Rustboy

A Rust journey into Game Boy dev

ffex @ fosdem2026

01. Introduction

My story

My gameboys

My story

Me and the gameboy

The not-so-exciting life of a programmer

In the daily routine

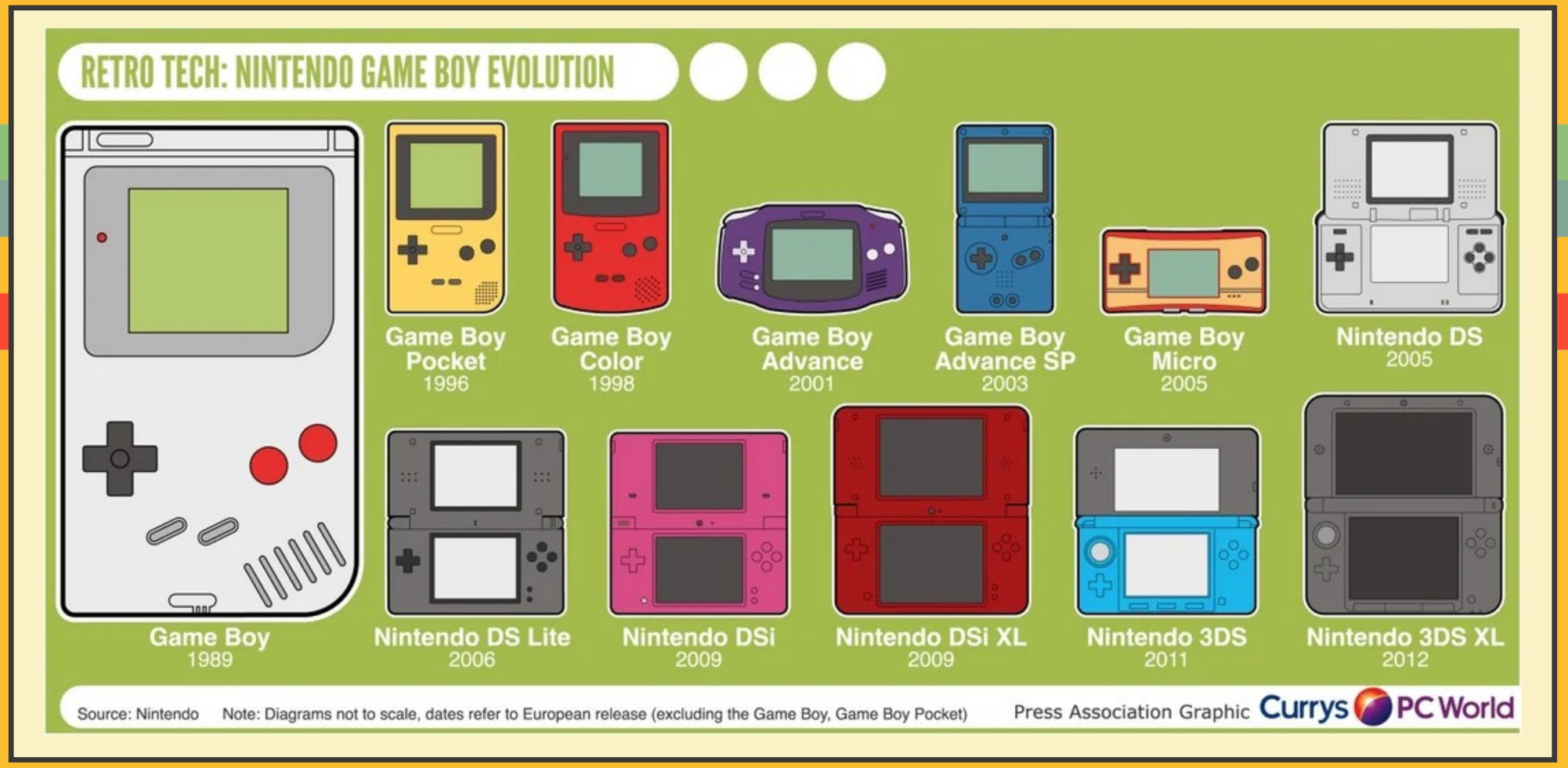
- We have a **problem**
- Find and replicate the **problem**
- Search for the **problem** online
- Ask AI about the **problem**
- Copy a *solution* of the **problem**
- Test the *solution*
- Is it the best *solution*?
- Search for the best *solution* online
- .....

Fosdem 2025

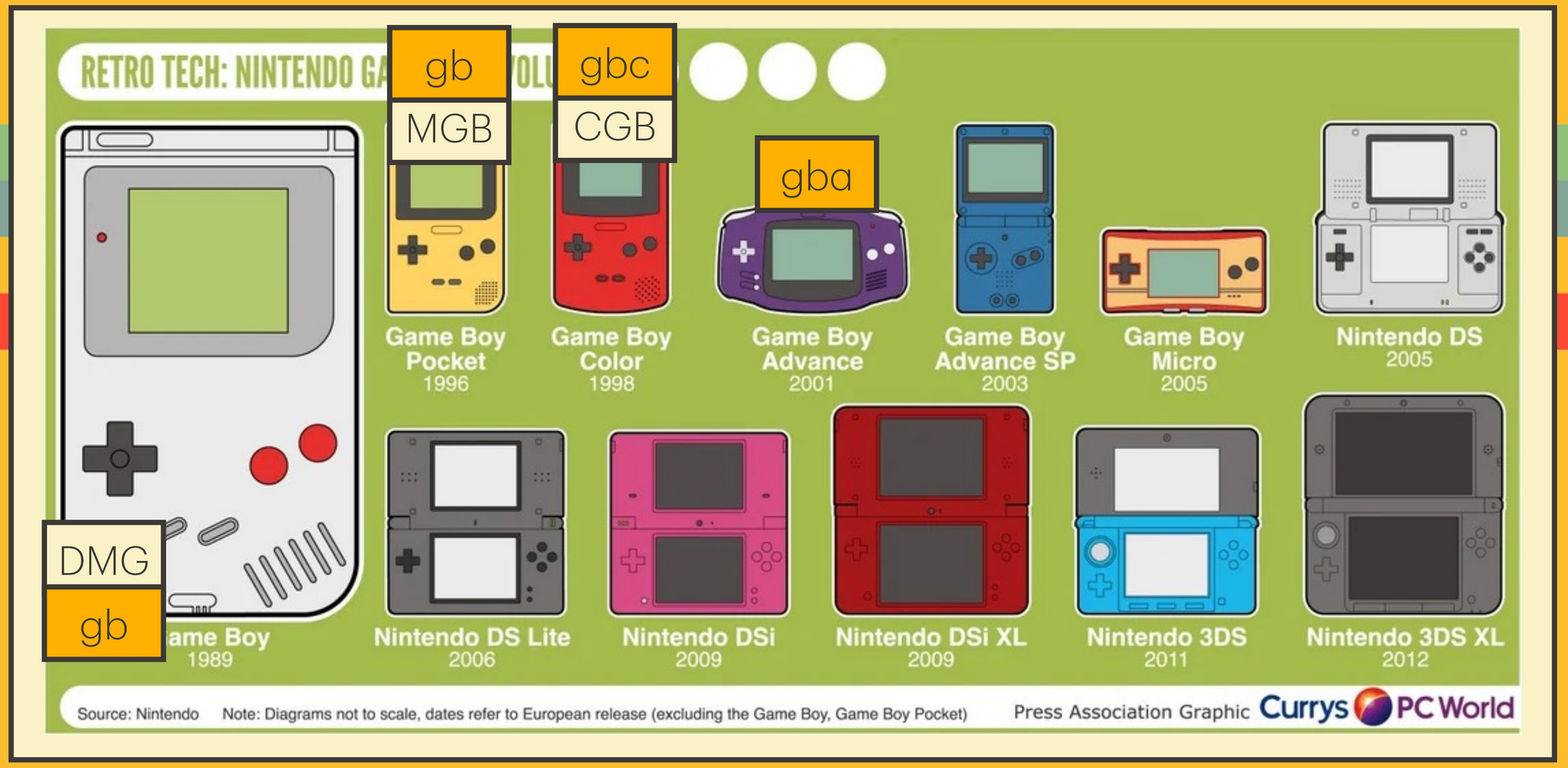
Boom!

