Partial Emulation of the Nintendo Game Boy

Ian Thomas Brassard
Bard College, ib3655@bard.edu

Follow this and additional works at: https://digitalcommons.bard.edu/senproj_s2023

Part of the Software Engineering Commons

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 4.0 License.

Recommended Citation
https://digitalcommons.bard.edu/senproj_s2023/140

This Open Access is brought to you for free and open access by the Bard Undergraduate Senior Projects at Bard Digital Commons. It has been accepted for inclusion in Senior Projects Spring 2023 by an authorized administrator of Bard Digital Commons. For more information, please contact digitalcommons@bard.edu.
Partial Emulation of the Nintendo Game Boy

Senior Project Submitted to
The Division of Science, Math, and Computing
of Bard College

by
Ian Brassard

Annandale-on-Hudson, New York
May 2023
## Table of Contents

Introduction...................................................................................................................................1  
Background.....................................................................................................................................3  
The Product.......................................................................................................................................8  
Technical Detail.................................................................................................................................11  
Evaluation.......................................................................................................................................17  
Results............................................................................................................................................20  
Discussion.......................................................................................................................................21  
Conclusion.........................................................................................................................................25  
Appendices.......................................................................................................................................26  
Bibliography......................................................................................................................................33
Partial Emulation of the Nintendo Game Boy

Introduction

There are two things that make up a computer: hardware and software. Hardware is, of course, the physical body of the computer, the circuits and wires that carry electrons around. Software is the programs that run on that hardware, manipulating those electrons to produce some effect. But hardware is fragile and temporary, and software is tied to that ephemeral hardware. So what happens, then, if we have a program we want to run but not the machine it was made to run on? In that case, we might consider building an emulator of that machine. An emulator is a piece of software which simulates the functioning of a given piece of hardware. If we were to feed in our orphaned program into that emulator it should, if made correctly, spit out the exact results we would expect if we ran that program on its original hardware.

Emulation has a long and storied history, and there are different kinds of emulators for different purposes. Only in recent years (which is to say, the past decade or two) has emulation achieved widespread familiarity to the average consumer. Much of this success has been through the popularization of video game consoles and the general increase in computing power across the board. The latter is the foundation of the issue, in fact. At risk of stating the obvious, to emulate a given computer you must have a computer which is notably more powerful. If you were to attempt to emulate a computer which has 32 kilobytes of memory and runs at 6
megahertz, but your computer only has 32 kb of memory total and your CPU can barely reach 8 MHz on a good day, any emulation you manage to get running is going to be unstable at best.

This made emulation in the 20th century fairly niche and restrained. Though still notable and interesting, computers were not becoming more powerful quickly enough to reliably emulate previous hardware unless it was on the order of decades older. As computers began to grow exponentially more powerful, however, consumer-level hardware skyrocketed in computing capability. Suddenly, computers that an average consumer could reasonably get were able to emulate hardware from a decade ago or less. Handheld video game consoles such as the Nintendo DS and its successors could run old games from their predecessors, and computer emulators like DOSBox arose as newer hardware generations broke compatibility with old software. Affordable, accessible, and accurate emulation opened new avenues for video game developers and publishers to bring old games to new audiences for little cost. It also made learning to make an emulator of one’s own easier than ever.

This explosion in popularity and greater ease of access has made emulation a go-to project for students and anybody else who is learning about programming, hardware, or video games. Emulators cover several distinct and challenging areas of programming, while giving many a personal connection to their own past or the history of video games. The system I chose for this project is an important part of both my own history and the history of video games. The Game Boy sold genuine boatloads in its heyday, and became so popular that even today it is often the only name recognizable to those unfamiliar with gaming. Personally, I grew up just after the height of its popularity, and played most of its games through the built-in Game Boy
emulator on the Nintendo DS. For me, this project is a return to my roots with new knowledge and perspective, more than a decade later.

**Background**

When I first thought of making an emulator for my senior project, I had never done anything of the sort before. In preparation for making a more complex emulator, I wanted to start with something smaller and simpler. A prototype, perhaps, of the final project. To this end, I spent most of my first semester making an emulator of the CHIP-8 system, the link to which can be found in Appendix C. CHIP-8 was a system specification from the 1970’s, and as such was smaller and less complex than any more modern system I wanted to emulate. Specifically, it only had 4 kb of memory, 30-40 instructions (depending on revision version), a 32x64 pixel screen, and a hexadecimal keypad. Having such a limited amount of instructions makes CHIP-8 an order of magnitude less complex than the Game Boy. The differences in their instruction sets are pretty interesting, too, and reflect different priorities held by each system.

CHIP-8 had much reduced bitwise capabilities. It had the ability to do logical operations like AND and OR, of course, but little else. It had no concept of arithmetic shifting, no ability to test or set bits, and no way to compare two values without doing some consequential mathematical operation. It also had no way to push an arbitrary value to the stack, among other seemingly-essential things. The Game Boy has all of these. CHIP-8, however, also has certain things that the Game Boy does not. For instance, there are opcodes in CHIP-8’s instruction set that test for player input, which is totally absent from the Game Boy’s instruction set. It doesn’t have conditional jumps directly, but it does have two instructions which essentially function as
if-statements. In a way, that system is actually more versatile than the Game Boy’s, since the latter has no single-opcode ways to make conditional statements other than jump instructions.

I got that emulator partially done, complete enough to be considered a demo, and then moved on to the next phase of the project after winter break. For that phase, I wanted to select a system that would be larger, more complex, and more impactful. I was, however, conscious of the limited time available to me, and tried to rein in my expectations of how complex a system I could work on. I mostly looked at handheld video game consoles from the 1980’s and 90’s. This era of video gaming was rich with innovation, proliferation, and as many obscure game systems as there were successful ones. Initially, I wanted to choose one that was less well known, and more in need of care and attention. This process took about a month, and I had to scrap some plans and research on a too-obscure system with little information available. Eventually, however, I settled on the Game Boy. It seemed the perfect system on which I could put into practice the lessons I learned from CHIP-8.

Lessons from CHIP-8

Making a partial CHIP-8 emulator was an educational experience in several ways. It was my first sizable software project, after all. The two most important things I took away from it were these: my choice of programming language was going to be very important, and graphics are difficult and somewhat confusing to work with. For CHIP-8, I decided to use Java in the IntelliJ IDE. This was for a number of reasons, chief of which was my own familiarity with the language (both personally and academically). Java has several undesirable features for hardware emulation, however. The most personally frustrating to me was its refusal to provide small unsigned integer data types, such as a char or byte. Many low-level calculations and operations
need unsigned integer types, and Java only provides signed versions. Using larger integer types was needlessly tricky and annoying, too.

