Design, Implementation and Comparison of Software Emulation Techniques

by Oniga Andrei-Mihai
under the supervision of
Lect. Dr. Mihai Andrei

Motivation & Scope

Why was this done? Who for?
Why Brainfuck and CHIP-8?

Project Goals

Implementation
(interpreter / JIT compiler),

Program Analysis,

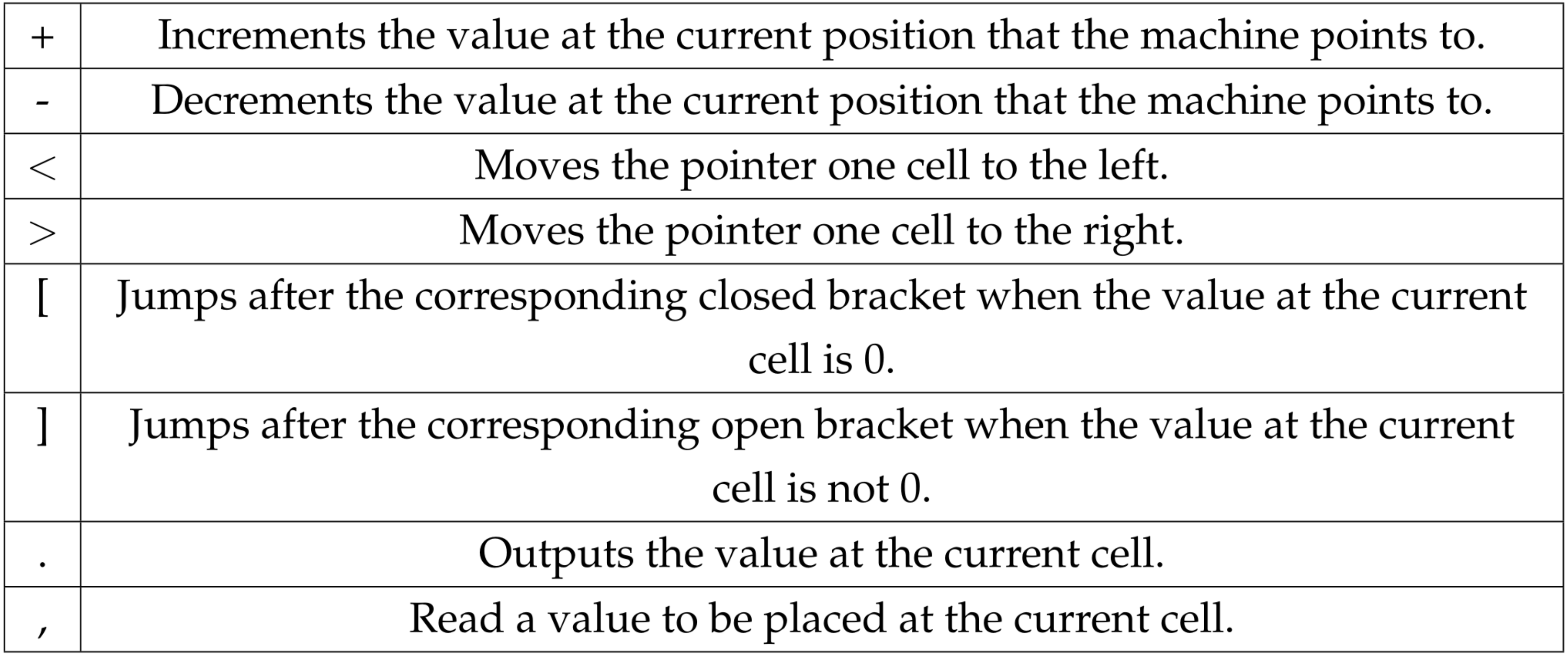
and Optimizations

Brainfuck Machine Model

Relatively infinite memory tape

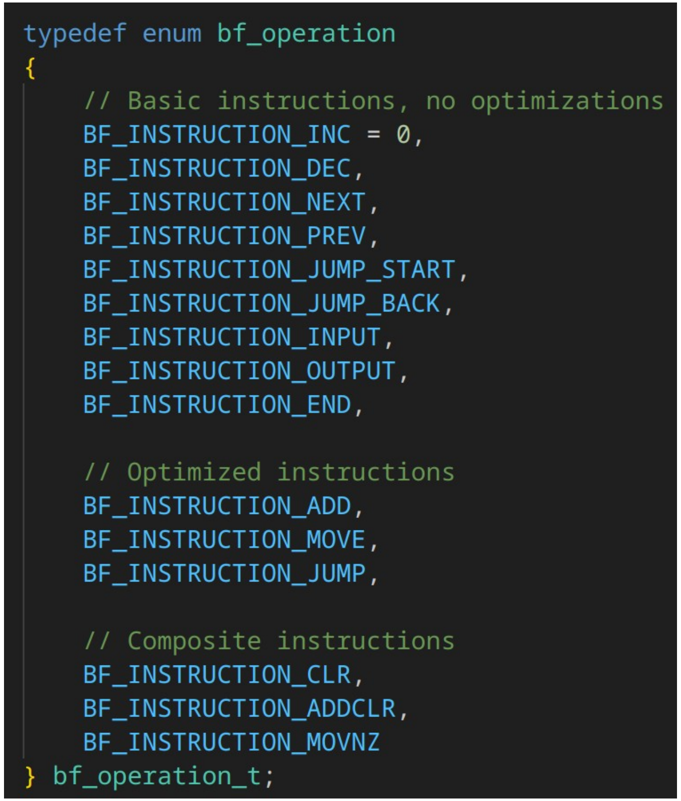
with a pointer

Brainfuck Language Overview



+	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.