02. Hardware

All the game boys



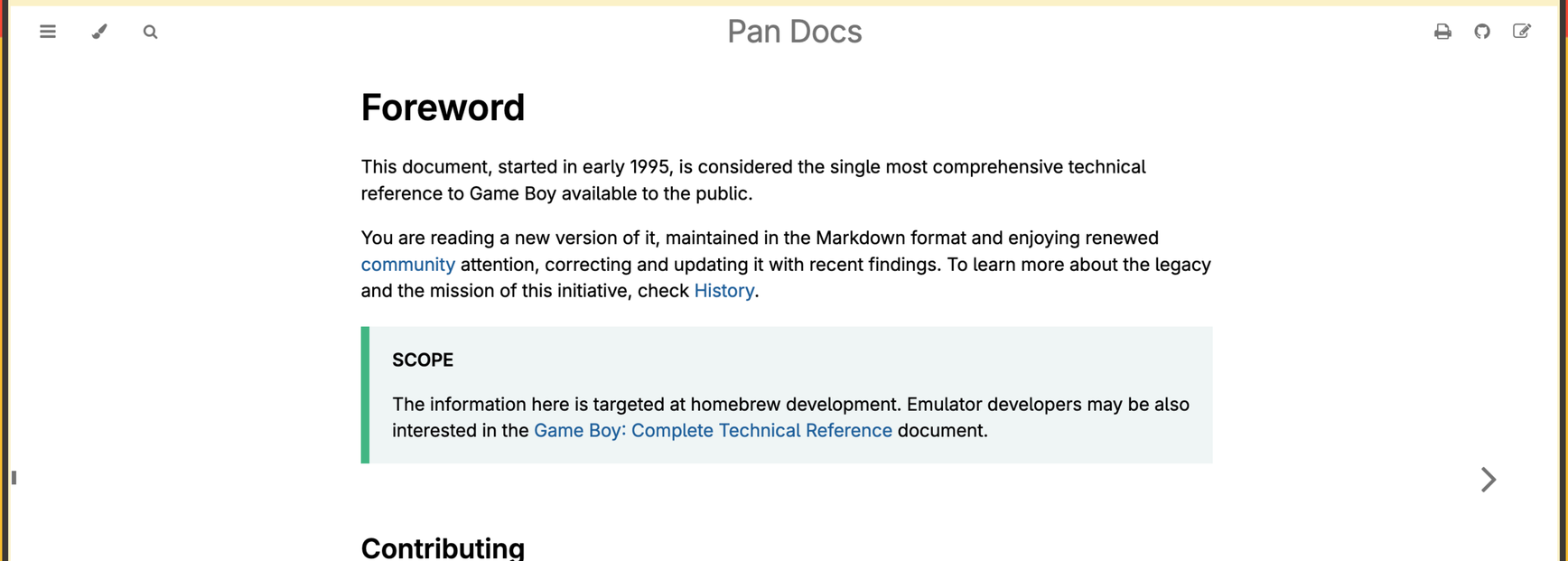
All the game boys



Pandocs

This document, started in early 1995, is considered the single most comprehensive technical reference to Game Boy available to the public.

Link: **https://gbdev.io/pandocs/**



Memory Map

Start	End	Description	Notes
0000	3FFF	16 KiB ROM bank 00	From cartridge, usually a fixed bank
4000	7FFF	16 KiB ROM Bank 01–NN	From cartridge, switchable bank via mapper (if any)
8000	9FFF	8 KiB Video RAM (VRAM)	In CGB mode, switchable bank 0/1
A000	BFFF	8 KiB External RAM	From cartridge, switchable bank if any
C000	CFFF	4 KiB Work RAM (WRAM)	
D000	DFFF	4 KiB Work RAM (WRAM)	In CGB mode, switchable bank 1–7
E000	FDFF	Echo RAM (mirror of C000–DDFF)	Nintendo says use of this area is prohibited.
FE00	FE9F	Object attribute memory (OAM)	
FEA0	FEFF	Not Usable	Nintendo says use of this area is prohibited.
FF00	FF7F	I/O Registers	
FF80	FFFE	High RAM (HRAM)	
FFFF	FFFF	Interrupt Enable register (IE)	

SoC

gb, gbc, gba

- Game Boys use only a single integrated System-on-a-Chip (SoC)
- SoC includes the processor (CPU) core, some memories, and various peripherals
- The Game Boy SoC is sometimes called the “CPU”

More about CPU: https://gekkio.fi/files/gb-docs/gbctr.pdf
Photos: https://raphaelstaebler.medium.com/

CPU

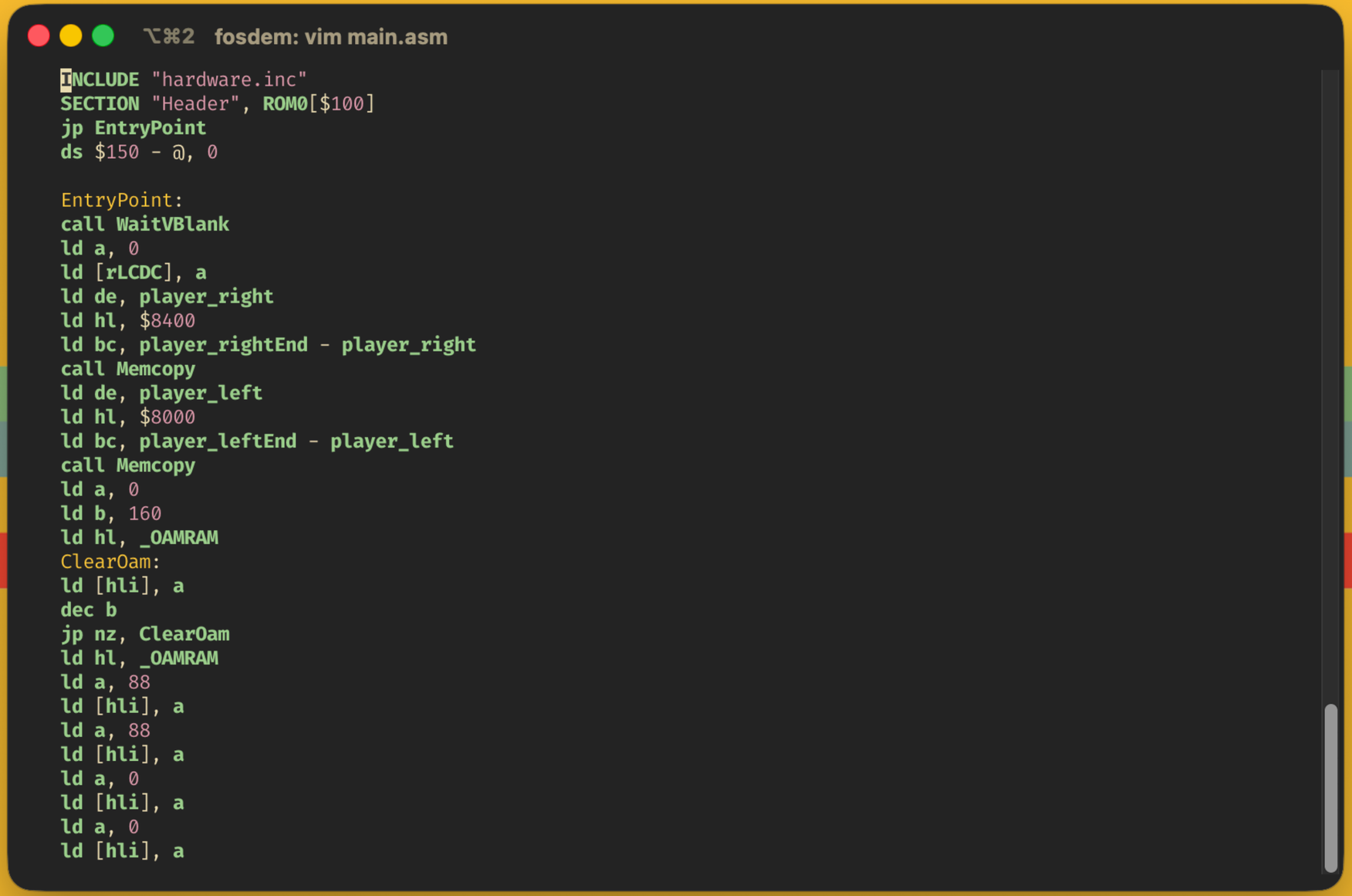
gb, gbc

- The CPU core in the Game Boy SoC is a custom Sharp design without a name.
- Some sources claim Game Boy uses a “modified” Zilog Z80 or Intel 8080.
- Using old datasheets and databooks, the core has been identified to be a **Sharp SM83.**