Another problem with Java is one of its more desirable features in higher-level applications: the garbage collector. Garbage collection can delay exactly when operations take place, and when trying to emulate a system which operates at a certain frequency, that delay can cause unexpected and unwanted behaviors. It also simply was not present on the original system, for better or for worse. These were precision-built machines which operated on down-to-the-wire accuracy. There was no room for excess, and as much as modern features like garbage collection are very handy to have, in such a restrained environment they merely represented unnecessary bloat. For CHIP-8 this was not such a problem for me, since I was doing a rather “high level” or abstracted emulation, but my plan for the final project was to be more exacting. Java, much as I love it, had to go.

The second takeaway influenced my choice of language along with the aforementioned problems with Java. Graphics can be quite finicky, and oftentimes the built-in solutions are perhaps passable at best. A language with wide support and easy access to ready-made graphics libraries, such as OpenGL, was a must. Audio libraries were also in consideration, but I never seriously thought I would get to that point in the project, so they were of lesser importance. If not for its poor speed performance, Python would have been a great choice, as it has a vast and robust selection of both audio and graphical capabilities. Alas, it is dynamically typed and thoroughly high-level, so it would not suit the rest of my needs. I needed something lower-level, preferably C-like so as to leverage my familiarity with Java.
I considered two primary candidates: C++ and Rust. I had never worked directly with either, so they were both equally foreign to me. Rust has a lot going for it: it’s the hip and fresh lower-level language, so there’s a lot of good, recent learning materials for it. Of note is the interactive Rust manual, which is something more languages should have. C++, on the other hand, has been around for a while and its users have accrued a deep wealth of knowledge over the years. C++ is also a superset of C, allowing for both classic C-style programming and more modern, fancy ideas like “classes” and “object-oriented programming” and “string literals.” I ultimately chose C++ as my language of choice, for those reasons mentioned previously and also because “proficient in C++” sounds good on a job application.

Motivation

But why an emulator at all? Computers, even very simple ones, are complex machines. Generally, emulators are made over a long period of time or by a team of people working together – often both. Indeed, I knew going into this project that I would not have a totally complete emulator by the end of it. Even the CHIP-8 emulator, limited in size as it was, was not wholly finished by the time of its demonstration. It was important to me to get the basis of an emulator at least, however – something I could continue working on past the end date of this project. If I were able to make something that I could extend at some point in the future, I could potentially finish and release it at that point in time. Not only would this be cathartic in a way, it would also help satisfy the primary motivation for the project itself to have people using it to enjoy the games they love or experience new ones.

The primary motivation for my project has always been that of preservation. Computer software, and more specifically computer games, are serious cultural artifacts. They are also
ephemeral and delicate; hard discs break, data rots, and storage media fall into obsolescence. Unlike other, similarly delicate artifacts, however, there is often very little popular desire to preserve software for future generations. A painting can hang nigh-indefinitely in a museum, and anybody can take a picture of it to have their very own perfect replica. A complex video game may be infinitely replicable, but once the hardware it is built for obsolesces, nothing short of totally remaking it can bring it back. Of course, I would be remiss not to mention the fact that there is often a financial incentive for software to remain unpreserved. The companies who own the intellectual property rights of beloved retro games, for example, can leverage the scarcity and inaccessibility of those games to increase profits. They can sell merchandise rooted in nostalgia, remake the game on new machines with increased monetization (and often poor quality control), or force consumers to buy new entries in a game series instead of replaying old ones.

Thus, emulators enter the fray. Software left unpreserved, either intentionally or apathetically, is still enjoyed and desired by many. Emulators allow them to access that software where otherwise they might not be able to. DOSBox, for example, is an emulator for old DOS programs. Ever since Windows NT, DOS programs have been left mostly in the dust by first party support. DOSBox allows these programs to be run easily and accurately on modern hardware, preserving for future generations the majesty of such games as King’s Quest, Leisure Suit Larry, and Phantasmagoria (Veenstra et al., 2021). These are cultural and historical touchstones in the story of American video game culture, and just as any other such artifacts they reveal things about and connect us to our past. Without community preservation and emulation efforts, they would be lost to time as companies move on to new things. All of this does, unfortunately, raise the question of software piracy. Since it is possible to acquire illicit copies of
software, whether new or old, emulators can facilitate software piracy by making it easier for pirates to actually use their ill-gotten goods. Many of the more “professional” emulators do, however, take some level of anti-piracy efforts. Also, I can say that most users of popular emulators are not pirates, but are trying to actually access programs they paid for and no longer have the hardware for (although this is only anecdotal). In any case, my emulation is not meant to be released publically in its current state, so it takes no steps to prevent piracy.

I have been asked previously why I did not simply finish my CHIP-8 emulator for my senior project instead of moving onto an entirely new system. The answer to that lies in the motivation for this project, which I’ve mentioned already as being preservation. The thing about CHIP-8 is that it was very niche, even for its time, and notable games from the period are few and far between. It is now, and has been for most of its life, a toy system. It’s generally one of the first stops along the route of learning how to make an emulator, and most of its cultural production is from modern-day developers making little games for retro emulators. In saying all this, I do not mean to imply that silly or simple things are less worthy of preservation. What I do mean, however, is that I wanted to focus on a system with a greater and wider cultural impact. Whereas CHIP-8 was a fun and interesting bit of trivia, the Game Boy revolutionized handheld video gaming. It is still one of the top selling video game consoles of all time, both in its native Japan and internationally. Notably, it is only beaten out by the Nintendo Switch, Nintendo DS, and PlayStation 2.

**The Product**

What I have is not an emulation of the Nintendo Game Boy as intended, but of its CPU core. The link to its GitHub repository can be found in Appendix C. The CPU utilized by the
Game Boy is a modified version of the Sharp SM83, stripped of much of its 16-bit arithmetic capabilities (and a few other odds and ends). Nintendo did not make this readily accessible, and it has taken significant community effort over many years to identify the CPU and document its differences from its mother chip. In this way I owe a great debt to the Pan Docs (GBDev, 2023), which have been in continuous development since the 1990’s, the Complete Technical Reference (Javanainen, 2023), which explains in detail most of the CPU’s functions, and the table of opcodes maintained by the community (GBDev, 2022), helpfully color-coded and wonderfully laid out. These resources have been compiled through the efforts of many community members, involving reverse engineering software, physical testing of circuits, and endless research work.

