BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
ENGLISH SPECIALIZATION

DIPLOMA THESIS

Design, implementation and comparision of different software emulating methods

Supervisor
Lect. dr. Mihai Andrei

Author
Oniga Andrei-Mihai

2025

ABSTRACT

This thesis explores foundational software emulation techniques, with a particular focus on interpretation and just-in-time (JIT) compilation. The objective is to examine the trade-offs between these approaches in terms of performance and implementation complexity,

To evaluate these techniques in a controlled and educational context, interpreters and JIT compilers were implemented for two minimalistic and well-documented languages, especially well suited to beginners in emulation and thereby offering a comprehensive introduction to emulator design: *Brainfuck and CHIP-8*. They have been selected due to their simplicity, which enables the study to center on emulator behavior rather than language-specific intricacies.

The paper's analysis includes performance benchmarking, implementation complexity, and the impact of a selection of optimizations such as instruction folding, jump precalculation, hot loop replacement, and efficient register allocation in the case of JITs. Empirical results also demonstrate that JIT compilation significantly outperforms interpretation, particularly for longer-running programs, primarily due to reduced instruction-fetching and decoding overhead.

However, the study also highlights diminishing returns from aggressive optimization strategies such as excessive hot loop replacement which eventually leads to scenarios where entire programs are reduced to singular instructions, limiting further gains. Additionally, some code replacements were found to generate equivalent machine code to the original instruction sequences, rendering the optimization ineffective while increasing code complexity.

These findings underscore the balance between simplicity and performance, while aiming to offer accessible insights for readers new to emulation concepts, particularly those engaged in experimental or hobbyist development with languages like the two selected languages.

All components discussed have been implemented within a dedicated *C* application, whose architectural design and implementation details are thoroughly documented and analyzed. This practical framework serves to illustrate the theoretical concepts presented, providing a concrete basis for understanding the trade-offs and design decisions inherent in emulator development.

Note: Some of the paper's content has been rewritten and refined to enhance academic clarity and tone by an AI language model. All of the ideas and their implementations have been originally thought of and implemented by the author.

Keywords: computer engineering, software emulation, interpretation, JIT compilation, Brainfuck, CHIP-8, C, performance analysis, optimization techniques

Contents

1 Introduction

1.1 About

Emulators play a vital role in computing by enabling software to run on hardware or virtual platforms distinct from their original execution environments.

Their applications span a wide range of domains, including legacy video game preservation, processor simulation, the development of sandboxed runtime systems, and more.

Two fundamental techniques underpin most emulator implementations: interpretation and just-in-time (JIT) compilation.

A clear understanding of the strengths and limitations of these approaches is crucial for those engaging in low-level systems development or virtual machine construction.

This thesis offers an introductory exploration of interpretation and JIT compilation, supported by practical implementation and performance analysis.

To illustrate these techniques in a manageable and pedagogically valuable context, the study focuses on two minimal programming languages: Brainfuck and CHIP-8.

Brainfuck, a minimalist yet Turing-complete esoteric language, and CHIP-8, a simple virtual machine historically used for teaching early computing and game development concepts, serve as effective case studies due to their simplicity and well-defined behavior.

The primary objectives of this thesis are as follows:

- To implement both interpreters and JIT compilers for Brainfuck and CHIP-8.
- To explore and apply basic runtime optimizations.
- To compare the performance and complexity of each technique using a set of representative benchmark programs.
- To explore application design by creating a proper modularised architecture whith unit and / or integration tests.

The organization of the thesis is outlined as follows:

- **Chapter 2** examines the Brainfuck programming language, focusing on its implementation through both interpretation and just-in-time (JIT) compilation, along with a comparative analysis of their respective performance results.
- **Chapter 3** explores the historical evolution and variations of the CHIP-8 virtual machine, detailing the implementation strategies adopted.
- **Chapter 4** provides an in-depth analysis of the modular architecture of the main application, describing the integration and interaction of its components and the rationale behind key design decisions.
- **Chapter 5** summarizes the core findings of the study, discusses its limitations, and proposes potential directions for future research and development in the field of software emulation.

1.2 Related work

The programming languages analyzed in this study, have seen limited engagement in academic literature, likely due to Brainfuck's intentionally provocative nomenclature and CHIP-8's roots in hobbyist computing, far from academia.

Nevertheless, both languages have found specific applications in research and education. CHIP-8 has been employed as an instructional tool for teaching computer architecture and virtual machine concepts in undergraduate curricula, demonstrating its relevance in contemporary pedagogy despite its retro origins [1].

Brainfuck, on the other hand, has appeared in more niche academic contexts. It has been discussed in the broader framework of esoteric and conceptual programming languages [3, 7], explored in parallel hardware design implementations [5], used in preliminary studies on self-interpreting languages [2], and even utilized as a domain for reinforcement learning in program synthesis and compiler fuzzing research [8].

These contributions, while scattered, highlight the potential of these languages as compact and expressive models for exploring core principles in interpretation, compilation, and system design.

2 Brainfuck

Brainfuck (commonly abbreviated as BF in academic literature) is an esoteric programming language created in 1993 by Swiss computer science student Urban Müller [4]. It was conceived as a minimalist language that challenges conventional programming approaches by reducing syntactic elements to the bare essentials. The language's primary intent is not practical software development, but rather to serve as a vehicle for exploring computational theory, language design, and the boundaries of human-readable code.

Brainfuck operates on a simple model of computation: a one-dimensional array of values initialized to zero, and a single data pointer that traverses this memory. Loops in Brainfuck are defined by matching `[` and `]` brackets, executing the enclosed code as long as the current cell is non-zero. The language lacks named variables, functions, or high-level abstractions, requiring the programmer to construct all control flow and data manipulation manually.

Despite its extreme simplicity, Brainfuck is Turing-complete. This implies that, in theory, any computable function can be implemented using Brainfuck, provided unlimited memory and time. The Turing-completeness of Brainfuck underscores its value as a pedagogical tool, illustrating how a minimal set of operations can express any arbitrarily complex computation.