More about CPU: https://gekkio.fi/files/gb-docs/gbctr.pdf
Photos: https://www.copetti.org/writings/consoles/game-boy/

03. Make games

ASM



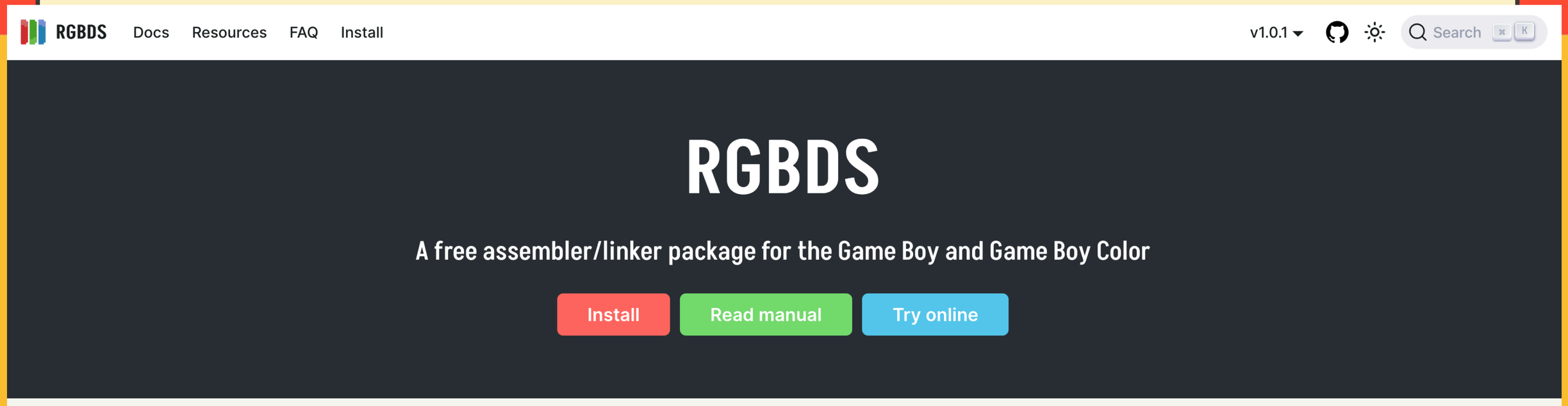
https://gbdev.io/gb-asm-tutorial/

ASM

Rednex Game Boy Development System

- Four programs to cover the whole compilation pipeline:
- Image converter / Assembler / Linker / Fixer

https://rgbds.gbdev.io/



C

GBDK-2020 - Game Boy Development Kit

gbdk-2020

An updated version of GBDK, C compiler, assembler, linker and set of libraries for the Nintendo Gameboy, Nintendo Entertainment System, Sega Master System, Sega Game Gear.

View the Project on GitHub
gbdk-2020/gbdk-2020

GBDK-2020

GBDK is a cross-platform development kit for sm83, z80 and 6502 based gaming consoles. It includes libraries, toolchain utilities and the SDCC C compiler suite.

Supported Consoles: (see docs)

- Nintendo Game Boy / Game Boy Color
- Analogue Pocket
- Sega Master System & Game Gear
- Mega Duck / Cougar Boy
- NES

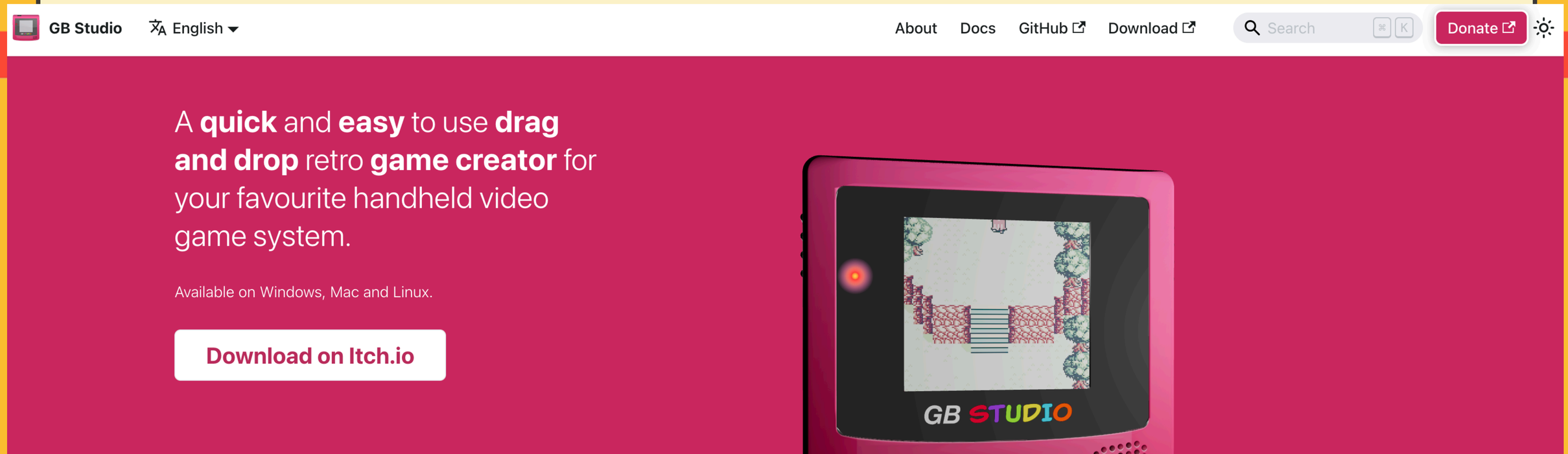
Experimental consoles (not yet fully functional)

- MSXDOS

https://gbdk.org/

GB Studio

- It is the most advanced retro game creator. It is a complete engine to create complete games. It is based on **GBDK** and **GBVM**
- **https://www.gbstudio.dev/**



04. ...and Rust?

What we have

Emulators! Emulators everywhere!

- **Mooneye GB** – A Game Boy research project and emulator written in Rust
  - Code: https://github.com/Gekkio/mooneye-gb
- **Boytacean** – Full-featured Rust emulator with Web, SDL & Libretro frontends
  - Code: https://github.com/joamag/boytacean
- **Retro Boy –** Cycle-accurate emulator compiled to WebAssembly
  - Code: https://github.com/smparsons/retroboy
- **Wasm-GB –** Game Boy emulator in WebAssembly + WebGL 2.0 (Rust)
  - Code: https://github.com/andrewimm/wasm-gb
- **gameboy** – Game Boy emulator written in Rust
  - Code: https://github.com/raphamorim/gameboy

What we have

Crates for gba

- There are some crates to make games for gba:
  - **gba**
  - **agb**
  - Others project educational / list of utilities

- As said, the gba has a different architecture and a different processor. gba have an ARM CPU.