As mentioned, my emulation is one of the Game Boy’s CPU. Although the original plan was to be somewhat exacting in this phase of the project, like many ideas I had, this was changed after an exceptionally rough winter break and semester start. I was going to maintain strict cycle accuracy, timing accuracy, and object-oriented notions of the various busses and chips that made up the Game Boy. Instead, I have a fairly high-level emulation of the CPU. It is not cycle- or time-accurate, though the latter would be relatively straightforward to implement if given more time. The goal of this emulation is to simulate the functions of the CPU in the way that the average end user might see them; that is to say, the various opcodes perform the correct arithmetic functions, memory manipulations, bit shifts, etc. Though I do not simulate the number of CPU cycles each instruction would take, what’s most important to this project is that these functions are working as intended otherwise.

How, then, does this emulator work? First, it takes as an argument a file path. This should point to a binary file containing legal opcodes and associated data for the CPU to read. The file is
read in and inserted byte-by-byte into a static array of length $2^{16}$, or 65,536 bytes. This serves as the notion of memory in my emulator. Once the array is filled, the emulator reads through each opcode and any associated data in order, performing the appropriate functions at each turn. It also outputs the memory address (in the emulator’s memory array) and instruction mnemonic of each operation. It terminates operation either at the end of the array or upon reaching a certain “stop” instruction. In terms of operation handling, there is very little difference between what I have here and what my completed Game Boy emulator would have looked like. The biggest difference is probably in file format.

In Game Boy games, the memory region $0x100-0x14F$ is reserved for the file header. This header contains information about the Game Boy hardware version, the memory cartridge version, the MBC (a switchable memory bank management system) if one is present, checksums, and some other things. There is also the built-in boot ROM in the region $0x000-0x0100$, which displays a start-up animation and little else. The boot ROM, having responsibility only for an animation which my emulator cannot play, is safe to ignore. The cartridge header, which deals with features my emulator does not have (being only a CPU and not the full Game Boy), is also safe to ignore. If one were to feed into my emulator a file with both of these things, it would operate fine, but it would read and operate on most of these two regions as if they contained valid instructions. In particularly unlucky circumstances, this could get the emulator stuck in an unintended loop (if, for example, it encountered a backwards-pointing JUMP of some kind). Though it would be nice to utilize both the boot ROM and the cartridge header, without a more complete emulator of the entire Game Boy system, they make no difference here.

**Technical Detail**
Game Boy Overview

The SM83 is, of course, only a part of the Game Boy. The whole system has more than just the one part. There is also the PPU, or Picture Processing Unit, which is the processor that manages video display output and shares responsibility of the VRAM. Then there is the APU, or Audio Processing Unit, which is predictably the processor that manages audio output. These three are all contained on one chip on the Game Boy’s motherboard, called (somewhat confusingly) DMG-CPU. Other parts include I/O ports for player input devices, connection ports for Game Boy peripherals (such as the printer and link cable), and data buses between the various chips.

Figure 1, to the left, is a simple diagram of the Game Boy system, not including memory allotments. In the rounded rectangle is the SM83, and in circles are other parts of the Game Boy itself. The double arrow between the PPU and the SM83 signifies their shared management of the VRAM, allowing them to talk to each other (so to speak). Though they share read/write access to this memory space, neither can access it simultaneously with the other. This
is controlled by which mode the PPU is in. There are four of these, two of which allow the CPU complete access to VRAM and one of which allows it partial access (Axelzon et al., 2021).

**Emulator Overview**

The emulator comprises three classes plus the main file. These classes are “memory,” “cpu_instr,” and “file_reader.” Figure 2, shown below, demonstrates the three classes, their member variables, and notable functions. Arrows between the classes signify an ownership relationship. For example, the “memory” class owns a copy of the “file_reader” class, so the latter has an arrow pointing to the former. For the sake of space and to avoid clutter, most functions have been omitted from this diagram. The only explicit exception to this is the “file_reader” class, which has no member variables and exactly one function. In this case, that function is listed. The “cpu_instr” has all of its functions condensed down to one line to avoid listing close to a hundred individual functions.

**Memory and Registers**

Currently, the emulator simulates system and cartridge memory, CPU registers, function calls, and the system stack. It also, as mentioned previously, reads through and acts on legal opcodes and attendant data. The system and cartridge memory are fairly straightforward. In my emulation, both are encapsulated in a single array which contains $2^{16}$ unsigned bytes, with no
differentiation between external or cartridge memory and local memory. This mirrors the memory map of the Game Boy, which this CPU would operate in. In that system, cartridge memory and local memory are contiguous in the memory map, even when a switchable memory bank is present. Because I do not simulate the entire Game Boy system and thus do not run Game Boy games, many of which (especially later in the console’s life) use such memory banks to circumvent the limited space available on the cartridges and system, that functionality is not currently implemented. The notion of memory in my emulation is static.

The CPU registers are somewhat interesting. They are not contiguous with the normal memory, of course, although in my emulation they are included in the Memory class rather than the CPU class. There are eight conventional registers in total, plus the stack pointer and the program counter. The eight registers are called A, F, B, C, D, E, H, and L, and they are each one byte long. The F register is the “flags” register and is normally not manipulated directly, except in a small handful of instructions which mostly have to do with pushing and popping from the stack. There are only four flags (Z, N, H, and C), which obviously does not fill an eight-bit byte, so the flags are stored in the upper nybble of F. H and L are the “high” and “low” registers respectively. This is because of a certain quirk of how the registers are handled; while they are literally one-byte registers, they are frequently treated as two-byte registers instead. For this purpose, they are joined in pairs: A and F become AF, B and C become BC, and so on. Since it contains the F register, AF is rarely treated this way, again mostly having to do with stack operations. HL is usually used to store a memory address for indirect arithmetic and addressing operations and jump instructions. Significantly, these double-wide registers are essentially big-endian, which is a break from the normally little-endian operations of the system. They are
treated as such by my emulation, in that any time the appropriate instruction is used the first 8-bit register in the pair is used as the high byte. Everything else, whether in cartridge memory, local memory, or the stack, behaves in a little-endian fashion.

Though I am not sure why this quirk is the way it is, as no special attention seems to be paid to it in the technical literature I have read, I can postulate a reason. Imagine a situation in which a developer wants to load the register pair HL with the address 0x0123. To do this, they would give the instruction to load the value into the register pair in the usual little-endian fashion (e.g. LOAD HL 0x23 0x01). Then, when they used the instruction to jump to the memory address stored in HL, it would also read the register pair in a little-endian fashion. Suddenly, instead of jumping to 0x0123 as intended, the program is now jumping to 0x2301! Clearly, this is not desirable. This could be solved by simply treating the load instruction as big-endian, but then the inconsistency is still there, just in a different place. In fact, it would then be on the developer to remember that this one instruction is big-endian, rather than abstracting that inconsistency away from their immediate attention. As it is, the developer simply has to treat the load instruction as they would any other instruction in the set, and the system handles the big-endian oddness for them.