Intermediate Representation (IR)



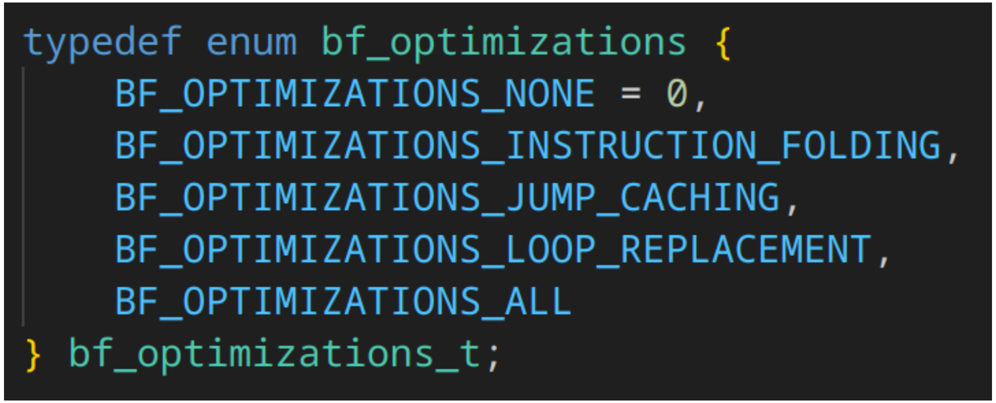
```c
typedef enum bf_operation
{
    // Basic instructions, no optimizations
    BF_INSTRUCTION_INC = 0,
    BF_INSTRUCTION_DEC,
    BF_INSTRUCTION_NEXT,
    BF_INSTRUCTION_PREV,
    BF_INSTRUCTION_JUMP_START,
    BF_INSTRUCTION_JUMP_BACK,
    BF_INSTRUCTION_INPUT,
    BF_INSTRUCTION_OUTPUT,
    BF_INSTRUCTION_END,

    // Optimized instructions
    BF_INSTRUCTION_ADD,
    BF_INSTRUCTION_MOVE,
    BF_INSTRUCTION_JUMP,

    // Composite instructions
    BF_INSTRUCTION_CLR,
    BF_INSTRUCTION_ADDCLR,
    BF_INSTRUCTION_MOVNZ
} bf_operation_t;
```

Translating Brainfuck into Efficient Structures

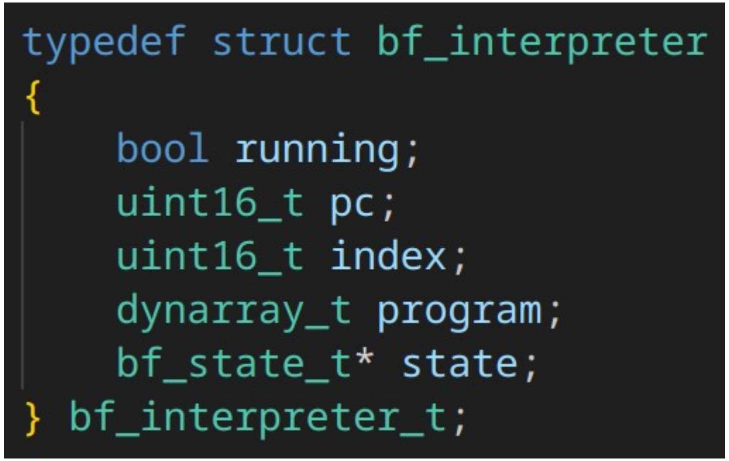
Optimizations Applied



```
typedef enum bf_optimizations {
    BF_OPTIMIZATIONS_NONE = 0,
    BF_OPTIMIZATIONS_INSTRUCTION_FOLDING,
    BF_OPTIMIZATIONS_JUMP_CACHING,
    BF_OPTIMIZATIONS_LOOP_REPLACEMENT,
    BF_OPTIMIZATIONS_ALL
} bf_optimizations_t;
```