The language's deliberately obfuscated syntax forces programmers to think at the level of memory and instruction cycles, drawing parallels with low-level systems programming, particularly in assembly or machine code. This requirement to engage directly with memory manipulation and flow control makes Brainfuck a unique platform for studying program optimization, instruction translation, and interpreter design.

In contemporary settings, Brainfuck is primarily used for educational and experimental purposes. It remains a popular subject in programming contests, obfuscation challenges, and academic discourse within the esoteric programming language (esolang) community. Its design exemplifies the power of minimalism in programming language theory and continues to inspire discussions on language expressiveness, compiler construction, and the boundaries of syntactic design.

2.1 Machine specification

The language is defined by an extremely compact instruction set consisting of only eight commands:

+	Increments the value at the current position that the machine points to.
-	Decrements the value at the current position that the machine points to.
<	Moves the pointer one cell to the left.
>	Moves the pointer one cell to the right.
[	Jumps after the corresponding closed bracket when the value at the current cell is 0.
]	Jumps after the corresponding open bracket when the value at the current cell is not 0.
.	Outputs the value at the current cell.
,	Read a value to be placed at the current cell.

Table 2.1: Brainfuck commands and their descriptions.[4]

The size and number of memory cells in the language are not strictly defined and are typically left to the discretion of the programmer. In the context of this implementation, a memory model consisting of *65,536* cells was adopted (corresponding to the addressable range of a 16-bit pointer) with each cell represented as an unsigned 8-bit value.

2.1.1 Programming in the language

To facilitate a practical understanding of the Brainfuck language and its idiomatic constructs, let's examine a well known instruction sequence that outputs the string *'Hello World!'* to the screen:

```
1  >++++++++[<+++++++++>-]<.
2  >++++[<+++++++>-]<+.
3  +++++++..+++.
4  >>++++++[<+++++++>-]<++.
5  ------------.
6  >++++++[<+++++++++>-]<+.
7  <.
8  +++.
9  ------.
10 --------.
11 >>>++++[<++++++++>-]<+.
```

For clarity, the code has been formatted so that each line corresponds to the generation of a single character in the output. This organization aids in demonstrating how Brainfuck instructions directly map to output generation.

Consider the first line:

```
>++++++++[<+++++++++>-]<.
```

This sequence first increments the second memory cell (cell #1) to the value 8. It then enters a loop that decrements cell #1 on each iteration and simultaneously increments the third cell (cell #2) by 9 per iteration. Once cell #1 reaches zero, the loop terminates, and cell #2 holds the value 72, corresponding to the ASCII code for 'H'. The final instruction outputs this value.

Similar patterns are applied across subsequent lines to construct the remaining characters of the phrase. Characters such as *'l'*, which appear multiple times, are output using consecutive `.` commands. The *'o'* from *"Hello"* is reused in *"World"*, demonstrating efficient reuse of computed values.

This example illustrates common patterns and idioms in Brainfuck programming. One foundational construct is the multiplication loop:

```
(any number of +)[(movement)(any number of +)(reverse movement)-]
```

This structure multiplies the initial value of a cell by the number of additions performed inside the loop, effectively replicating multiplication via repeated addition.

Another fundamental idiom is the cell-clearing pattern, which resets the current cell's value to zero:

```
[(- or +)]
```

Additionally, a commonly used construct involves transferring the value of one cell to another:

```
[(pointer movement)+(reverse pointer movement)-]
```

This operation assumes that the destination cell is initially zero. If this is not the case, the original value will be added rather than overwritten. To enforce a pure move, the destination cell should be cleared first using the aforementioned clearing pattern.

These patterns provide valuable insights into the low-level logic of Brainfuck programs and are instrumental in designing optimized interpreters.

2.2 Emulator implementation

Because of the language's simplicity, there have been many emulators made for it in all kinds of programming languages. As such, the focus will be put on the techniques utilised in the creation of optimized emulators rather than the emulators themselves, as they have been a rather exhausted subject.

2.2.1 Simple interpreter

A basic Brainfuck interpreter functions by sequentially parsing each character of the input program and executing the corresponding operation on a simulated memory model.

This model typically comprises a data array, initialized to zero, alongside a data pointer that tracks the currently active cell. As the interpreter reads the source code, it maps each command to a specific operation: incrementing or decrementing the cell's value (+ or -), shifting the data pointer left or right (¡ or ¿), reading a byte of input (,), or outputting the current cell's value (.).

Control flow in Brainfuck is handled through loop constructs denoted by the [and] characters. When encountering a [symbol, the interpreter checks the value of the current cell. If the value is zero, execution jumps forward to the instruction immediately following the corresponding closing bracket]. Conversely, when a] is reached and the current cell is non-zero, execution jumps backward to the matching opening bracket [.

To enable efficient execution of these control flow constructs, the interpreter usually performs a preprocessing step that generates a mapping between matching bracket pairs, allowing for constant-time jumps during interpretation.

The entire execution logic is structured as a large `switch`-based dispatch mechanism, which simplifies the implementation and provides a modular structure that facilitates later extensions, particularly in the context of optimization.

Preliminary performance evaluations for this execution model indicate the following average times after *5* runs:

Program	**Instruction Count**	**Execution Time (ms)**
mandlebrot.b	11452	121035
hanoi.b	53885	92000
alphabet.b	186	8000
squares.b	197	18.7
sierpinski.b	125	3.7

Higher times have been rounded as the error is small enough to be negligible.

2.2.2 Static compilation

Choosing the code emmiting library

In selecting a library for runtime machine code generation, several options were evaluated, including *libjit*, *asmjit*, and *dynasm*.

While these libraries offer varying levels of functionality and abstraction, they presented challenges in terms of integration and control. Specifically, *libjit* and *asmjit* were found to be non-trivial to use within the context of this project, as *asmjit* was intended for use in C++, not C and *libjit* was proven to be outdated and not supported anymore.

Also *dynasm* introduced its own post-processing phase in the compilation of the program and in the compilation of the Brainfuck program as well, the second of which effectively negated custom optimizations performed at the code generation level, as similar transformations were already applied by its backend.