**Function Calls and the Stack**

Function calls and the stack are also fairly straightforward. The stack is simply another array of unsigned bytes, this time only 64 bytes long. When pushing to the stack, 16-bit data is inserted little-end first. When performing a function call, the current program counter is pushed to the stack, and the program counter is then updated to the indicated address. Upon encountering a return instruction, the old program counter is popped from the stack in place of
the current one, plus an offset to prevent an endless call/return loop. The function call instruction

does not deal with function parameters. Any necessary parameters are stored by developers

either in the stack or in CPU registers as they see fit. Without developer intervention, the stack

would contain only memory addresses and nothing else. None of this is particularly surprising,

and it is very similar to previous Nintendo consoles and other contemporaries.

**Opcodes**

As mentioned previously, the entire list of opcodes and data is held in an array of

unsigned bytes. The main loop of the CPU class involves reading through and operating on

these. In order to do so, the Memory class has a `read()` function which returns the byte stored

at `memory[PC]` and increments the program counter. The `read()` function can be implemented

as a simple one-liner, such as this pseudocode:

```c
unsigned byte read() { return memory[PC++]; }
```

The resultant opcode is then used in a switch statement. There are just over 500 instructions, so

the switch statement that identifies and calls the appropriate function is quite lengthy. Depending

on the operation, `read()` is called again to retrieve any necessary data, and then the appropriate

function from the CPU class is called to perform the operation. Hopefully understandably, there

are not 500 or more unique functions in that class. The bulk of the opcodes are near-duplicates.

For example, the instructions in the region 0x0040-0x007f are exclusively `LOAD REGISTER

1, REGISTER 2`, with one exception at 0x0076. The function implemented to handle all of

these takes the exact same generic format (that is, the function takes the form of `load(reg1,

reg2)`). In the switch statement, each individual case simply fills in the appropriate register,

value, address, etc. for the specific opcode in question (e.g. `load(A, B)`).
Obviously, such a large switch statement is ugly, unwieldy, and annoying to debug. Other, more elegant solutions than that monster were considered. Namely, a table of function pointers and associated hash function, which would have mapped each opcode one-to-one onto its appropriate function. There are a couple of problems with that, however. The most fundamental, and probably most prohibitive, is that not all of the functions are of the same type. The majority of the functions are of the **void** return type. There are some essential ones, however (e.g. `pop()`) which are not of that type. Arrays in C++ would not allow for the inclusion of more than one type of function pointers. Additionally, those pointers did not play nice when I attempted to convert them all to **void** pointers. It’s possible that having multiple different tables and hash functions for the various different types of functions present in the emulation would solve that issue, but at that point the system becomes almost as unwieldy as the one it is meant to replace.

On the topic of opcodes, nearly all of them are implemented in this emulation. Perhaps this is unsurprising, considering that most of them take the form of **LOAD REGISTER 1, REGISTER 2** or **ADD REGISTER 1, REGISTER 2** or something of that kind. Notable exceptions include **STOP**, **HALT**, **EI**, and **DI**. The first two have to do with various system clocks, allowing for the cessation of computations at various levels, and the last two enable and disable interrupts respectively. Interrupts are not implemented at all in this emulation. The reason for this is that, as it stands, the emulation would not utilize them at all. Two primary uses of interrupts within the Game Boy are in graphics and player input. Interrupts are used in graphics to create novel visual effects by changing various sprite values mid-render. Input interrupts can be used to halt computation until the player takes an action, for example. Since neither player input nor graphics are a part of this emulation, interrupts are non-essential. The absence of interrupts
leaves the HALT instruction in somewhat of an odd place, which is why it is not implemented. HALT puts the CPU (and the Game Boy as a whole) into a “low-power mode” (GBDev, 2023) that includes stopping the system clock until an interrupt occurs. Without interrupts, the CPU would simply never resume. This is, of course, undesirable in an emulation of the CPU alone. As such, HALT is left unimplemented for the sake of the emulation.

This emulation is single-threaded. There are several use-cases for multithreading that arise in an emulation of the entire Game Boy system, and a few that arise even in just an emulation of the CPU. Namely, multithreading can be useful for maintaining timing accuracy. Sleep and wake functionality is a more elegant and accurate solution than, say, running a busy loop for however many milliseconds between emulated CPU cycles. The SM83 runs at about 4 MHz by default (8 MHz in the Game Boy Color’s double-speed mode). Obviously, modern processors outpace that by several orders of magnitude. Were I to continue on with this project, keeping stricter accuracy to the SM83 and perhaps emulating the rest of the Game Boy, this would be highly desirable. Graphics, especially, rely on consistent and precise timing in order to function correctly – one of the valuable lessons from the CHIP-8 emulation earlier in the project timeline. Specific to the Game Boy is the aforementioned PPU, which is notable because both it and the CPU can access VRAM, but not at the same time.

**Evaluation**

Many tests exist for Game Boy emulators which are thorough and very high quality. These tests, compiled by a member of the emulation community known only as “Blargg,” are widely used and generally accepted as exceedingly accurate. They are, in fact, the “gold standard” for Game Boy emulation (Victoria et al., 2020, p. 1). Part of the beauty of Blargg’s
tests are that they have been run on real, physical Game Boys and the results recorded publicly. Any emulator of the system which passes (and, in some cases, fails) those tests in the same way as the real system can be said to be highly accurate. Unfortunately, they rely on a complete Game Boy emulation in order to run. In fact, there are no publicly available test systems (as far as I have been able to find, anyway) for just the Sharp SM83 core. As such, I have had to mark out my own testing grounds. Though this leaves the accuracy of those tests in my own novice hands, I have been able to better tailor them to this specific project, and I carry no doubts as to their proper implementation and precision. For more information on the tests I wrote and carried out on the emulator, see Appendix A.