Jump caching,
Instruction Folding
and Pattern Matching

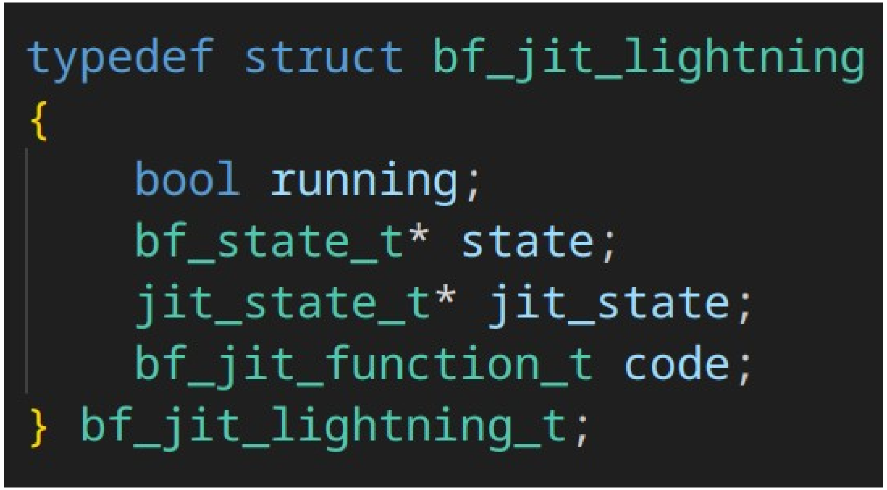
Interpreter Implementation



```c
typedef struct bf_interpreter
{
    bool running;
    uint16_t pc;
    uint16_t index;
    dynarray_t program;
    bf_state_t* state;
} bf_interpreter_t;
```

Basic Dispatch Loop and The Fetch Decode Execute cycle

Static JIT Compilation



```
typedef struct bf_jit_lightning
{
    bool running;
    bf_state_t* state;
    jit_state_t* jit_state;
    bf_jit_function_t code;
} bf_jit_lightning_t;
```

Choosing a library,

why GNU Lightning,

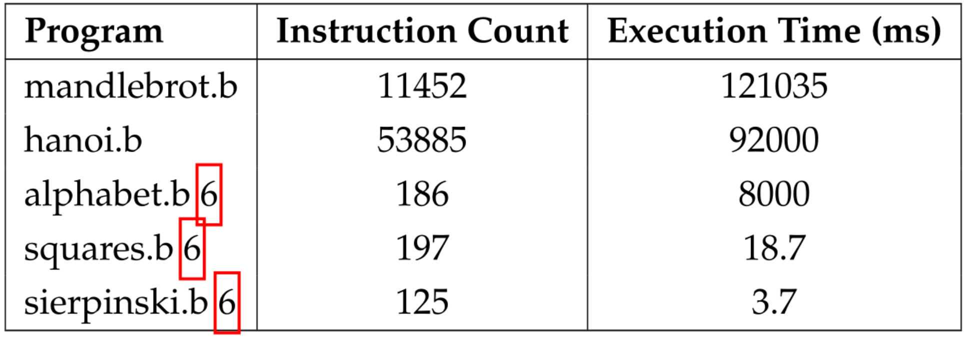
implementation Details

Optimization Insights

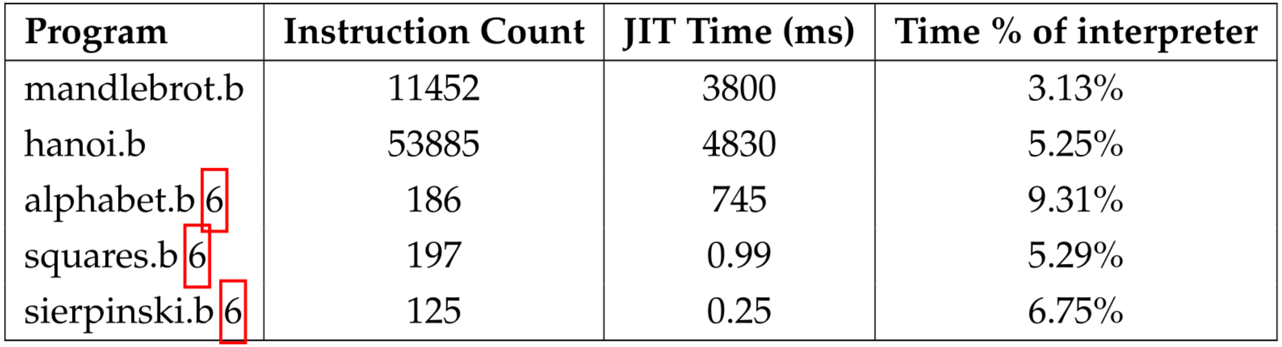
When Simplicity Beats Aggressiveness: replacing too many loops becomes redundant as the generated JIT code is identical

Performance Results (Brainfuck)

Results before all optimizations:



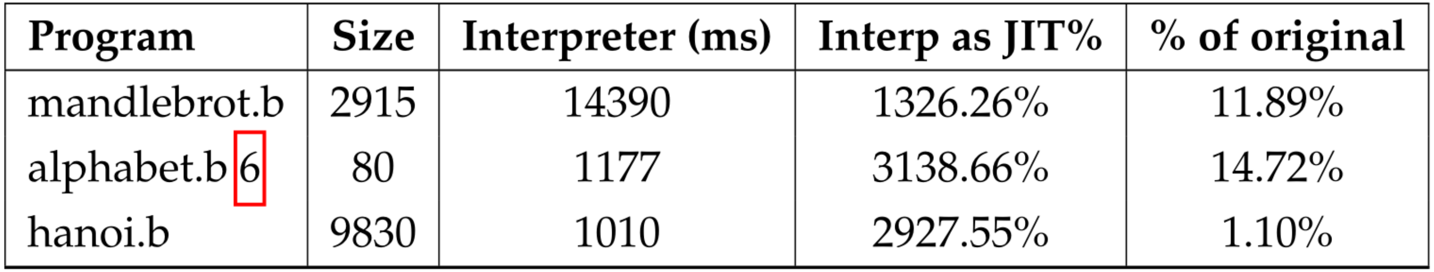
Program	Instruction Count	Execution Time (ms)
mandlebrot.b	11452	121035
hanoi.b	53885	92000
alphabet.b[6]	186	8000
squares.b[6]	197	18.7
sierpinski.b[6]	125	3.7