Rust-GB

By zlfn

- Is a project work in progress...
- And try to obtain the build with a “workaround.”
  1. The **Rust** compiler can generate **LLVM-IR** for the **ATMega328**
  2. **LLVM-IR** can be converted in **C** with *llvm-cbe*
  3. **C** compiled to **Z80 assembly** with *sdasgb*
  4. **Z80 Assembly** can be assembled into **GBZ80** with *sdasgb*
  5. **GBZ80 object code** can be linked in a **ROM gb** with *GBDK*
- There is another project by zlfn: cranelift-z80

https://github.com/zlfn/rust-gb

https://github.com/zlfn/cranelift-z80

05. Rustboy

The idea

- We have in front of us only one CPU…
- Can we do a specific Rust compiler for the **SM83**?
- **Great idea!** I always develop a compiler!

https://github.com/ffex/rust-boy

PoC

Proof of concept

RustBoy

- rust_boy
- gb_std
- gb_asm

To speed up the development, I put a solid working base.

- ASM
- RGBDS

How show results?

Unbricked - an Arkanoid copy

- In gbdev.io, as an example to illustrate how to create games in asm, the initial example is a copy of the famous Arkanoid.
- This example is important to see what happens when we go up to the high level to the code
- So let me explain some part of this game in asm

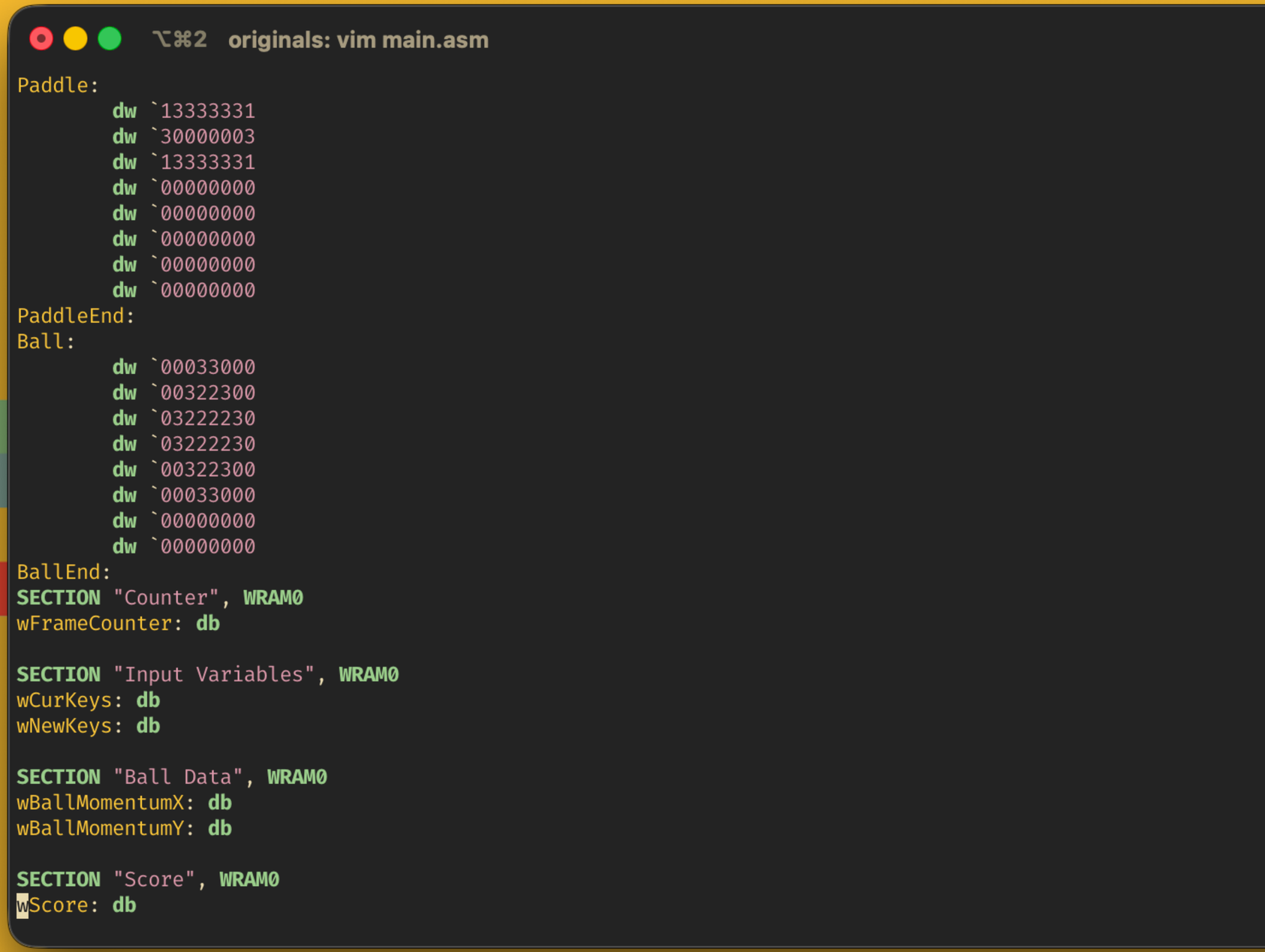
Unbricked

Inits

```
⌥⌘2 originals: vim main.asm
INCLUDE "hardware.inc"

DEF BRICK_LEFT EQU $05
DEF BRICK_RIGHT EQU $06
DEF BLANK_TILE EQU $08
DEF DIGIT_OFFSET EQU $1A
DEF SCORE_TENS EQU $9870
DEF SCORE_ONES EQU $9871

SECTION "Header", ROM0[$100]

jp EntryPoint

ds $150 - @, 0 ; room for header

EntryPoint:

WaitVBlank:
    ld a, [rLY]
    cp 144
    jp c, WaitVBlank

    ; Turn off LCD
    ld a, 0
    ld [rLCDC], a

    ; Copy tiles data
    ld de, Tiles
    ld hl, $9000
    ld bc, TilesEnd - Tiles
    call Memcopy

    ; Copy the tilemap
"main.asm" 757L, 15183B
```