**Testing Process**

In order to verify the accuracy of my emulation, I have taken representative functions from each “grouping” of opcodes and tested those to see if they match expected outputs. For instance, instead of testing `ADD A, B, ADD A, C, ADD A, D`, and so on, I test only the first one. There is no meaningful difference, after all, between adding register B to register A and adding register C to register A. Where there is a minor but significant difference, e.g. between adding two registers and adding two registers with carry, representatives from both groups have been tested. Some opcodes are unique, and have been tested as such. Also, due to the nature of writing in Game Boy assembly, some secondary opcodes have also been used; as such, the rest of the tests only work if they do, functionally providing some level of implicit verification to them as well. Most of these secondary opcodes are in the LOAD group, as well as some logical and arithmetical operations.
There are 46 distinct groups of opcodes by my organization, although some of these are simply unique opcodes in no larger group. The groups are as follows, in ascending order by first opcode in the group: No Operation, Load Data to Register (16-bit), Increment Register (16-bit), Increment Register, Decrement Register, Load Data to Register, Rotate Register Left through Carry, Add (16-bit), Load Register from Address, Decrement Register (16-bit), Rotate Register Right through Carry, Stop, Rotate Register Left, Jump (relative), Rotate Register Right, Conditional Jump (relative), Load to Register from Address (increment), Complement Accumulator, Set Carry Flag, Complement Carry Flag, Load Register to Register, Add, Add with Carry, Subtract, Subtract with Carry, And, Exclusive-Or, Or, Compare Register, Conditional Return, Pop from Stack, Conditional Jump, Jump from HL, Conditional Function Call, Push to Stack, Function Return, Function Call, Load Data to Register from Offset, Add Signed Data to Stack Pointer, Shift Register Left, Shift Register Right (arithmetic), Swap Register, Shift Register Right (logic), Test Bit of Register, Set Bit of Register, Reset Bit of Register.

The annotated contents of the test file, including mnemonics and additional comments, can be found in Appendix A. It is my hope that the comments I have added will help those unfamiliar with Game Boy assembly to trace the program more easily. I have made note of all changes to the CPU registers as well as all changes to the flags. The latter are denoted with a capital letter representing their name followed by a lowercase “f” for “flag.” An astute observer may notice the use of the opcode 0xFD in this file, despite it being an illegal opcode. In order to make evaluation as simple and straightforward as possible, I have “hijacked” this illegal opcode for my own purposes. When this opcode is used, the emulator outputs the current contents of all of the CPU registers (except F), the current stack pointer, and finally the flags in their proper
order instead of the F register as a whole. Because I am not testing every individual operation, it is possible that some may produce unexpected results in practice. However, this would be from something like a typographical error, rather than the underlying function implementation. This is why a representative of each group of opcodes is tested, to ensure the underlying implementation is solid. Typographical errors are easy to make and easy to fix, and don’t reflect any sort of deeper misunderstandings or complex technical problems. As such, they are not particularly interesting or illuminating on the topic of the project as a whole.

A list of all opcodes (legal, implemented, or otherwise) can be found in Tables 1.1 and 1.2 in Appendix B. They are separated this way by convention and convenience. Table 1.1 contains all “normal” opcodes, while Table 1.2 contains all “prefixed” opcodes, which are opcodes that follow the normal opcode 0xCB. These prefixed opcodes are exclusively bit-level operations, such as rotate, set, and reset. Illegal opcodes have been left blank, except for 0xFD which has been bolded to indicate my use of it for testing purposes. Also bolded is STOP at 0x10; while this opcode remains mostly unimplemented, I do utilize it to stop the main loop of the interpreter at the end of the test file. Opcodes which appear directly in my tests, whether as primary or secondary focuses, have been represented in bold. Opcodes which do not appear directly in my tests, but which are represented by another opcode from their group, are placed in italics. Those which I have left unimplemented for whatever reason, those being DAA, HALT, EI, DI, and RETI, have been highlighted in gray.

Results

It was exciting to see these tests pass as handily as they did. As you can see, referring to the test file in Appendix A, I recorded expected values to be checked for at every test instruction.
For instance, at the second test instruction, the CPU registers were expected to contain the values 254, 240, 14, 1, 224, 255, and 224 - and they did. Furthermore, the flags (first seen at the third test instruction) were also entirely as expected. This is a pattern that held for all tests. While it is impossible for me to say that a real Game Boy would behave the same exact way (as I do not have one on hand), their expected behavior is derived from the meticulously-assembled community resources that drive essentially all Game Boy emulators. Specifically, these tests agree with the behavior put forward in the *Pan Docs* (GBDev, 2023) and in Joonas Javanainen’s *Complete Technical Reference* (2023). The one thing I don’t have is a dump of the entire memory. This is for a couple of reasons. First, it would be rather unwieldy to print out 65,536 bytes just to check if one or two are a certain value. Second, there are opcodes which serve that purpose better (namely, by loading any arbitrary memory value into a register).

**Discussion**

**Popularity**

The Game Boy is, again, quite a popular video game system. With such popularity comes a trade-off for a project like this. The Game Boy already has a bunch of popular emulators. Is another really necessary? The problem is that lesser-known (or poorer-selling) consoles do not have as much information available on their functionality. To compile all the useful information on a console such that a novice like myself can make one takes years of dedicated work from the community. Without knowledge of hardware, a console to test on, tools to test with, etc. I would not be able to even start that documentation process for a more obscure console. For instance, at one point I was going to emulate a handheld called the Neo Geo Pocket Color instead of the
Game Boy, but I simply could not find enough information about it. Limited by this lack of information, I chose to do a more accessible console instead.

That is the paradox faced by program preservationists of past and present. The less popular some piece of hardware is, the more important the preservationist’s job; yet, as the popularity of some piece of hardware declines, that very job becomes harder and harder. The foundation of all emulation work is a deep understanding of the hardware which it seeks to revive, to some extent, from the grave (or, probably more accurately, the landfill). That understanding only comes from the work of a whole host of very passionate and dedicated people who use their time and expertise to document these systems thoroughly and accurately.

Furthermore, I do think it is valuable to have many different emulations of a given system within the digital ecosystem. Just as biodiversity benefits a real life ecosystem, diversity of ideas and implementations is helpful to any community of developers and code contributors. Especially when done as a scholarly project, crucial choices are made about what to focus on and why. The most popular general-use Game Boy emulators seek to emulate as much of the system as possible and as accurately as possible, of course, but smaller projects tend to focus on one thing or another. It is undeniably good to have examples of all the different choices one might make when programming their own emulator. For instance, I follow a similar line of thinking to a group from the University of Gothenburg; that group prioritized improving the “overall impression” of their emulator rather than strict accuracy to the hardware (Axelzon et al., 2021, 45). Although their exact implementation of that idea is different from mine, as I am unfortunately not a group of six people, our idea is the same. Having these examples out there allows for cross-pollination and inspiration for future programmers.
Test File

Writing the test file was, in itself, an interesting experience. I do not have a compiler for the Game Boy assembly language, so I wrote it manually, opcode by opcode. Save a short project in one class a year ago, I have had no experience doing this sort of programming before. It is certainly more error-prone than writing in assembly language. I do, however, believe that working so close to the “metal” (as removed from the metal as it is, being an emulation) lends a sort of intimate knowledge of the simulated machine. It sells the illusion, if you will, of writing for the real thing; indeed, although I don’t have one available to me, with the proper formatting this code would run on the real thing.