Ultimately, *GNU Lightning* was chosen for its simplicity, minimal overhead, and direct mapping between the emitted instructions and the final machine code. Its lightweight nature and relative portability across platforms made it a suitable choice for this project, particularly in this scenario which requires fine-grained control over the emitted code.

GNU Lightning exposes a platform-agnostic register abstraction layer through a set of symbolic registers that represent underlying physical registers on the target architecture. These include general-purpose registers, denoted as *R0* to *R2* in `x86`, which are typically used for integer operations and control flow. Additionally, it provides volatile or temporary registers, such as *V0* to *V4*, which are typically used for transient calculations and are not preserved across function calls.

For handling floating-point operations, it defines a separate set of registers from *F0* to *F7*. These represent floating-point hardware registers and are intended for operations involving real numbers.

GNU Lightning implementation

Basic operations such as memory incrementation (+), pointer movement (>, <) are directly mapped to *GNU Lightning* instructions, such as add *jit_addi* which adds an immediate value to a register, or *jit_ldr_c* / *jit_str_c* which loads / stores an 8-bit register to an address in memory.

I/O functions (`.` and `,`) are mapped to a function call of their respective pointers from the state, and for input, the returned result is stored in *JIT_R0* which is then stored in memory.

Control flow constructs (`[`, `]`) are handled using label stacks and conditional branches, with jumps managed via preallocated loop label arrays.

Register usage is explicit, relying on general-purpose virtual registers (*JIT_R0* and *JIT_R1*) and pointer management through *JIT_V0*, which holds the memory base.

The JIT state is finalized with a prolog and epilog, and the emitted function pointer is stored for later execution.

Program	Instruction Count	JIT Time (ms)	Time % of interpreter
mandlebrot.b	11452	3800	3.13%
hanoi.b	53885	4830	5.25%
alphabet.b	186	745	9.31%
squares.b	197	0.99	5.29%
sierpinski.b	125	0.25	6.75%

A big reduction in execution time can be already observed, but there's still a lot of room for improvements to be done.

2.3 Applying optimizations

2.3.1 Precalculating jumps

In the Brainfuck language, control flow is implemented using the loop constructs `[` and `]`, which form conditional jump instructions based on the value at the current memory cell. A naive implementation might process these instructions dynamically by scanning forward or backward through the code to locate the corresponding matching bracket whenever a jump is needed.

At runtime, if the condition is met, the interpreter may search forward to find the matching `]`, or backward to locate the corresponding `[`.

While functionally correct, this approach is highly inefficient, especially in programs with nested or frequent loops, as it introduces linear-time overhead on each loop entry and exit.

Precomputed Jump Table Optimization

To mitigate this inefficiency, jump destinations can be precalculated during the parsing phase of the program. This transforms dynamic control flow resolution into a constant-time operation, enabling the use of a more efficient intermediate representation.

The revised instruction format introduces a single unified jump operation:

- **JMP** x: Performs a conditional jump of x instructions relative to the current position in the code.

  - If x is **positive**, the instruction corresponds to a `[` and is executed **only if** the value at the current memory cell is **zero**.
  - If x is **negative**, the instruction corresponds to a `]` and is executed **only if** the value at the current memory cell is **non-zero**.

This structure allows for efficient execution by directly referencing the jump destination, avoiding the need for runtime matching of bracket pairs.

2.3.2 Instruction folding

Another fundamental and immediate optimization in Brainfuck program analysis involves the elimination of redundant sequences of repeated instructions as Brainfuck source code often includes consecutive repetitions of basic commands, such as:

```
+++++++++   // Increment the current cell by 9
>>>>>>      // Move the pointer 6 cells to the right
```

These repetitive patterns can be consolidated to improve both interpretive efficiency and give more performant emmited code by the JIT compiler.

2.3.3 Sequence matching for common patterns

Transformation to Intermediate Representation

To address this redundancy, a simplified intermediate set instruction is introduced. It aggregates repeated operations into parameterized instructions, more specifically, the following transformations are applied:

- **ADD x**: Replaces sequences of `+` and `-` instructions. The signed integer x indicates the net change to the value at the current cell.
  - Example: `+++++` becomes `ADD 5`, and `---` becomes `ADD -3`.
- **MOV x**: Replaces sequences of `>` and `<` instructions. The signed integer x represents the net change to the data pointer position.
  - Example: `>>>` becomes `MOV 3`, and `<<` becomes `MOV -2`.

These transformations serve two primary purposes:

1. **Performance Enhancement**: Interpreters and compilers can process compact instructions more efficiently than long sequences of primitive commands.
2. **Readability and Analysis**: The intermediate representation is more succinct and expressive, facilitating further optimization and program analysis.

Regex-like Pattern Recognition in Parsing

The code parser operates in a stack-like fashion:

1. Read the current instruction.
2. Determine its type.
3. If it differs from the top of the stack, push it as a new entry.
4. If it matches the type, attempt to merge or extend the existing instruction.

This parsing mechanism can be extended to support loop pattern recognition: after each new instruction is added to the stack, the parser inspects the top of the stack for known patterns (such as `3j xa -1j`).

If a match is found, the sequence is replaced with the appropriate high-level instruction, such as `CLR`.

Consider the Brainfuck loop:

```
[-]
```

Semantically, this loop is equivalent to setting the current memory cell to zero. It repeatedly decrements the cell until it reaches zero, and the loop exits. Despite its simplicity, the execution time of this loop is proportional to the value of the cell, which may range up to 255 (in the case of 8-bit cells). Thus, the runtime cost of this idiom can be substantial.

Replacing Common Loops with Custom Instructions

To address this inefficiency, a specialized instruction can be introduced into our intermediate representation:

- **CLR**: Sets the value at the current memory cell to zero.

This transformation avoids executing multiple decrement operations and conditional jumps, replacing them with a single deterministic operation.

Identifying Loops Worth Optimizing

Not all loops merit optimization; hence, we employ a heuristic based on dynamic profiling to identify high-frequency loops.