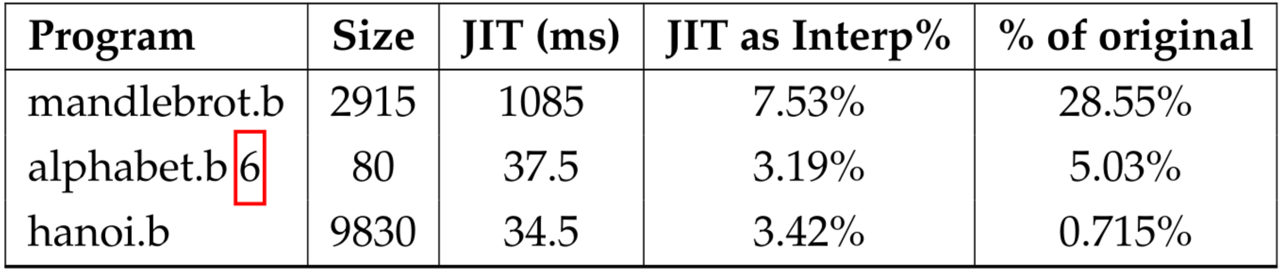
Program	Instruction Count	JIT Time (ms)	Time % of interpreter
mandlebrot.b	11452	3800	3.13%
hanoi.b	53885	4830	5.25%
alphabet.b[6]	186	745	9.31%
squares.b[6]	197	0.99	5.29%
sierpinski.b[6]	125	0.25	6.75%

Performance Results (Brainfuck)

Results after all optimizations:



Program	Size	Interpreter (ms)	Interp as JIT%	% of original
mandlebrot.b	2915	14390	1326.26%	11.89%
alphabet.b 6	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 6	80	37.5	3.19%	5.03%
hanoi.b	9830	34.5	3.42%	0.715%