The only drawback to writing my own test file is having to debug it - so many times I have typed a B where there should be an 8, or exchanged an F for a D. Nevertheless, I got it all cleaned up and running well, and had fun in the process. Were I to do this project over, a Game Boy assembly interpreter would be a great asset, and definitely worth the time investment to program if no extant interpreters served my needs well enough. Not only would it make writing the test file less error-prone, it would also speed up the whole testing process. In some ways, I think the popularity and simplicity of CHIP-8 may have blinded me to this issue in the beginning. With the absolute abundance of freely-available files on which to test that prototype, as well as the ease of getting it to a point at which they could be tested, I had no need to worry about writing my own tests.

Emulation Standards

What makes this emulation an acceptable one? How do these tests demonstrate the emulator is good? As previously stated, this emulation does not hold to the highest degree of
rigorous accuracy to the hardware. Its timing is not accurate at all in the sense of clock speed, since nothing is done to restrict monumentally fast modern processors to the much slower SM83. What makes this emulator good is that it accurately recreates the function of the CPU. In other words, if this emulator and the original CPU were given the same task, they would come back with the same answer. The focus has been on this metric because all others can be easily worked around. Modern hardware can be slowed and functions can be modified, but if the fundamental idea of the program is divorced from the original hardware, then it is hardly an emulation at all. By performing accurately to the functions of the original CPU, despite being inaccurate or incomplete in other details of the hardware emulation, the program is legitimized as an emulation of the specific hardware it is modeled on.

**Looking Forward**

Though I believe the current project stands well on its own merits, it could be integrated quite smoothly into an emulation of the entire Game Boy if I were to continue working on this project in the future. Part of the beauty of hardware like this is that each individual chip is a discrete unit which carries out (mostly) unique functions on its own. This both makes it ideal for object-oriented programming and quite easy to slot previous work and new work together with an API. It is easy to imagine a simulated data bus carrying sprite data requests from the PPU to the CPU, for instance, and returning with the necessary sprites. Player input could be as simple as hitting a few keyboard keys in place of the directional pad and face buttons of the Game Boy.

Of course, some necessary adjustments would have to be made to the existing project to bring it into a ready state for expansion. It does not currently simulate everything needed to accommodate the rest of the system, just what is necessary for the CPU to function. It would also
be desirable to bring the emulation into compliance with the stricter standards of accuracy discussed earlier, namely that of timing. Though, again, I think this project is satisfactory as is, bringing it in line with those expectations would make testing more straightforward and bring it closer to community standards. This would, naturally, be in conjunction with video output, player input, and (ideally) quality-of-life features such as audio output.

**Conclusion**

The emulation of the Sharp SM83 CPU core functions very well. Although it cannot be tested by the methods generally agreed upon by the Game Boy emulation community, it passes custom tests with flying colors. Every test performed exactly as expected, and there were no major problems to be solved during testing. As an emulation of the core of the Game Boy system, it could be smoothly expanded in the future, rendering it more complete and open to further testing and validation. By emulating a particular system, and a monumentally impactful system at that, this project is rooted in the history of computing and of video gaming. It reflects hardware limitations of the 1990’s as well as the rapid growth of computational power in the years since.
Appendix A

Initial Loads
In this first section, the first instruction is the “no operation” instruction. The following five instructions load certain values into registers A-E. These mostly serve to populate the registers with useful values for later instructions. The test instruction is included to check that the registers all contain the expected values, and that all of the flags are false. The comment after the test instruction shows the expected values.

NOP
LD A 0x19
LD B 0xF0
LD C 0x0E
LD D 0x01
LD E 0xE0

Push, Pop, and Addition
In this section, register A is assigned the value of register B. A logical “or” is then performed on register A (containing 0xF0) with register C (containing 0x0E), with register A being assigned the resulting value. The value of the register pair AF is then pushed to the stack, before being popped into the register pair HL. The value of the register pair DE (containing 0x01E0) is then added to HL. The ADD instruction is the first in the file to update the flags in any way, although in this case they all remain false.

LD A B   # A = 0xF0
OR C    # A = 0xFE
PUSH AF
POP HL    # HL = 0xFE00
ADD HL DE  # HL = 0xFFE0; Zf = 0, Nf = 0, Cf = 0

Increments and Decrements
Here, the register pair BC is loaded with the value 0xEE0F. The register pair is then incremented by one, followed by the register C on its own being incremented by one. The register D is then decremented by one, which is the first instruction in the file to change any flags to true (in this case, the zero flag Zf and the subtraction flag Nf). The register A is then loaded with the value at memory address 0xFF02, which is 0x19 in this case. Finally, the register pair HL is decremented by one.

LD BC 0xEE0F
INC BC    # BC = 0xEE10; Zf = 0, Nf = 0, Cf = 0
INC C    # C = 0x11; Zf = 0, Nf = 0, Cf = 0
DEC D    # D = 0x00; Zf = 1, Nf = 1
LDH A 0x02  # A = 0x19 (from address 0xFF02)
DEC HL    # HL = 0xFFDF; Zf = 0, Nf = 1
Jumps and Functions
This section mostly has to do with jump instructions and function calls. It opens with a conditional jump with a false condition, so no jump is actually performed. The registers H and L are then loaded with certain values in order to facilitate the next instruction, which is an unconditional jump instruction. This only jumps one byte forwards for simplicity’s sake. Next is a conditional function call which calls the block of code starting at SUB C (which subtracts the value of register C from A). After the subtraction is the instruction to set the carry flag Cf to true, which also has the effect of setting the subtraction flag to false. After this is a conditional return which checks that both the subtraction and carry flags are set. Since this is false, the return is not executed. The next instruction is an unconditional return. After this function is returned from, the relative jump instruction which follows the function call instruction is performed, jumping forwards four bytes (past the unconditional return). Finally, the value of register A is compared with register B, setting the subtraction and carry flags to true.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP Z 0x001F</td>
<td># Z is false, no jump</td>
</tr>
<tr>
<td>LD H 0x00</td>
<td></td>
</tr>
<tr>
<td>LD L 0x24</td>
<td></td>
</tr>
<tr>
<td>JP HL</td>
<td># jumps one instr forwards</td>
</tr>
<tr>
<td>CALL 0x0029</td>
<td># calls SUB C</td>
</tr>
<tr>
<td>JR 0x04</td>
<td># jumps past RET</td>
</tr>
<tr>
<td>SUB C</td>
<td># A = 0x08; Zf = 0, Nf = 1, Cf = 0</td>
</tr>
<tr>
<td>SCF</td>
<td># Nf = 0, Cf = 1</td>
</tr>
<tr>
<td>RET NC</td>
<td># N &amp; C is false, no return</td>
</tr>
<tr>
<td>CP B</td>
<td># Zf = 0, Nf = 1, Cf = 1</td>
</tr>
<tr>
<td>test</td>
<td># Exp: 0x08, 0xEE, 0x11, 0x00, 0xE0, 0xFF, 0xDF; Flags: 0, 1, 0, 1</td>
</tr>
</tbody>
</table>