During an initial instrumentation pass of the program, we count the number of executions of each top-level loop. Each loop is recorded with the following format:

For example, in the `mandlebrot.b` program, the following raw data can be observed (`j` - jump, `a` - add, `m` - move):

```
number of executions | sequence
157090277 6j -1a 9m 1a -9m -4j
46993495  3j -1a -1j
25555337  6j -1a -2m 1a 2m -4j
...
```

Structural patterns can now be easier to observe, for instance, the pattern:

```
3j xa -1j
```

(where `x` is any non-zero constant) represents any loop of the form:

```
[-]  [+]  [++++]  [---]
```

Each of these is semantically equivalent to clearing the current cell. As such, they can be replaced by the custom `CLR` instruction in the intermediate representation.

Program	Size	Interpreter (ms)	Interp as JIT%	% of original
mandlebrot.b	2915	14390	1326.26%	11.89%
alphabet.b	80	1177	3138.66%	14.72%
hanoi.b	9830	1010	2927.55%	1.10%

Program	Size	JIT (ms)	JIT as Interp%	% of original
mandlebrot.b	2915	1085	7.53%	28.55%
alphabet.b	80	37.5	3.19%	5.03%
hanoi.b	9830	34.5	3.42%	0.715%

These results demonstrate that the JIT offers a significant performance advantage over the interpreter, primarily due to the elimination of instruction fetching and decoding overhead and that the applied optimizations have a substantial impact on the execution time across the tested programs.

2.4 Testing

The test runner initializes the testing sequence by invoking a series of focused unit tests: one that verifies the correct setup of the memory state, one that checks the fidelity of source code parsing and transformation, and others that validate the interpreter's initialization and execution logic.

For builds configured with *GNU Lightning* support, the test suite conditionally includes one that tests the initialization of the JIT backend and one that verifies the execution of JIT-compiled code.

3 CHIP-8

3.1 About

CHIP-8, sometimes spelled as CHIP8, is a programming language and virtual machine specification developed by Joseph Weisbecker on the 1802 processor of the COSMAC VIP computer in the mid-1970s.

It was meant to be an educational tool mainly designed around creating simple video games with much more ease and less resources than conventional programming languages of the time such as BASIC.

Even today it is widely used as an introduction for people that are taking up software emulation as a programming hobby because of its simplicty and ease of implementation.

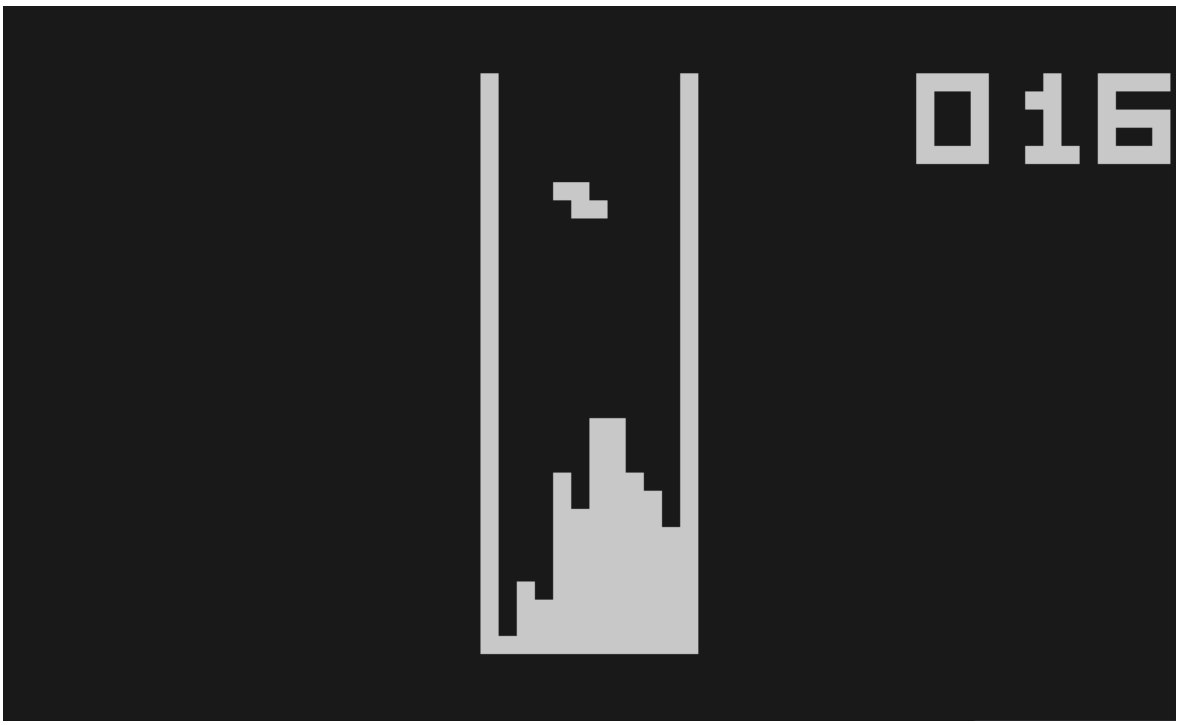

Figure 3.1: Tetris clone written in CHIP-8 by Fran Dachille in 1991 running in `Edra`.

3.2 Virtual machine description

Memory

The CHIP-8 virtual machine provides *4KB* of accessible memory, reflecting the addressable range of its 16-bit memory registers. An exception to this is the `I` register, primarily utilized for drawing and memory operations, which is implemented as a 12-bit register rather than 16-bit. Multi-byte data within the system is stored in big-endian format, meaning the most significant byte precedes the least significant byte in memory.

Historically, the virtual machine's memory space overlapped with that of the interpreter, resulting in programs typically being loaded starting at address `0x200`. In contemporary implementations, this memory region is often repurposed to store font data, as emulators maintain separate memory spaces for code and data.

Furthermore, the uppermost *256* bytes of memory (addresses `0xF00`-`0xFFF`) were originally allocated for display data. The adjacent lower segment (addresses `0xEA0`-`0xEFF`) was reserved for the interpreter's internal use, including the call stack and various system variables, though this reservation is frequently disregarded in modern emulator designs where the display data and emulator implementation is stored in a separate memory location.

Registers

The CHIP-8 virtual machine contains *16* general-purpose 8-bit registers, labeled `V0` through `VF`. The `VF` register is reserved for use as a flag by certain instructions and should generally be avoided for storing general data, as its value may be altered implicitly during execution.

