objects hidden this way will not count towards the PPU's 10 objects per scanline limitation.
Object X Position	1	The horizontal screen position of the left of the object in pixels minus 8. An off-screen value of 0 or more than 167 hides the object.
Tile Pattern Number	2	Selects a tile pattern from the Tile Pattern Table at address \$8000 to \$8FFF in memory. A pixel palette colour number of 0 is always treated as transparency.
Additional Attributes	3	Specifies additional attributes for the object.

		<p>The bits in this byte operate in the same manner as those explained in Section 3.5.3.1, but with the following differences:</p> <ul style="list-style-type: none"> • Bit 4 specifies the palette number to use when in DMG compatibility mode. • Bits 5 and 6, which are used to mirror the object horizontally and vertically, may be used in DMG compatibility mode. • Bit 7, when set, allows tile pixels from the background layers to display above the object if their palette colour numbers (defined within the tile pattern) are 1 to 3.
--	--	---

10.7 Generated Class Diagram

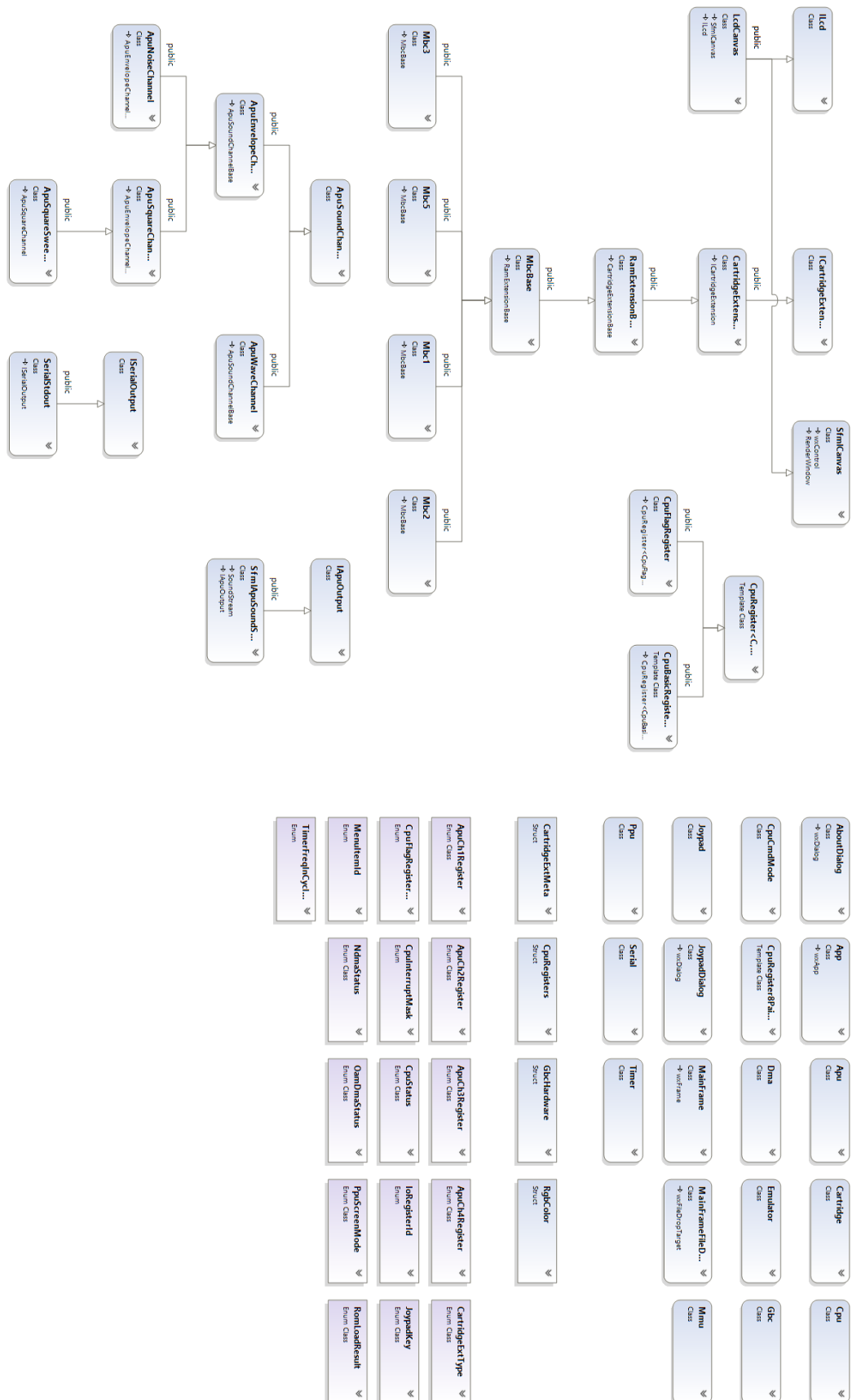


Figure 55 – A class diagram of the emulator generated by Visual Studio.