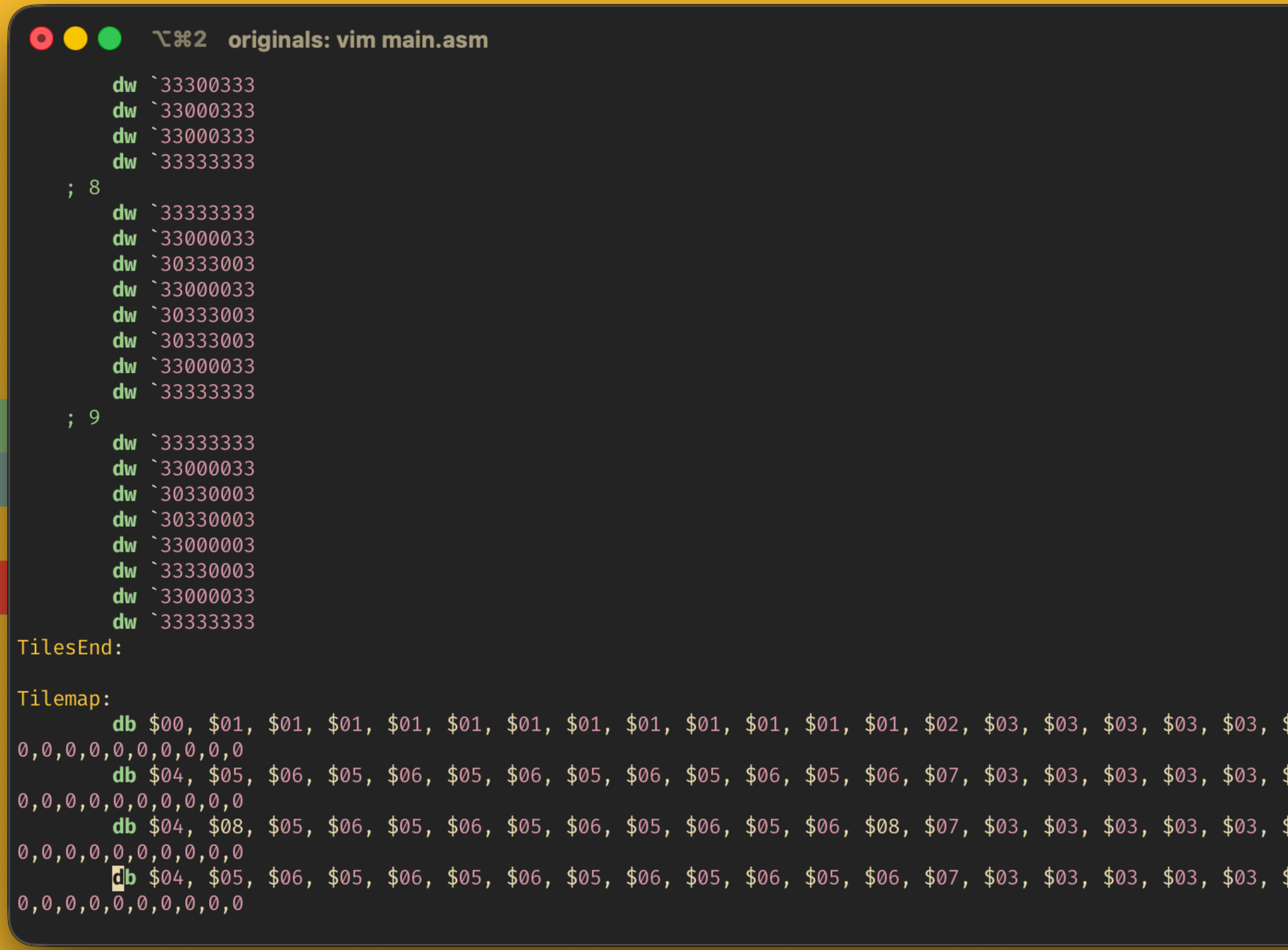
Unbricked

Variables



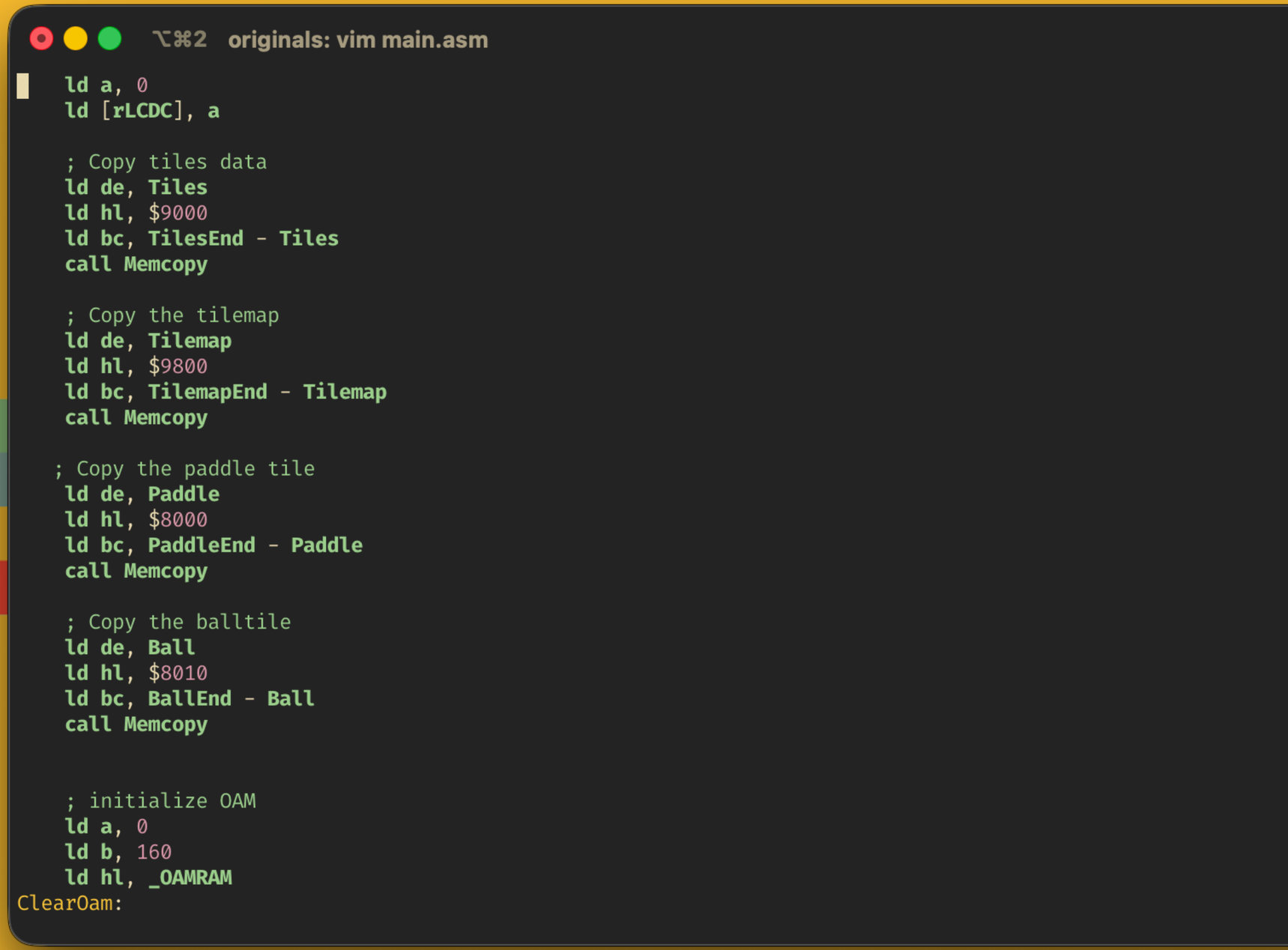
Unbricked

Tiles and Tilemap



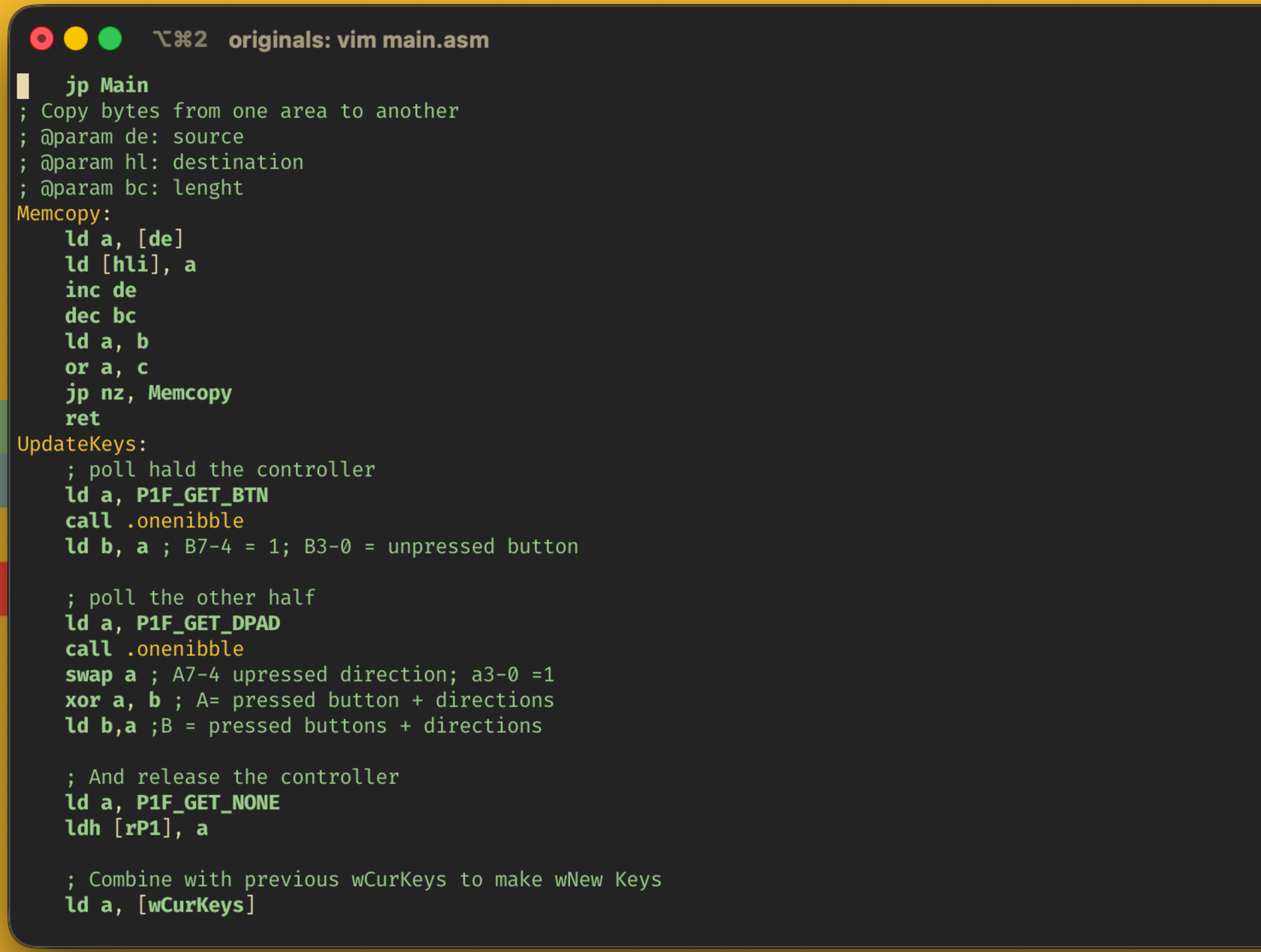
Unbricked

Memcopy



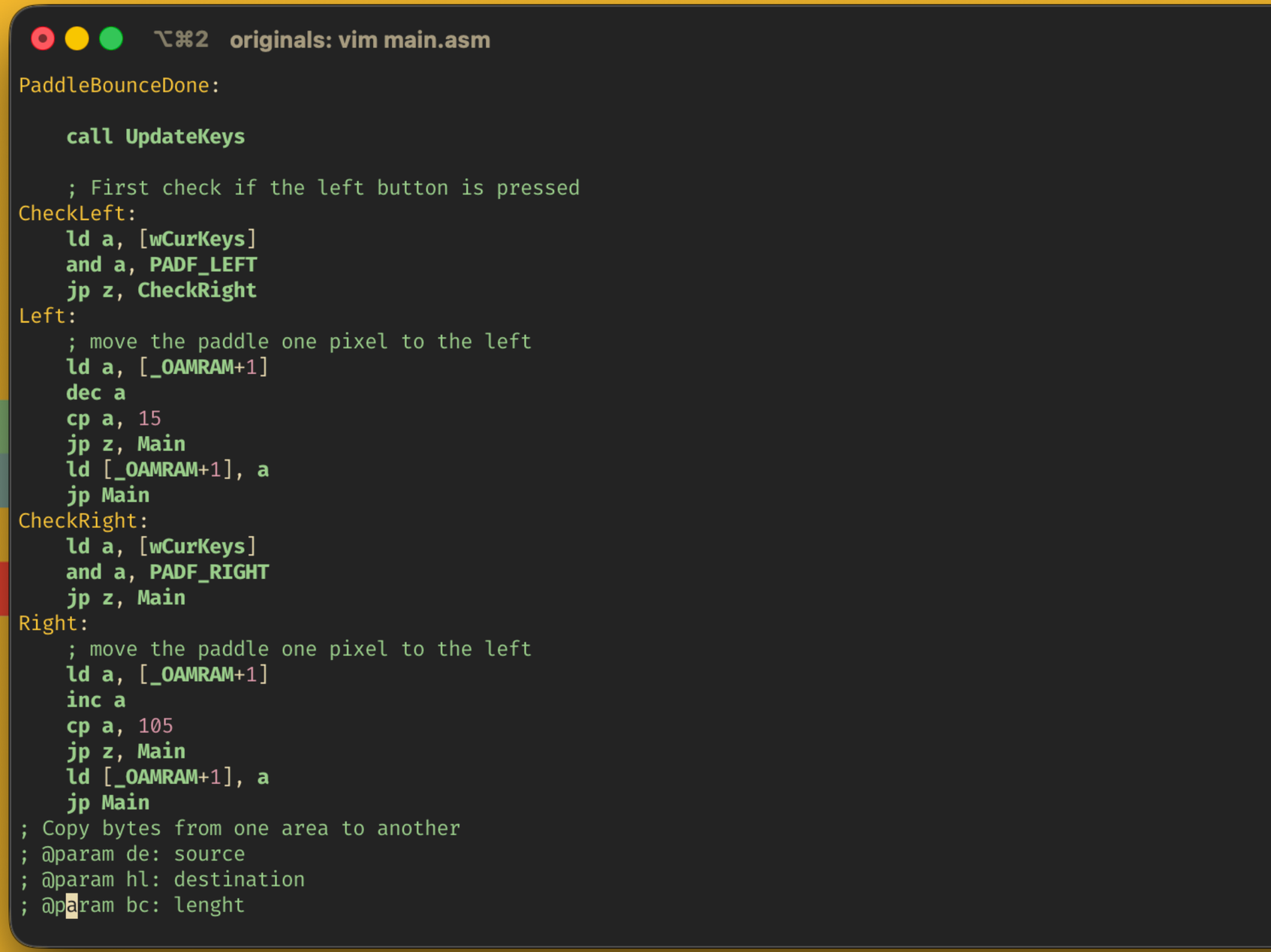
Unbricked