Display

The CHIP-8 display operates at a resolution of `64x32` pixels, rendering graphics in monochrome. Sprites are rendered onto the screen using a bitwise XOR based drawing method, whereby the sprite pixels are bitwise XORed with the existing framebuffer content. This technique allows for simple sprite toggling and erasure.

Additionally, the interpreter sets the flag register `VF` to *1* if any pixel collision occurs during this process, specifically when a pixel is turned off as a result of overlapping pixels, providing a mechanism for collision detection in games and applications.

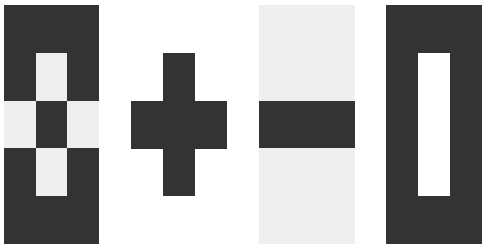

Table 3.1: Illustration of sprite XOR-ing, transforming an 8 (left) into a 0 (right), `VF` is set in this case to *1*. The + sign is not part of the sprite.

The CHIP-8 virtual machine includes two timers: the delay timer and the sound timer. Both decrement at a fixed rate of *60hz*. While the delay timer is typically used for timing-related logic within programs, the sound timer serves as an audio trigger. When the sound timer holds a non-zero value, it indicates that a sound should be produced.

The exact characteristics of the sound are not formally specified in the original CHIP-8 documentation, leaving its implementation to the discretion of the emulator developer. The most commonly adopted convention is to emulate the sound as a simple buzzer tone, consistent with the limitations of the original hardware.

Input

Input on the CHIP-8 system is handled via a hexadecimal keypad consisting of *16* keys, labeled with values ranging from *0* to *F*.

A common convention is to arrange the keys in a 4 by 4 grid that mirrors the original layout, allowing intuitive input translation between physical keyboards and the virtual machine environment:

1	2	3	C
4	5	6	D
7	8	9	E
A	0	B	F

Table 3.2: Original input layout on the COSMAC VIP

1	2	3	4
Q	W	E	R
A	S	D	F
Z	X	C	V

Table 3.3: Commonly used layout on modern keyboards

Instructions [6]

Opcode	Mnemonic	Description
00E0	CLS	Clear the display.
00EE	RET	Return from a subroutine.
1NNN	JP addr	Jump to location NNN.
2NNN	CALL addr	Call subroutine at NNN.
3XNN	SE Vx, byte	Skip next instruction if Vx is equal to NN.
4XNN	SNE Vx, byte	Skip next instruction if Vx is not equal to NN.
5XY0	SE Vx, Vy	Skip next instruction if Vx is equal to Vy.
6XNN	LD Vx, byte	Set Vx = NN.
7XNN	ADD Vx, byte	Set Vx = Vx + NN.
8XY0	LD Vx, Vy	Set Vx = Vy.
8XY1	OR Vx, Vy	Set Vx = Vx OR Vy.
8XY2	AND Vx, Vy	Set Vx = Vx AND Vy.
8XY3	XOR Vx, Vy	Set Vx = Vx XOR Vy.
8XY4	ADD Vx, Vy	Set Vx = Vx + Vy, set VF = carry.
8XY5	SUB Vx, Vy	Set Vx = Vx - Vy, set VF = NOT borrow.
8XY6	SHR Vx , Vy	Set Vx = Vx SHR 1.
8XY7	SUBN Vx, Vy	Set Vx = Vy - Vx, set VF = NOT borrow.
8XYE	SHL Vx , Vy	Set Vx = Vx SHL 1.
9XY0	SNE Vx, Vy	Skip next instruction if Vx is not equal to Vy.
ANNN	LD I, addr	Set I = NNN.
BNNN	JP V0, addr	Jump to location NNN + V0.
CXNN	RND Vx, byte	Set Vx = random byte AND NN.
DXYN	DRW Vx, Vy, nibble	Display n-byte sprite starting at memory location I at (Vx, Vy), set VF = collision.
EX9E	SKP Vx	Skip next instruction if key with the value of Vx is pressed.
EXA1	SKNP Vx	Skip next instruction if key with the value of Vx is not pressed.
FX07	LD Vx, DT	Set Vx = delay timer value.
FX0A	LD Vx, K	Wait for a key press, store the value of the key in Vx.
FX15	LD DT, Vx	Set delay timer = Vx.
FX18	LD ST, Vx	Set sound timer = Vx.
FX1E	ADD I, Vx	Set I = I + Vx.
FX29	LD F, Vx	Set I = location of sprite for digit Vx.

Opcode	Mnemonic	Description
FX33	LD B, Vx	Store BCD representation of Vx in memory locations I, I+1, and I+2.
FX55	LD [I], Vx	Store registers V0 through Vx in memory starting at location I.
FX65	LD Vx, [I]	Read registers V0 through Vx from memory starting at location I.

The original CHIP-8 implementation supported a total of 35 instructions, although only 31 were explicitly documented in the initial specification. Notably, opcodes such as `8XY3`, `8XY6`, `8XY7`, and `8XYE` were omitted from the formal documentation.

This omission is likely attributable to the historical context of CHIP-8's design, which was implemented atop the RCA 1802 microprocessor. The `8XY*` instruction group closely mirrors operations available in the 1802's Arithmetic Logic Unit (ALU), suggesting that these opcodes may have originated from the hardware's native capabilities rather than being explicitly defined by the CHIP-8 interpreter itself.

Subsequent extensions to the CHIP-8 instruction set, such as those introduced in SCHIP (Super CHIP) and other variants, have redefined or augmented the behavior of certain instructions. For example, instructions `FX55` and `FX65` originally incremented the index register `I` after each memory write or read, respectively. However, later implementations, including SCHIP, omit this side effect, reflecting divergence in interpretation and a lack of standardization across extensions, which will be discussed later.

3.3 Emulator implementation

The core CHIP-8 library, `cchip8`, is organized into modular components: the state, the interpreter, and the static compiler. This architecture enforces a clear separation of concerns, where only the execution runners depend on the state module, and each module operates independently, promoting maintainability and extensibility.