Brainfuck Testing Strategy

Ensuring Correctness with Unit Tests

CHIP-8 Architecture

64 KB Memory,

16 Registers,

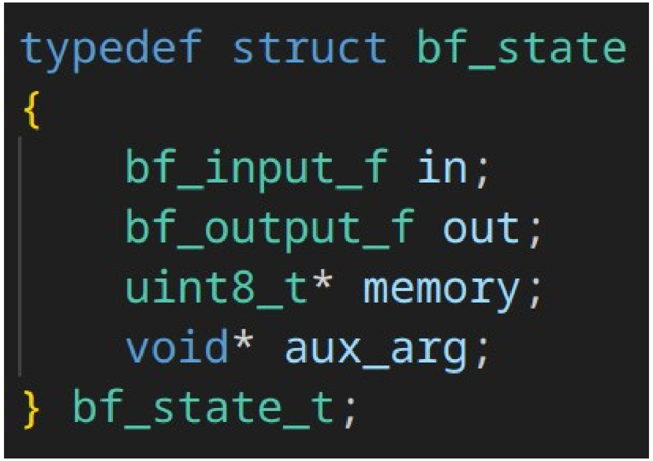
35 Instructions, 2 Timers,

64 x 32 XOR Display

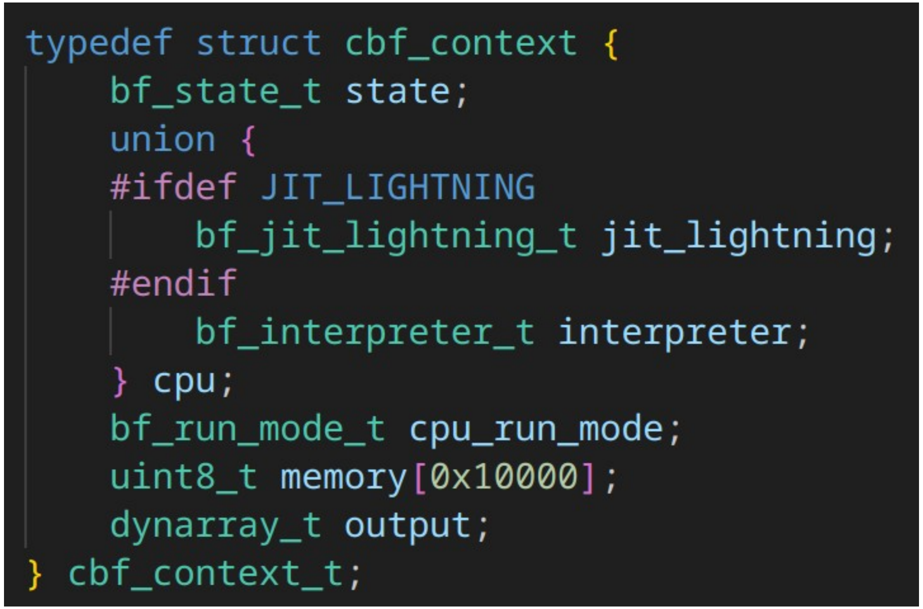
and 16-key Input

Interpreter Implementation

Modular Design and Execution Strategy



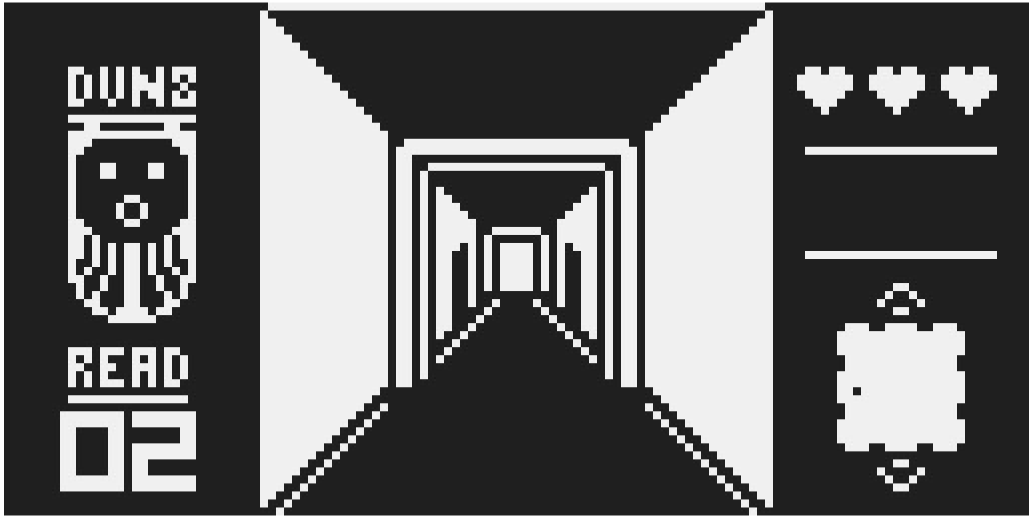
```c
typedef struct bf_state
{
    bf_input_f in;
    bf_output_f out;
    uint8_t* memory;
    void* aux_arg;
} bf_state_t;
```



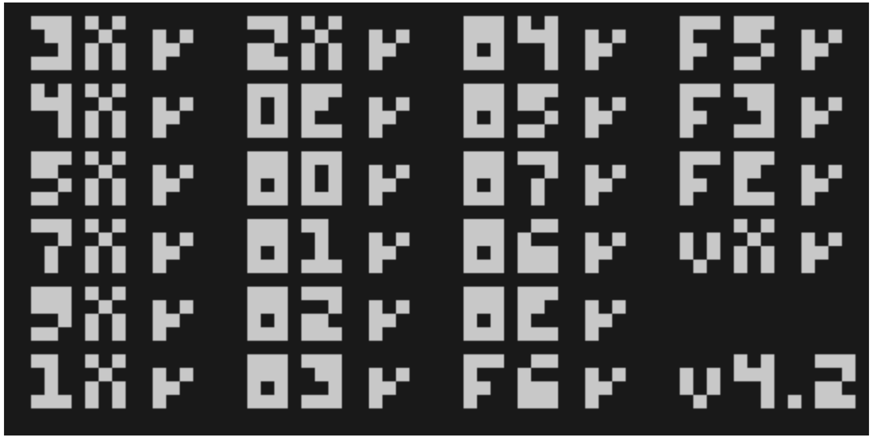
```c
typedef struct cbf_context {
    bf_state_t state;
    union {
    #ifdef JIT_LIGHTNING
        bf_jit_lightning_t jit_lightning;
    #endif
        bf_interpreter_t interpreter;
    } cpu;
    bf_run_mode_t cpu_run_mode;
    uint8_t memory[0x10000];
    dynarray_t output;
} cbf_context_t;
```

Quirks & Extensions

Modern SCHIP-8 Support and Differences

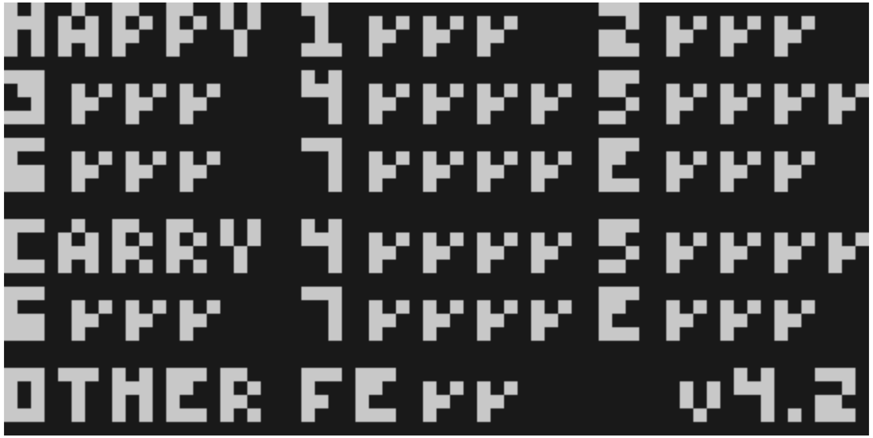


CHIP-8 Testing – Timendus’ Test Suite



```
3XNN ✓  2NNN ✓  8XY4 ✓  FX55 ✓
4XNN ✓  00EE ✓  8XY5 ✓  FX33 ✓
5XY0 ✓  00E0 ✓  8XY7 ✓  FX1E ✓
7XNN ✓  8XY1 ✓  8XY6 ✓  VXNN ✓
9XY0 ✓  8XY2 ✓  8XYE ✓
1NNN ✓  8XY3 ✓  FX65 ✓  v4.2
```