Bit Rotations and Final Jumps
This section primarily deals with bit rotations, as well as finishing the jump and function call tests. The PREFIX instruction, which corresponds to opcode 0xCB, is necessary to access the bit shift, rotate, and set instructions. First, the value of register C, 0x11, is rotated right once. It is then rotated right again, this time through the carry flag. Register B is then assigned the current value of C in order to see it when the test instruction is carried out. Register C is then rotated left once, and then once more but through the carry flag. This is followed by the “complement carry flag” instruction which sets the subtraction flag to false and the carry flag to true. Then comes a conditional function call using the carry flag, which is now true, and it calls the code block starting at the second CCF. This sets the carry flag to false for the following SBC instruction. This is the “subtract with carry” instruction, and it subtracts the value of register A plus the carry flag from itself. Then comes the return instruction, and finally the conditional relative jump which jumps past the return instruction.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFIX</td>
<td></td>
</tr>
<tr>
<td>RR C</td>
<td># C = 0x08 (0b00001000); Cf = 1</td>
</tr>
</tbody>
</table>
PREFIX
RRC C  # C = 0x84 (0b10000100); Cf = 0
LD B C  # B = 0x84
PREFIX
RL C  # C = 0x08 (0b00001000); Zf = 0, Nf = 0, Cf = 1
PREFIX
RLC C  # C = 0x11 (0b00010001); Zf = 0, Nf = 0, Cf = 0
CCF  # Nf = 0, Cf = 1
CALL C 0x003E
JR NZ 0x03
CCF  # Nf = 0, Cf = 0
SBC A A  # A = 0; Zf = 1, Nf = 1, Cf = 0
RET
test
# Exp: 0x00, 0x84, 0x11, 0x00, 0x00, 0x00, 0x24; Flags: 1, 1, 0, 0

Final Loads and Arithmetic
This section tests the last groups of load and arithmetic instructions. First, the register A is loaded with the byte stored at the memory address contained in the register pair HL (0x0024). The value of HL is then incremented as part of the same instruction. Then register D is assigned the value of A in order to preserve that value for output. Register A is then loaded with the byte stored at memory address 0x0008. Register H is then assigned the value of A, again for the purposes of output. Register A is then assigned the value of itself plus H. The value of A is preserved one last time in register B, and finally those two are added together with the carry flag (which is currently false).
LD A (HL+)  # A = 0xCD, HL = 0x0025
LD D A  # D = 0xCD
LD A 0x0008  # A = 0x01 (from mem. addr. 0x08)
LD H A  # H = 0x01
ADD A H  # A = 0x02; Zf = 0, Nf = 0, Cf = 0
LD B A  # B = 0x02
ADC A, B  # A = 0x04; Zf = 0, Nf = 0, Cf = 0
test
# Exp: 0x04, 0x02, 0x11, 0xCD, 0xE0, 0x01, 0x25; Flags: 0, 0, 0, 0

Bit Operations and Final Logical Operations
This section finishes up the logical opcodes and does most of the bit opcode testing. First, register A is complemented, which is essentially an XOR with itself. Next, a logical and is performed between it and register D. Then, a logical exclusive-or is performed between it and register L. Register A is then shifted right arithmetically, then shifted right logically. Register A then has its high and low nybbles swapped, and finally it is shifted left.
CPL  # A = 0xFB (0b11111011); Nf = 1
AND D  # A = 0xC9 (0b11001001); Zf = 0, Nf = 0, Cf = 0
XOR L  # A = 0xED (0b11101101); Zf = 0, Nf = 0, Cf = 0
PREFIX
SRA A  # A = 0xF6 (0b11110110); Zf = 0, Nf = 0, Cf = 1
PREFIX
SRL A  # A = 0x7B (0b01111011); Zf = 0, Nf = 0, Cf = 0
PREFIX
SWAP A # A = 0xB7 (0b10110111); Zf = 0, Nf = 0, Cf = 0
PREFIX
SLA A  # A = 0x6E (0b01101110); Zf = 0, Nf = 0, Cf = 1
test
Exp: 0x6E, 0x02, 0x11, 0xCD, 0xE0, 0x01, 0x25; Flags: 0, 0, 0, 1

Final Bit Operations
This section finishes both the bit operations and the test as a whole. First, bit seven of register A is tested, which is currently one. Then, bit four of register H is set to one. Next, bit zero of register C is set to zero. Then one is added to the stack pointer, the test values are output, and the emulation is terminated by STOP.

PREFIX
BIT 7 A  # Zf = 0, Nf = 0
PREFIX
SET 4 H  # H = 0x11
PREFIX
RESET 0 C # C = 0x10
ADD SP 0x01 # SP = 0x01; Zf = 0, Nf = 0, Cf = 0
test
Exp: 0x6E, 0x02, 0x10, 0xCD, 0xE0, 0x11, 0x25; SP = 0x01
STOP
Appendix B