The state module, defined in `state.h`, functions as the central interface between the emulator core and the external environment. It encapsulates the full virtual machine state as well as a set of callback function pointers that enable host-controlled behavior, such as memory access, rendering, input handling, and random number generation.

The `chip8_state`'s structure maintains essential CHIP-8 runtime components, including program counter (`pc`), index register (`i`), stack and stack pointer (`stack`,

`sp`), *16* general-purpose registers (`v[0x10]`), timers (`dt`, `st`), display resolution parameters, and font memory addresses.

In addition, it includes a flag `draw_flag` for screen redraw signaling and a variable `last_key` to track the most recently pressed key to emulate accurate behaviour for the *FX0A* instruction family.

The embedded function pointers (`read_b`, `read_w`, `write_b`, etc.) abstract interactions with the host system. These allow the emulator to delegate low-level operations such as I/O, drawing, screen clearing, input polling, and random number generation to external implementations, facilitating portability and customization.

An auxiliary argument pointer (`aux_arg`) is also provided to pass user-defined data to these callbacks, supporting more complex execution contexts if needed.

```
struct chip8_state
{
    bool draw_flag;
    chip8_run_mode_t mode;
    uint16_t pc, i, stack[0x100];
    uint8_t display_width, display_height;
    uint8_t v[0x10], sp, dt, st, last_key;
    uint16_t lowres_font_address, hires_font_address;

    // Callbacks.
    chip8_read_b_f read_b;
    chip8_read_w_f read_w;
    chip8_write_b_f write_b;
    chip8_draw_sprite_f draw_sprite;
    chip8_clear_f clear_screen;
    chip8_key_status_f get_key_status;
    chip8_random_f get_random;
    chip8_resize_f resize;
    chip8_scroll_f scroll;

    void* aux_arg;  // Used for passing data to the callbacks.
};
```

3.3.1 Interpreter

The interpreter, as defined in `cpu/interpreter.h`, encapsulates a minimal runtime structure responsible for controlling instruction execution.

The `chip8_interpreter_t` struct maintains three key fields: a boolean flag `running` to indicate whether the interpreter is actively executing instructions, a

`long` field `timer` to track internal timing mechanisms, and a pointer `state` to the shared `chip8_state_t` structure, which contains the core emulator state, mentioned above.

```
typedef struct chip8_interpreter
{
    bool running;
    long timer;
    chip8_state_t* state;
} chip8_interpreter_t;
```

This design ensures that the interpreter remains lightweight, delegating system-wide responsibilities to the state module while managing control flow and timing locally, and such, enables the interpreter to operate independently of the underlying platform or user interface, facilitating reuse and testing.

The main function `chip8_interpreter_step(chip8_interpreter_t* self)` is responsible for executing a single instruction cycle within the interpreter. It represents the core of the CHIP-8 execution loop, fetching the current instruction from memory, decoding it, and dispatching it for execution.

Internally, instruction decoding is implemented using a large `switch` statement, a common pattern in interpreter design, especially for simpler architectures. The instruction opcode, 16 bits in CHIP-8, is fetched from memory using the program counter (`pc`) and then used to determine which operation to execute.

Each `case` in the `switch` corresponds to a specific opcode or opcode pattern, allowing for structured, readable, and performant instruction handling.

This dispatch method, while straightforward, also benefits from modern compiler optimizations that often convert large `switch` statements into jump tables, providing relatively fast execution times.

Additionally, this layout makes it easier to maintain and extend the interpreter, as new opcodes can be added with minimal structural disruption.

3.4 Quirks and extensions

3.4.1 Modern Super CHIP-8

Figure 3.2: Octogon by John Earnest running in `Edra`'s modern SCHIP mode.

The Super-CHIP (SCHIP) extension, introduced by Erik Bryntse in the 1990s, builds upon the original CHIP-8 virtual machine to enhance its graphical capabilities and improve overall program flexibility.

The most notable addition is the expansion of the display resolution from `64x32` pixels to `128x64` pixels, enabling more detailed graphics.

SCHIP also introduces a set of extended instructions[6], particularly for drawing and scrolling operations, as well as instructions to modify the behavior of the display or interpreter, the ones implemented in `Edra` are as follows:

Opcode	Description
00CN	Scroll display down by `n` pixels.
00FB	Scroll display right by 4 pixels.
00FC	Scroll display left by 4 pixels.
00FD	Exit the interpreter (halt).
00FE	Enable low-resolution mode (64x32).
00FF	Enable high-resolution mode (128x64).
DXY0	Draw 16x16 sprite in high-resolution mode. In low-res mode, behaves like standard DXYN.

3.5 Testing

The current implementation has been tested for correctness using `Timendus' CHIP8 test suite`, a well known test suite in the community. It assures correct behaviour, identical to the original implementation unless otherwise specified.

The suite supports tests from the smallest subset of the instructions that could be implemented, enough for an image of the IBM logo to be displayed:

Opcode	**Description**
00E0	Clear the screen.
6XNN	Load normal register with immediate value.
ANNN	Load index register with immediate value.
7XNN	Add immediate value to normal register.
DXYN	Draw sprite to screen (unaligned, easier to implement).
1NNN	Jump (optional, only at the end).