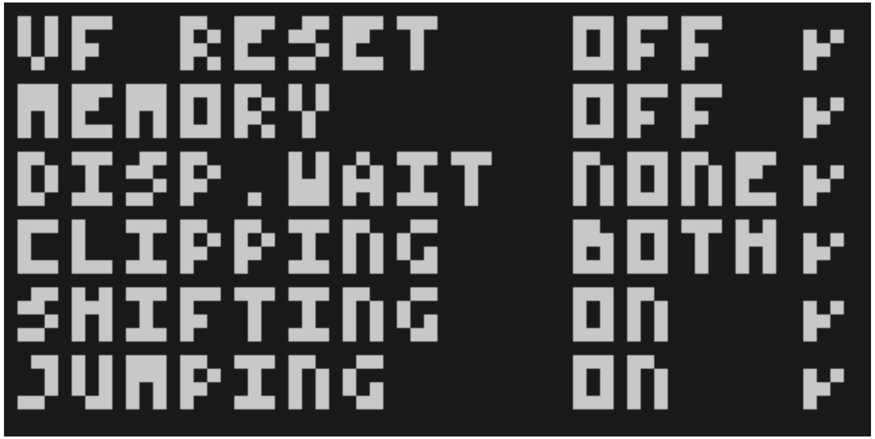
Instruction corectness check



```
HAPPY 1 ✓✓✓     2 ✓✓✓
3 ✓✓✓     4 ✓✓✓✓  5 ✓✓✓✓
6 ✓✓✓     7 ✓✓✓✓  E ✓✓✓
CARRY 4 ✓✓✓✓  5 ✓✓✓✓
6 ✓✓✓     7 ✓✓✓✓  E ✓✓✓
OTHER FE ✓✓          v4.2
```

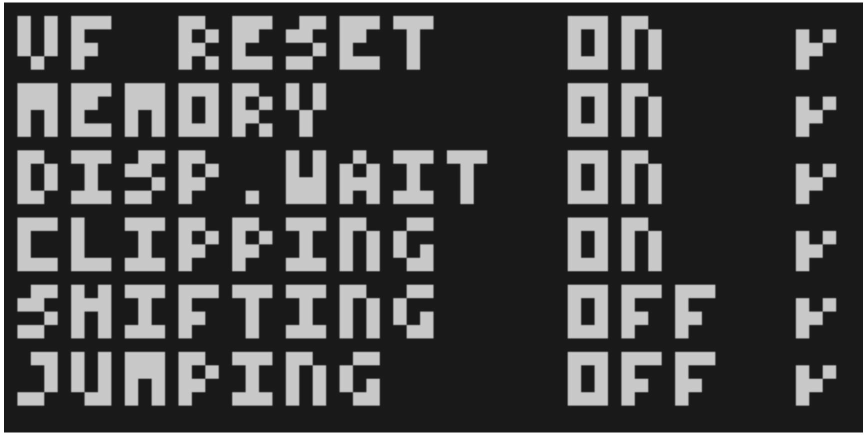
Flag corectness check

CHIP-8 Testing – Timendus’ Test Suite



```
VF RESET   OFF  ✓
MEMORY     OFF  ✓
DISP.WAIT  NONE ✓
CLIPPING   BOTH ✓
SHIFTING   ON   ✓
JUMPING    ON   ✓
```

Quirk Test in SCHIP Mode



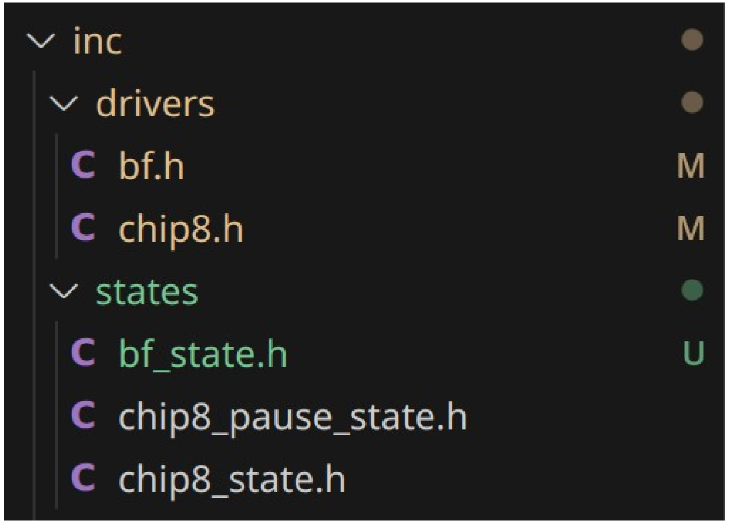
```
VF RESET   ON   ✓
MEMORY     ON   ✓
DISP.WAIT  ON   ✓
CLIPPING   ON   ✓
SHIFTING   OFF  ✓
JUMPING    OFF  ✓
```

