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, thereby offering a comprehensive introduction to emulator design.

To evaluate these techniques in a controlled and educational context, interpreters and JIT compilers were implemented for two minimalistic and well-documented languages: Brainfuck and CHIP-8. These languages were selected due to their simplicity, which enables the study to center on emulator behavior rather than language-specific intricacies.

The analysis includes performance benchmarking, implementation complexity, and the impact of selected optimizations such as instruction folding, jump precalculation, hot loop replacement, and efficient register allocation. Empirical results 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. For instance, excessive hot loop replacement 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 in emulator design and aim to offer accessible insights for readers new to emulation concepts, particularly those engaged in experimental or hobbyist development with languages like Brainfuck and CHIP-8.

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: software emulation, interpretation, JIT compilation, Brainfuck, CHIP-8, 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, and the development of sandboxed runtime systems.

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 which is unit and integration tested.

This thesis aims to provide readers with both a conceptual and practical understanding of how various emulation techniques perform and operate under comparable conditions.

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 characteristics.
- **Chapter 3** explores the historical evolution and variations of the CHIP-8 virtual machine, detailing the implementation strategies adopted in this thesis and their implications for emulator design.
- **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

CHIP-8 Applications in engineering [1]. Brainfuck in reinforcement learning [6]. Esoteric languages list with BF in it [3]. Brainfuck conceptual [5]. Brainfuck hardware [4]. Brainfuck self interpreter [2].

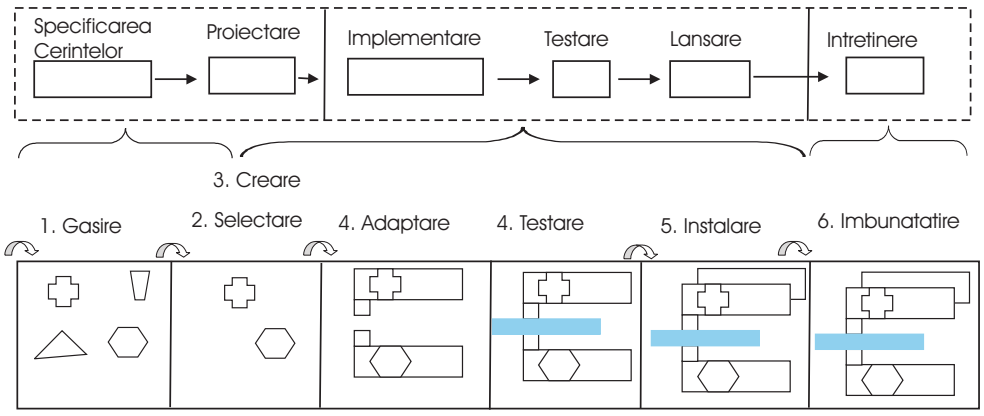

Figure 1.1: Ciclul de dezvoltare al sistemelor bazate pe componente adaptat modelului cascadã

Inserarea și Referirea la Tabelul 1.1.

Nume algoritm	Toate soluțiile	Soluția optimã
Nume 1	20	5
Nume 2	20	2

Table 1.1: Soluții obținute

Adaugarea și Referirea la o Ecuație 1.1.

$$ws_N4 = w_{14} * N1 + W_{24} + N2 + w_{34} * N3 \tag{1.1}$$

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. 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 (commonly 30,000 cells in most implementations) of bytes 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 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 community (esolangs). 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

2.1.1 Programming in the language

To facilitate a practical understanding of the Brainfuck language and its idiomatic constructs, we examine a canonical program 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:

```
1  >++++++++[<+++++++++>-]<.
```

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:

```
[-]
```

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 interpreters or optimizing compilers targeting esoteric or constrained instruction sets.

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 Emulating yourself?

2.2.2 Simple interpreter

A basic Brainfuck interpreter operates by sequentially reading each character of the source code and executing the corresponding operation against a simulated memory model. The interpreter typically maintains a data array—commonly 30,000 bytes in size—initialized to zero, along with a data pointer that tracks the current cell. As the interpreter parses the code, it translates each command into its respective behavior: incrementing or decrementing the cell's value (+ or -), moving the data pointer left or right (¡ or ¿), reading input (,), or writing output (.). Control flow constructs ([and]) are implemented by jumping to the corresponding matching bracket based on the current cell's value. If the cell is zero at [, execution jumps forward to the instruction following the matching]; otherwise, it continues normally. Similarly,] causes a backward jump to the matching [if the current cell is non-zero. To efficiently support these jumps, the interpreter often preprocesses the source code to map matching bracket pairs. The simplicity of this execution model makes Brainfuck interpreters ideal for illustrating core principles of language interpretation, including parsing, memory management, and flow control.