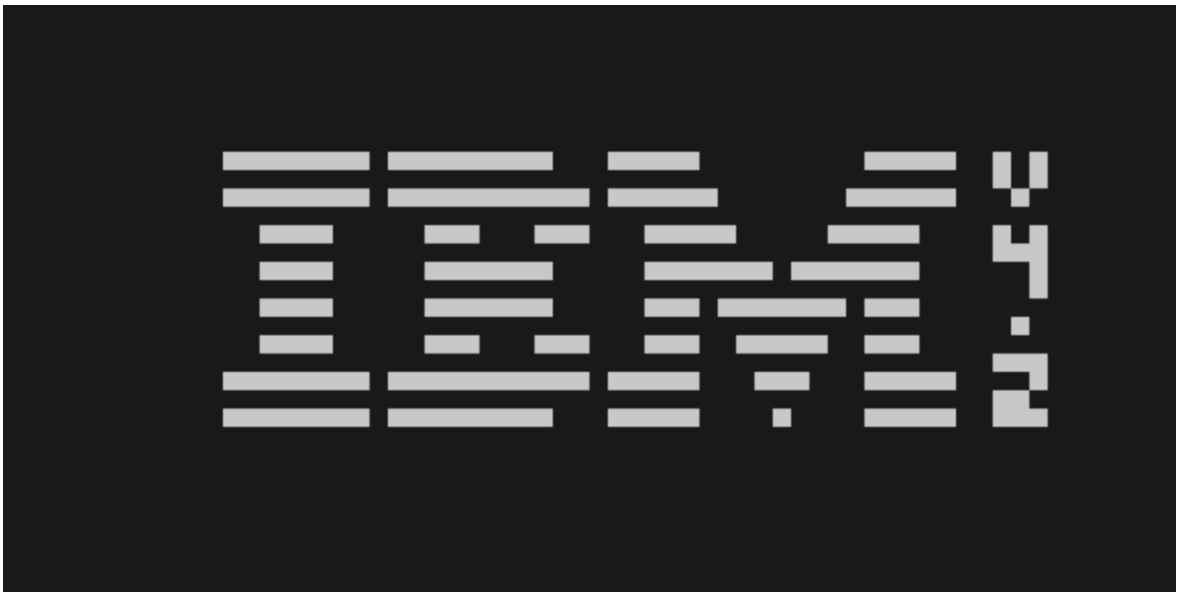

Figure 3.3: The IBM logo display running in `Edra` showing the correct result.

To the entire instruction set and the correct hardware timings and quirks, including the flags set by the instructions:

Figure 3.4: The Corax+ opcode and the flags tests running in `Edra` showing the correct result.

These tests have been ran manually in all possible configurations to make sure the emulator is behaving as expected, even in modern SCHIP mode.

Figure 3.5: The quirks test, normal CHIP8 and modern SCHIP variants running in `Edra` showing the correct result for each of the respective platforms.

4 The final application

The application, called `Edra`, has been designed from the outset with a primary focus on performance, emphasizing both speed and low latency. To meet these requirements, the C programming language was selected for the development of the emulator, due to its low-level access to system resources and minimal runtime overhead.

For graphical output and user interaction, the `SDL3` library was chosen for its cross-platform compatibility, lightweight nature, and ease of integration into C-based projects.

When evaluating options for graphical user interface (GUI) development in pure C, the available choices were rather limited. The GUI development space is predominantly occupied by C++ libraries such as *ImGui*, which are not directly compatible with C without the use of external bindings which would make the development process rather complicated.

Ultimately, the project adopted the `clay` library, a platform-agnostic C layout engine. This library facilitates GUI construction by generating a series of rendering operations that can be implemented using a backend such as `SDL3`, thereby enabling flexible and portable UI rendering within the application.

4.1 Design and implementation

The emulator's core functionality was subsequently modularized into a dedicated git submodule, referred to as `cchip8`, located in the *lib/cchip8* directory. This module encapsulates all fundamental components required for CHIP8 emulation.

Building upon this foundation, the CHIP8 driver, located in *drivers/chip8*, instantiates a machine capable of loading and executing CHIP8 programs.

This virtual machine is then integrated into an application state, which is responsible for managing its output within the broader context of `Edra`'s state management system.

Similarly, the Brainfuck emulator has its own module called `cbf` available at *lib/cbf* which handles all of the components necessary for the machine at *drivers/cbf* which is then used by the state in the app.

For cross-platform non-standard features, such as dynamic arrays, .ini file parsing and so on, custom implementations have been made and separated into their own separate module called `auxum` available in the *lib/auxum* folder.

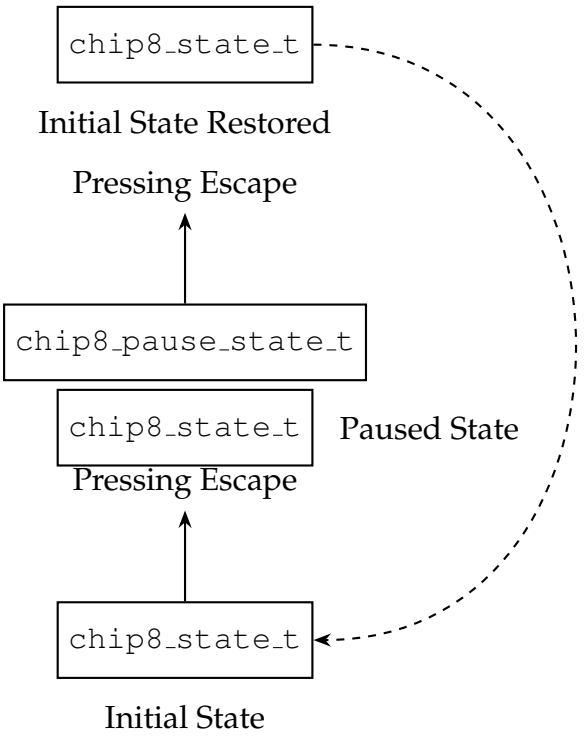

Figure 4.1: Stack-based state transitions, as used in the app.

4.2 Porting and compiling

During cross-platform porting efforts, several implementation constraints necessitated platform-specific adaptations:

- *PlayStation Vita*: JIT emulation was disabled due to the absence of POSIX-compliant threading support, which prevented successful compilation of the GNU Lightning library. Additionally, memory allocation thresholds required modification within the VitaSDK framework, as implemented in `auxum`.
- *Microsoft Windows*: JIT functionality was similarly disabled following segmentation faults during compilation attributed to GNU Lightning incompatibilities, potentially stemming from toolchain limitations within the GCC Windows environment.
- *Linux*: Full functionality was maintained without modification, demonstrating native compatibility with the implementation's architectural dependencies.