Quirk Test in Normal Mode

The Final Application – Edra

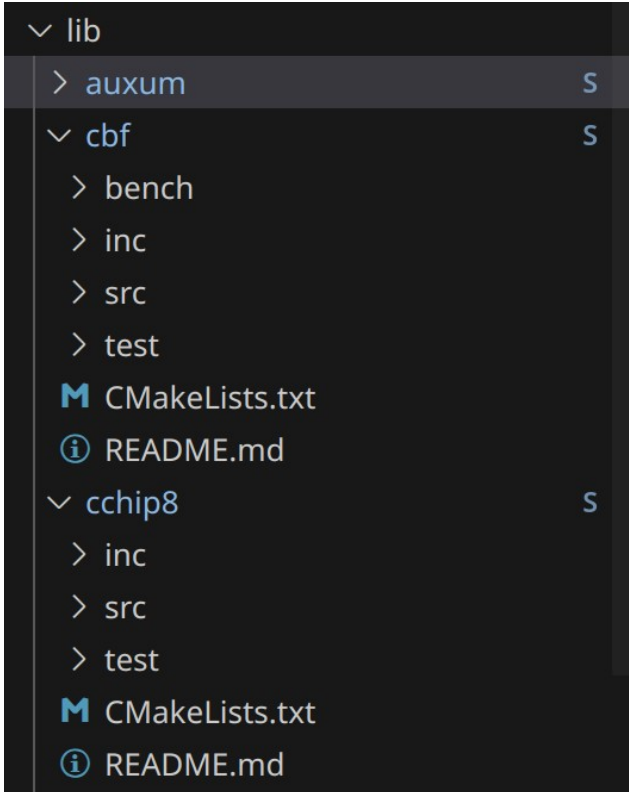
Glueing everything toghether in C17

Design and Architecture



```
∨ inc                          ●
  ∨ drivers                    ●
    C bf.h                     M
    C chip8.h                  M
  ∨ states                     ●
    C bf_state.h               U
    C chip8_pause_state.h
    C chip8_state.h
```

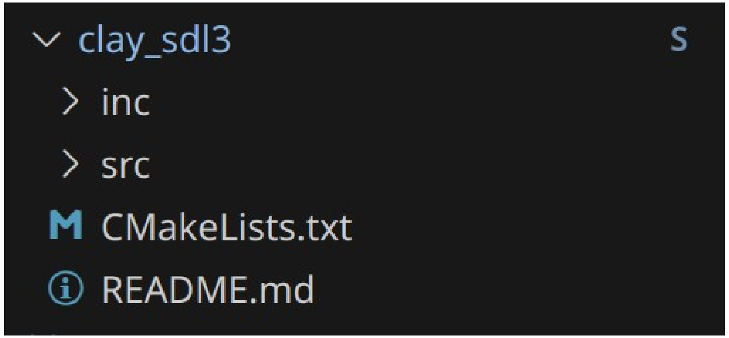
Modularity and separation of concerns.



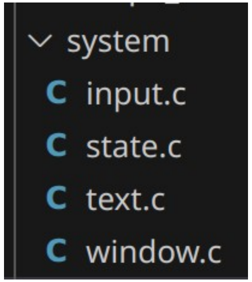
```
∨ lib
  > auxum                      S
  ∨ cbf                        S
    > bench
    > inc
    > src
    > test
    M CMakeLists.txt
    ⓘ README.md
  ∨ cchip8                     S
    > inc
    > src
    > test
    M CMakeLists.txt
    ⓘ README.md
```

User Interface in SDL3 using CLay

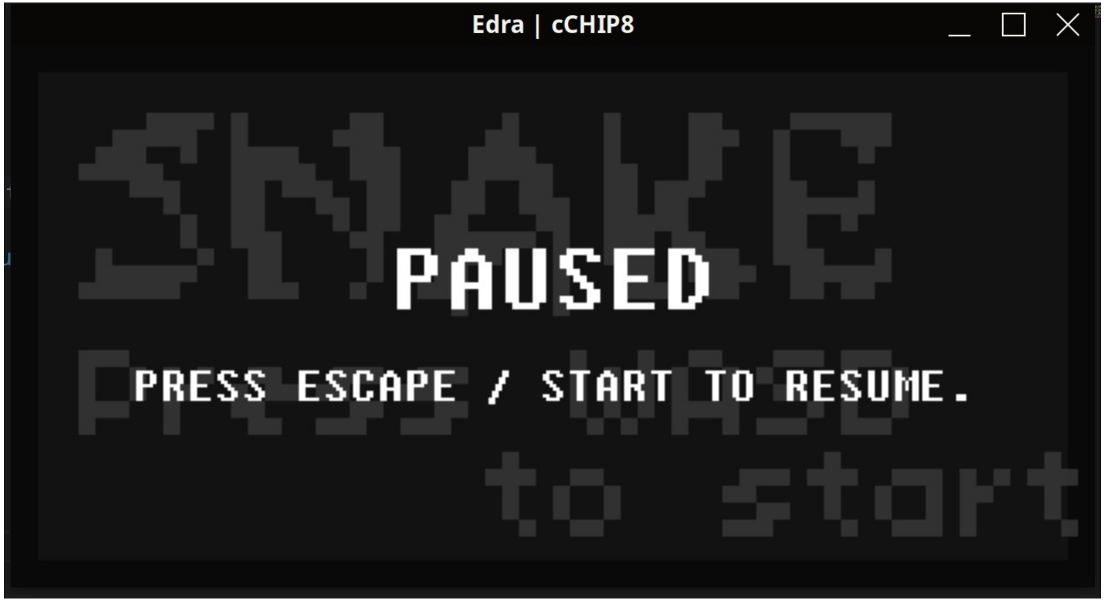
Handling Graphics, Input, and Display



```
∨ clay_sdl3                S
  > inc
  > src
  M CMakeLists.txt
  ⓘ README.md
```



```
∨ system
  C input.c
  C state.c
  C text.c
  C window.c
```