Functions



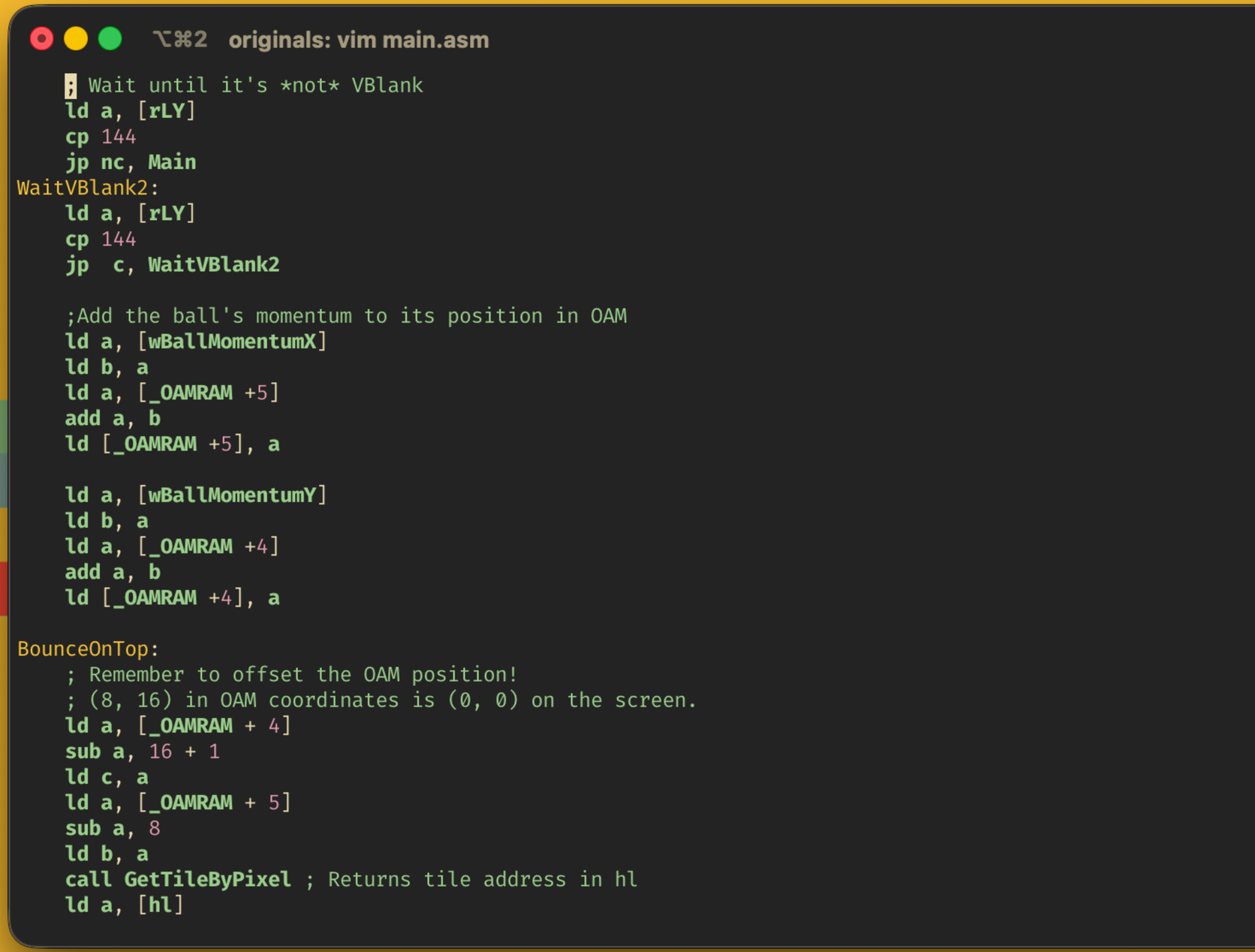
Unbricked

Main loop - Input



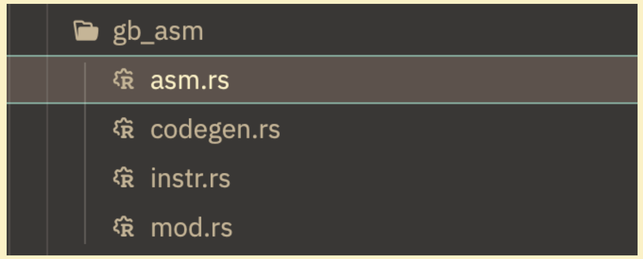
Unbricked

Main loop - Movement



gb_asm

- It is the most low-level library.
- Almost one-to-one with the asm but in Rust!