Table 1.1

<table>
<thead>
<tr>
<th>0x00</th>
<th>0x01</th>
<th>0x02</th>
<th>0x03</th>
<th>0x04</th>
<th>0x05</th>
<th>0x06</th>
<th>0x07</th>
<th>0x08</th>
<th>0x09</th>
<th>0x0A</th>
<th>0x0B</th>
<th>0x0C</th>
<th>0x0D</th>
<th>0x0E</th>
<th>0x0F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10</td>
<td>0x11</td>
<td>0x12</td>
<td>0x13</td>
<td>0x14</td>
<td>0x15</td>
<td>0x16</td>
<td>0x17</td>
<td>0x18</td>
<td>0x19</td>
<td>0x1A</td>
<td>0x1B</td>
<td>0x1C</td>
<td>0x1D</td>
<td>0x1E</td>
<td>0x1F</td>
</tr>
<tr>
<td>0x20</td>
<td>0x21</td>
<td>0x22</td>
<td>0x23</td>
<td>0x24</td>
<td>0x25</td>
<td>0x26</td>
<td>0x27</td>
<td>0x28</td>
<td>0x29</td>
<td>0x2A</td>
<td>0x2B</td>
<td>0x2C</td>
<td>0x2D</td>
<td>0x2E</td>
<td>0x2F</td>
</tr>
<tr>
<td>0x30</td>
<td>0x31</td>
<td>0x32</td>
<td>0x33</td>
<td>0x34</td>
<td>0x35</td>
<td>0x36</td>
<td>0x37</td>
<td>0x38</td>
<td>0x39</td>
<td>0x3A</td>
<td>0x3B</td>
<td>0x3C</td>
<td>0x3D</td>
<td>0x3E</td>
<td>0x3F</td>
</tr>
<tr>
<td>0x40</td>
<td>0x41</td>
<td>0x42</td>
<td>0x43</td>
<td>0x44</td>
<td>0x45</td>
<td>0x46</td>
<td>0x47</td>
<td>0x48</td>
<td>0x49</td>
<td>0x4A</td>
<td>0x4B</td>
<td>0x4C</td>
<td>0x4D</td>
<td>0x4E</td>
<td>0x4F</td>
</tr>
<tr>
<td>0x50</td>
<td>0x51</td>
<td>0x52</td>
<td>0x53</td>
<td>0x54</td>
<td>0x55</td>
<td>0x56</td>
<td>0x57</td>
<td>0x58</td>
<td>0x59</td>
<td>0x5A</td>
<td>0x5B</td>
<td>0x5C</td>
<td>0x5D</td>
<td>0x5E</td>
<td>0x5F</td>
</tr>
<tr>
<td>0x60</td>
<td>0x61</td>
<td>0x62</td>
<td>0x63</td>
<td>0x64</td>
<td>0x65</td>
<td>0x66</td>
<td>0x67</td>
<td>0x68</td>
<td>0x69</td>
<td>0x6A</td>
<td>0x6B</td>
<td>0x6C</td>
<td>0x6D</td>
<td>0x6E</td>
<td>0x6F</td>
</tr>
<tr>
<td>0x70</td>
<td>0x71</td>
<td>0x72</td>
<td>0x73</td>
<td>0x74</td>
<td>0x75</td>
<td>0x76</td>
<td>0x77</td>
<td>0x78</td>
<td>0x79</td>
<td>0x7A</td>
<td>0x7B</td>
<td>0x7C</td>
<td>0x7D</td>
<td>0x7E</td>
<td>0x7F</td>
</tr>
<tr>
<td>0x80</td>
<td>0x81</td>
<td>0x82</td>
<td>0x83</td>
<td>0x84</td>
<td>0x85</td>
<td>0x86</td>
<td>0x87</td>
<td>0x88</td>
<td>0x89</td>
<td>0x8A</td>
<td>0x8B</td>
<td>0x8C</td>
<td>0x8D</td>
<td>0x8E</td>
<td>0x8F</td>
</tr>
<tr>
<td>0x90</td>
<td>0x91</td>
<td>0x92</td>
<td>0x93</td>
<td>0x94</td>
<td>0x95</td>
<td>0x96</td>
<td>0x97</td>
<td>0x98</td>
<td>0x99</td>
<td>0x9A</td>
<td>0x9B</td>
<td>0x9C</td>
<td>0x9D</td>
<td>0x9E</td>
<td>0x9F</td>
</tr>
<tr>
<td>0xA0</td>
<td>0xA1</td>
<td>0xA2</td>
<td>0xA3</td>
<td>0xA4</td>
<td>0xA5</td>
<td>0xA6</td>
<td>0xA7</td>
<td>0xA8</td>
<td>0xA9</td>
<td>0xAA</td>
<td>0xAB</td>
<td>0xAC</td>
<td>0xAD</td>
<td>0xAE</td>
<td>0xAF</td>
</tr>
<tr>
<td>0xB0</td>
<td>0xB1</td>
<td>0xB2</td>
<td>0xB3</td>
<td>0xB4</td>
<td>0xB5</td>
<td>0xB6</td>
<td>0xB7</td>
<td>0xB8</td>
<td>0xB9</td>
<td>0xBA</td>
<td>0xBB</td>
<td>0xBC</td>
<td>0xBD</td>
<td>0xBE</td>
<td>0xBF</td>
</tr>
<tr>
<td>0xC0</td>
<td>0xC1</td>
<td>0xC2</td>
<td>0xC3</td>
<td>0xC4</td>
<td>0xC5</td>
<td>0xC6</td>
<td>0xC7</td>
<td>0xC8</td>
<td>0xC9</td>
<td>0xCA</td>
<td>0xCB</td>
<td>0xCC</td>
<td>0xCD</td>
<td>0xCE</td>
<td>0xCF</td>
</tr>
<tr>
<td>0xD0</td>
<td>0xD1</td>
<td>0xD2</td>
<td>0xD3</td>
<td>0xD4</td>
<td>0xD5</td>
<td>0xD6</td>
<td>0xD7</td>
<td>0xD8</td>
<td>0xD9</td>
<td>0xDA</td>
<td>0xDB</td>
<td>0xDC</td>
<td>0xDD</td>
<td>0xDE</td>
<td>0xDF</td>
</tr>
<tr>
<td>0xE0</td>
<td>0xE1</td>
<td>0xE2</td>
<td>0xE3</td>
<td>0xE4</td>
<td>0xE5</td>
<td>0xE6</td>
<td>0xE7</td>
<td>0xE8</td>
<td>0xE9</td>
<td>0xEA</td>
<td>0xEB</td>
<td>0xEC</td>
<td>0xED</td>
<td>0xEE</td>
<td>0xEF</td>
</tr>
<tr>
<td>0xF0</td>
<td>0xF1</td>
<td>0xF2</td>
<td>0xF3</td>
<td>0xF4</td>
<td>0xF5</td>
<td>0xF6</td>
<td>0xF7</td>
<td>0xF8</td>
<td>0xF9</td>
<td>0xFA</td>
<td>0xFB</td>
<td>0xFC</td>
<td>0xFD</td>
<td>0xFE</td>
<td>0xFF</td>
</tr>
</tbody>
</table>

A table showing all of the non-prefixed opcodes from 0x00 through 0xFF. Opcodes which have been tested directly in the test file are in bold, while those which have only been tested representatively are italicized. Unused opcodes are left blank. Unimplemented opcodes are highlighted in gray. Contiguous groups of like opcodes (e.g. 0x40 through 0x7F, the LOAD instructions) have no dividing lines between them.
A table showing all of the prefixed opcodes (meaning that each of these follows the opcode \(0xCB\)), which all have to do with bit operations (shifting, rotating, setting, etc). The same key holds as in Table 1.1.
Appendix C

The code for this project can be found at https://github.com/ib3655/brassard-ian-senior-project. The code for the CHIP-8 prototype is at https://github.com/ib3655/brassard-ian-chip8-emulator. The latter has no test file included, but there is an archive of modern games and animations at https://johnearnest.github.io/chip8Archive/?sort=platform, and a recommended file with which to test it is at https://johnearnest.github.io/chip8Archive/play.html?p=octojam1title.
Bibliography