PAUSED

PRESS ESCAPE / START TO RESUME.

Cross-Platform Porting

Windows:

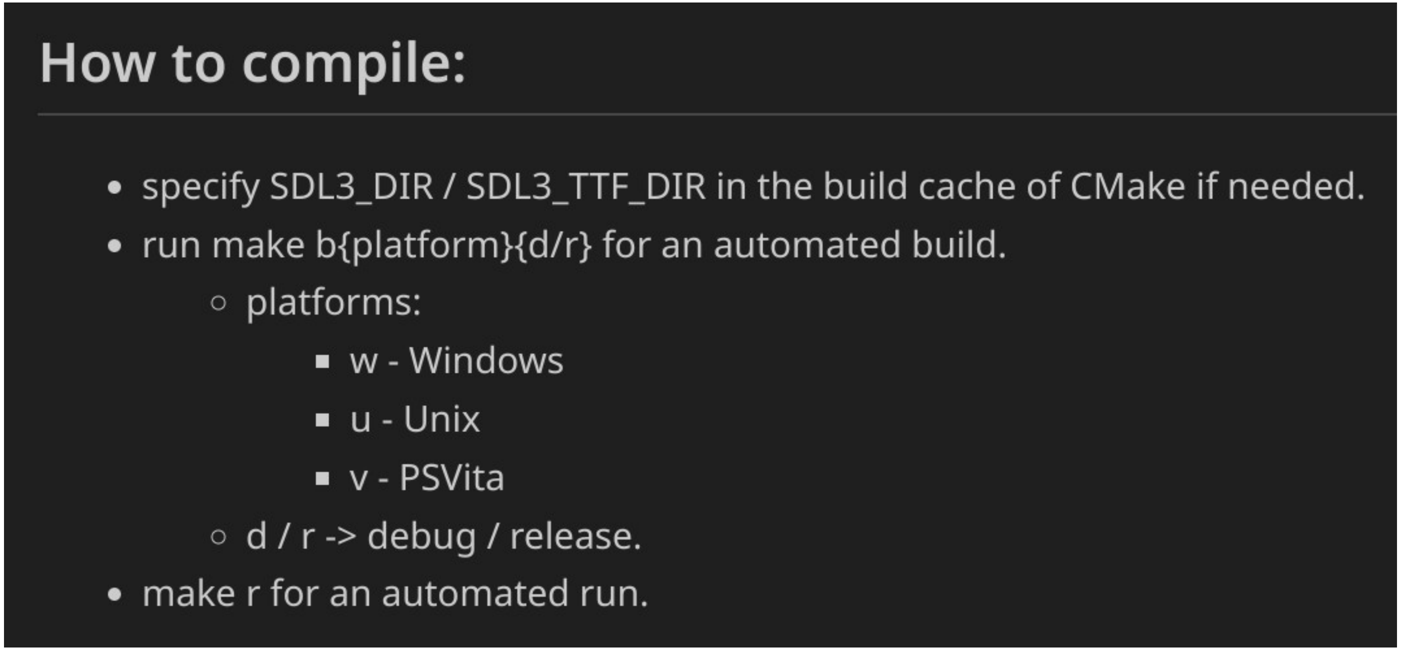
- No JIT as GNU Lightning works only on POSIX compliant systems.

Linux (+Android on Termux + Termux X11 display server):
- Works out of the box with all modules properly implemented.

PSVita:
- Gamepad input only and also no JIT as system is not POSIX compliant.

Build System & Usage

Makefile Structure and Platform Targets



How to compile:

- specify SDL3_DIR / SDL3_TTF_DIR in the build cache of CMake if needed.
- run make b{platform}{d/r} for an automated build.
  - platforms:
    - w - Windows
    - u - Unix
    - v - PSVita
  - d / r -> debug / release.
- make r for an automated run.

Future Work

UX Improvements, More Platforms,
More Architectures

Final Thoughts

Contributions to entry level

Emulation and to Education

Showcase and Demonstration

Running on Desktop and PS Vita
Live!

The End

Thank You