To facilitate compilation, a standardized build system was implemented using `Make`. This system provides template commands for all supported platforms, as documented in the project's `GitHub repository`:

1. *Environment Configuration*: Specify `SDL3_DIR` and `SDL3_TTF_DIR` variables in CMake's build cache when required for dependency resolution.
2. *Platform-Specific Compilation*: Execute the appropriate build command: *make b{platform}{configuration}*
   - `platform` denotes the target system:
     - `w`: Microsoft Windows
     - `u`: Unix-like systems (Linux, BSD)
     - `v`: PlayStation Vita
   - `configuration` specifies the build type:
     - `d`: Debug build
     - `r`: Release build

4.3 Usage

GUI

Figure 4.2: The app running on a PlayStation Vita system.

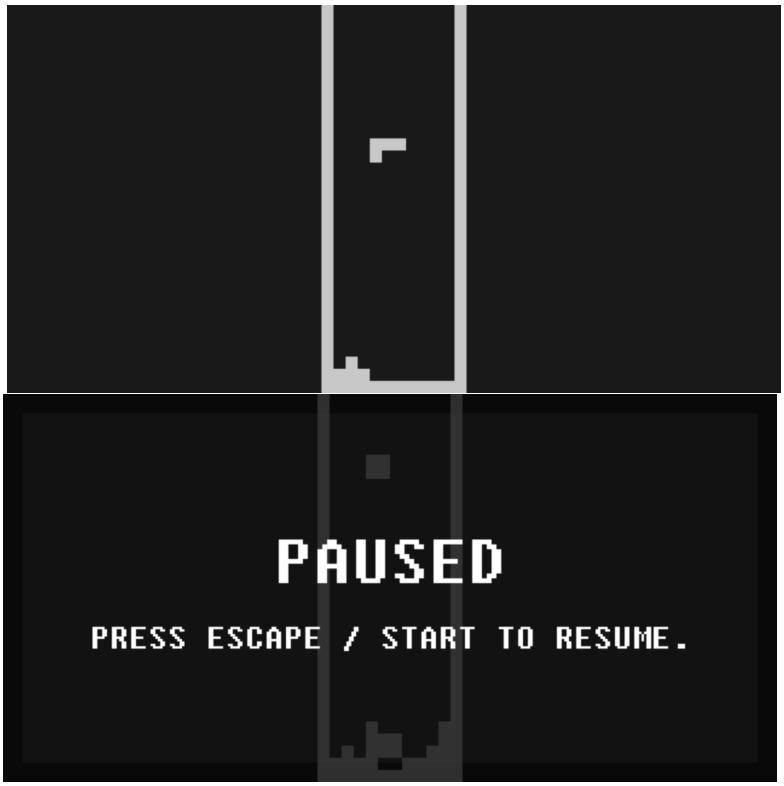

Figure 4.3: The app being paused by pressing Escape or Start on a gamepad.

5 Conclusions and Future Work

This thesis has systematically explored the design, implementation, and performance trade-offs between interpretation and just-in-time (JIT) compilation for software emulation, using the minimalist languages *Brainfuck* and *CHIP-8* as pedagogical case studies.

The key findings demonstrate that *JIT compilation consistently outperforms interpretation*, particularly for computationally intensive or long-running programs, due to the elimination of repetitive instruction fetching and decoding overhead. Empirical benchmarks revealed execution times as low as *3–9% of interpreter runtimes* for Brainfuck programs.

Optimizations such as *instruction folding*, *precalculated jumps*, and *hot loop replacement* proved highly effective, though diminishing returns were observed with excessive specialization. For instance, overly aggressive loop collapsing occasionally generated machine code identical to unoptimized sequences, increasing complexity without performance gains. This underscores the importance of *balancing optimization effort with practical returns*, especially in constrained environments.

The modular architecture of the final application, *Edra*, successfully decoupled emulation logic (via `cbf` and `cchip8` libraries) from platform-specific I/O (using SDL3 and Clay), enabling portable CHIP-8 and Brainfuck emulation. Test validation with established suites (e.g., Timendus for CHIP-8) confirmed correctness across standard and extended specifications like SCHIP.

Future Work

Several avenues merit further exploration:

1. *Better User Experience*: Improved GUI with program selection and configurable in-app settings.

2. *Broader Language Support*: Extending the framework to emulate more complex architectures (e.g. 6502, Z80, etc).

3. *More platforms supported*: Emulating game consoles / platforms that have already had their architecture reverse engineered and are openly available (e.g. Nintendo GameBoy, etc)

Bibliography

[1] N. Cruz, M. R. Ferrández, J. Redondo, and J. Alvarez. Applications of chip-8, a virtual machine from the late seventies, in current degrees in computer engineering. *11th International Conference on Education and New Learning Technologies, Palma de Mallorca, Spain*, 2019. doi:10.21125/edulearn.2019.0501.

[2] O. Mazonka and D. Cristofani. A very short self-interpreter. https://www.researchgate.net/publication/1881452_A_Very_Short_Self-Interpreter, 12 2003. Online; accessed 04 April 2025.

[3] S. Morr. Esoteric programming languages: An introduction to brainfuck, intercal, befunge, malbolge, and shakespeare. https://blinry.org/esolangs/esolangs.pdf. Online; accessed 04 April 2025.

[4] U. Müller. Brainfuck on esolang wiki. Online; accessed 04 April 2025. URL: https://esolangs.org/wiki/Brainfuck.

[5] J. Sang-Woo. 50,000,000,000 instructions per second: Design and implementation of a 256-core brainfuck computer. https://people.csail.mit.edu/wjun/papers/sigtbd16.pdf. Online; accessed 04 April 2025.

[6] S. Schümann. Chip-8 variant opcode table used in the cadmium emulator, 1993. Online; accessed 04 April 2025. URL: https://chip8.gulrak.net/.

[7] D. Temkin. Language without code: Intentionally unusable, uncomputable, or conceptual programming languages. *Journal of Science and Technology of the Arts*, 9:83, 12 2017. doi:10.7559/citarj.v9i3.432.

[8] L. Xiaoting, L. Xiao, C. Lingwei, P. Rupesh, and W. Dinghao. Alphaprog: Reinforcement generation of valid programs for compiler fuzzing. *Proceedings of the AAAI Conference on Artificial Intelligence*, 2022. doi:10.1609/aaai.v36i11.21527.