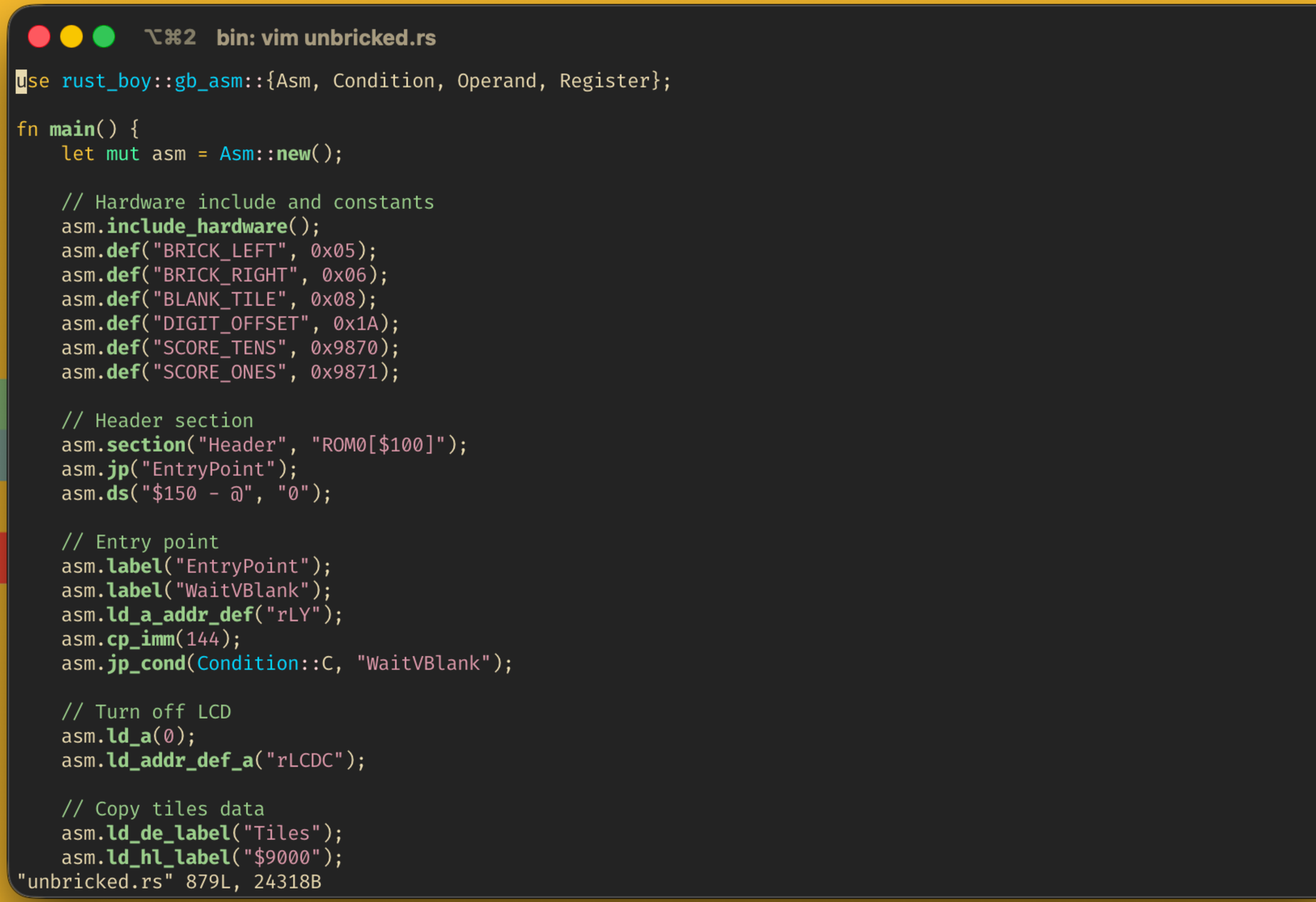


- Instructions are something like: asm.ld(...), asm.cp(...), ...
- Link to an article that inspire me:

https://tinycomputers.io/posts/building-z80-roms-with-rust-a-modern-approach-to-retro-computing.html