2.2.3 Static compilation

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

2.3 Applying optimizations

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nos-

trud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

2.3.1 Precalculating jumps

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

2.3.2 Instruction folding

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

2.3.3 Sequence matching for common patterns

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

2.4 Testing

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.

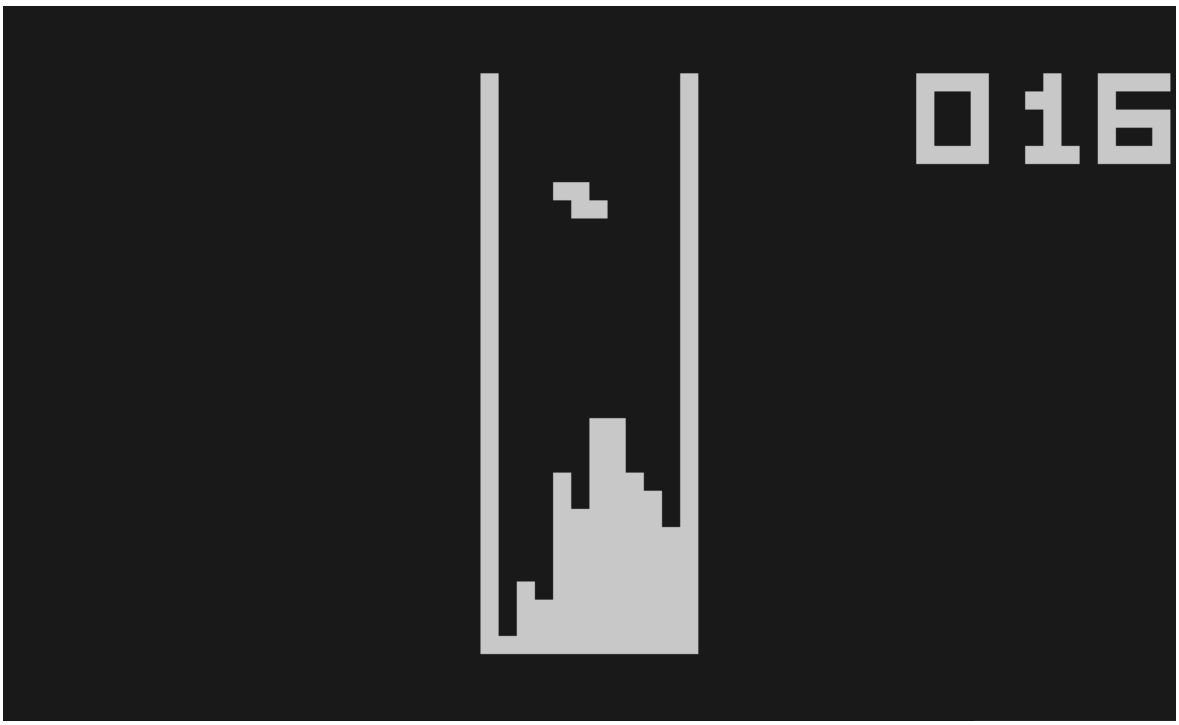

Figure 3.1: Tetris clone written in CHIP-8 by Fran Dachille in 1991 running in `Edra`.

3.2 Virtual machine description

Memory

The CHIP-8 virtual machine provides 4 KB 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.

Registers

The CHIP-8 virtual machine contains sixteen 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 64 by 32 pixels, rendering graphics in monochrome. Sprites are rendered onto the screen using an XOR 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.

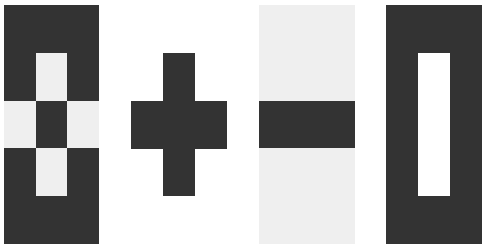

Table 3.1: Illustration of sprite XOR-ing, transforming an 8 (left) into a 0 (right), VF is set in this case to 1. The + sign is not part of the sprite.

The CHIP-8 virtual machine includes two timers: the delay timer and the sound timer. Both decrement at a fixed rate of 60 Hz. 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, like so:

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

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.

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 when needed.

```
struct chip8_state
{
    bool draw_flag;
    ... 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.

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.

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

The 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. 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.

3.3.2 Just-in-time compiler

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

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.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

3.5 Testing

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

4 The final application

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nos-

trud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

4.1 Design

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

4.2 Implementation

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

4.3 Usage

4.3.1 Command line

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

4.3.2 GUI

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

5 Conclusions

Concluzii ...

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] 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.

[5] 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.

[6] 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.