10.8 Test Results

This section includes the results of some notable software tests carried out during development as described in Section 6.

10.8.1 Blargg's cpu_instrs Tests

Test	Result Output
01-special	Passed
02-interrupts	Passed
03-op sp,hl	Passed
04-op r,imm	Passed
05-op rp	Passed
06-ld r,r	Passed
07-jp,jr,call,ret,rst	Passed
08-misc instrs	Passed
09-op r,r	Passed
10-bit ops	Passed
11-op a,(hl)	Passed

10.8.2 Miscellaneous Blargg CPU Tests

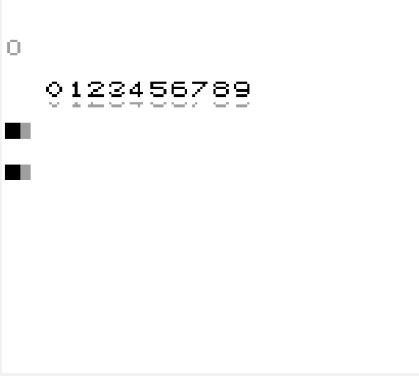
Test	Result Output
instr_timing	Passed
interrupt_time	00 00 00 00 08 0D 01 00 00 01 08 0D Passed

10.8.3 Blargg's cgb_sound Tests

Test	Result Output
01-registers	NR10-NR51 and wave RAM write/read Failed #2
02-len ctr	0 1 2 3 Passed
03-trigger	0

	Enabling in first half of length period should clock length Failed #3
04-sweep	Passed
05-sweep details	Exiting negate mode after calculation disables channel Failed #4
06-overflow on trigger	0555 0666 071C 0787 07C1 07E0 07F0 0556 0667 071D 0788 07C2 07E1 07F1 Passed
07-len sweep period sync	Passed
08-len ctr during power	40 00 40 40 Passed
09-wave read while on	89339D Failed
10-wave trigger while on	D7871ED6 Failed
11-regs after power	Passed
12-wave	Timer period or phase resetting is wrong Failed #2

10.8.4 Mooneye GB PPU Object Priority Test

Test	Result Output
sprite_priority	 <p><i>Passed (matches expected output)</i></p>

10.8.5 Manual Program Compatibility Tests

The following is a full list of performed program compatibility tests:

Program	Target System	Compatibility Comments
<i>The Legend of Zelda – Link’s Awakening</i>	DMG	No issues.
<i>The Legend of Zelda – Link’s Awakening DX</i>	CGB/ DMG	No issues.
<i>The Legend of Zelda – Oracle of Seasons</i>	CGB	No issues.
<i>Castlevania Legends</i>	DMG	No issues.
<i>Pokémon Yellow Version</i>	CGB/ DMG	No issues.
<i>Pokémon Crystal Version</i>	CGB	Loading a saved game may display a warning that the in-game date and time needs to be reset by the user. Additionally, the in-game date and time doesn’t progress. This is expected, as the MBC3 RTC is not being fully emulated.

<i>Perfect Dark</i>	CGB	Digitized voice and some sound effects played using the wave channel (channel 3) seems to not work. This may have something to do with how the length counter and DAC/channel enable switch is working.
<i>Nintendo Baseball</i>	DMG	No issues.
<i>Cannon Fodder</i>	CGB	The music played using the wave channel (channel 3) in the menus and intro sequence has a noticeable high-pitched tone.
<i>Donkey Kong Country</i>	CGB	No issues.
<i>Donkey Kong Land III</i>	DMG	No issues.
<i>Ghosts 'n Goblins</i>	DMG	No issues.
<i>Jurassic Park</i>	DMG	No issues.
<i>Kirby's Dream Land</i>	DMG	No issues.
<i>Kirby's Dream Land 2</i>	DMG	No issues.
<i>Super Mario Land</i>	DMG	No issues.
<i>Super Mario Land 2 – 6 Golden Coins</i>	DMG	No issues.
<i>Mega Man Xtreme</i>	CGB/ DMG	No issues.
<i>Toy Story Racer</i>	CGB	No issues.
<i>Tetris</i>	DMG	No issues.
<i>Tetris DX</i>	CGB/ DMG	No issues.
<i>Prehistorik Man</i>	DMG	Some graphical effects rely on changing palette colours while the PPU is actively rendering a scanline. Due to how the emulator's PPU scanline rendering is implemented (see Section 5.1.7), these effects will not render as intended.