gb_asm

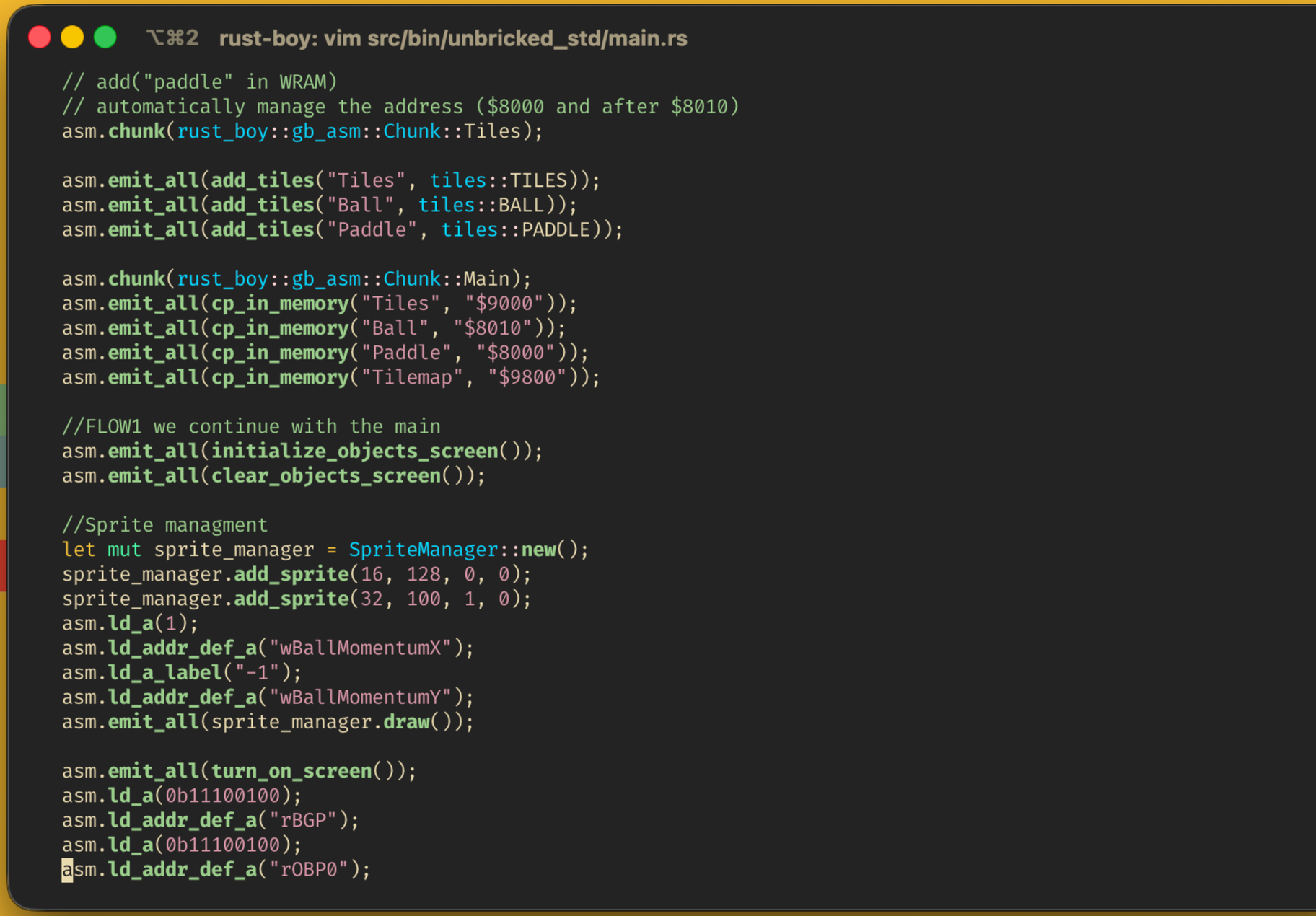
unbricked.rs



gb_std

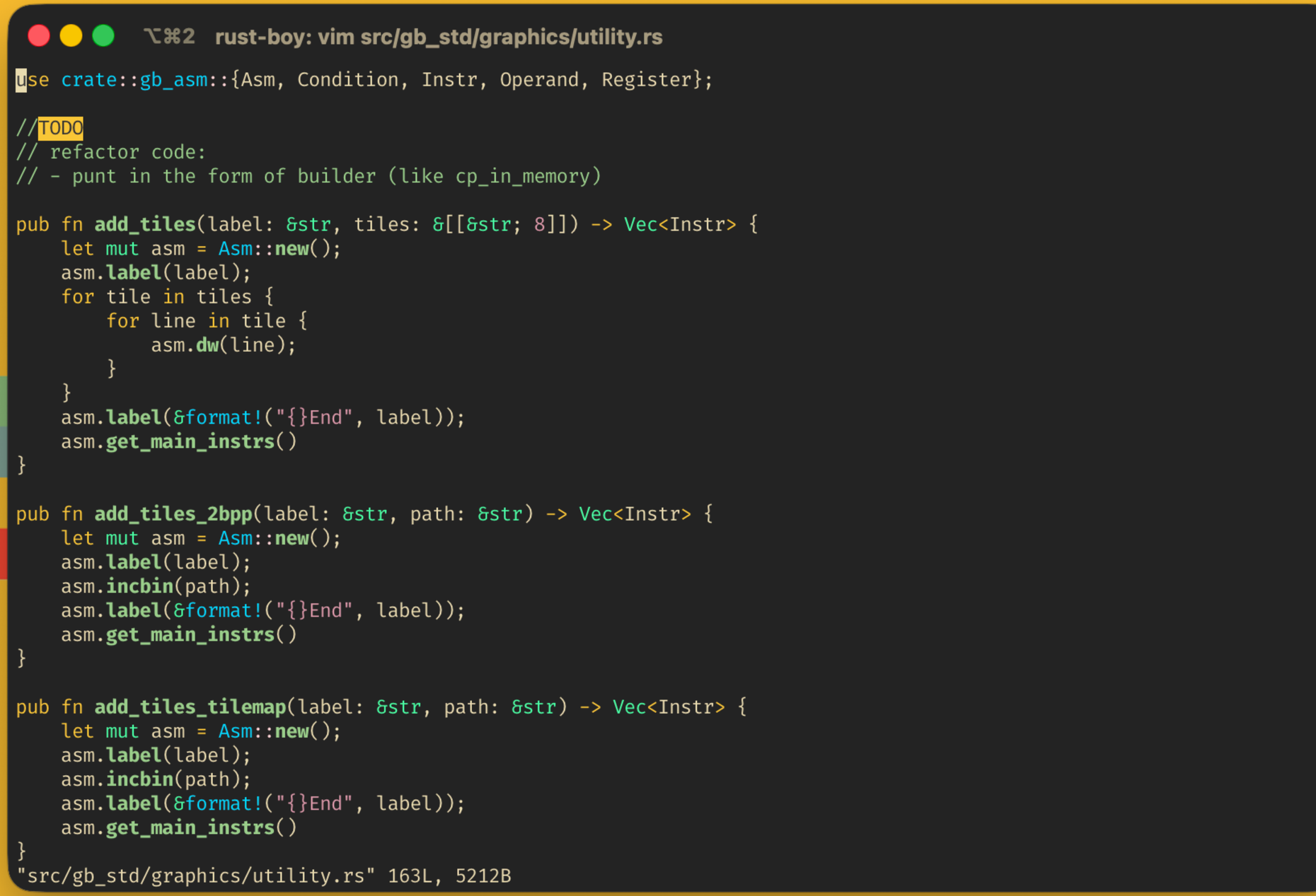
- This lib is more of a high-level and implements:
  - A chunk system (Main, Functions, Tiles, etc.) so you can put code from everywhere, and at the time of the generation are put in the right section.
  - Tile and tilemap utilities
  - A sprite manager
  - The initial attempt at an if statement.

gb_std


⌥⌘2 rust-boy: vim src/bin/unbricked_std/main.rs
// add("paddle" in WRAM)
// automatically manage the address ($8000 and after $8010)
asm.chunk(rust_boy::gb_asm::Chunk::Tiles);
asm.emit_all(add_tiles("Tiles", tiles::TILES));
asm.emit_all(add_tiles("Ball", tiles::BALL));
asm.emit_all(add_tiles("Paddle", tiles::PADDLE));
asm.chunk(rust_boy::gb_asm::Chunk::Main);
asm.emit_all(cp_in_memory("Tiles", "$9000"));
asm.emit_all(cp_in_memory("Ball", "$8010"));
asm.emit_all(cp_in_memory("Paddle", "$8000"));
asm.emit_all(cp_in_memory("Tilemap", "$9800"));
//FLOW1 we continue with the main
asm.emit_all(initialize_objects_screen());
asm.emit_all(clear_objects_screen());
//Sprite managment
let mut sprite_manager = SpriteManager::new();
sprite_manager.add_sprite(16, 128, 0, 0);
sprite_manager.add_sprite(32, 100, 1, 0);
asm.ld_a(1);
asm.ld_addr_def_a("wBallMomentumX");
asm.ld_a_label("-1");
asm.ld_addr_def_a("wBallMomentumY");
asm.emit_all(sprite_manager.draw());
asm.emit_all(turn_on_screen());
asm.ld_a(0b11100100);
asm.ld_addr_def_a("rBGP");
asm.ld_a(0b11100100);
asm.ld_addr_def_a("rOBP0");

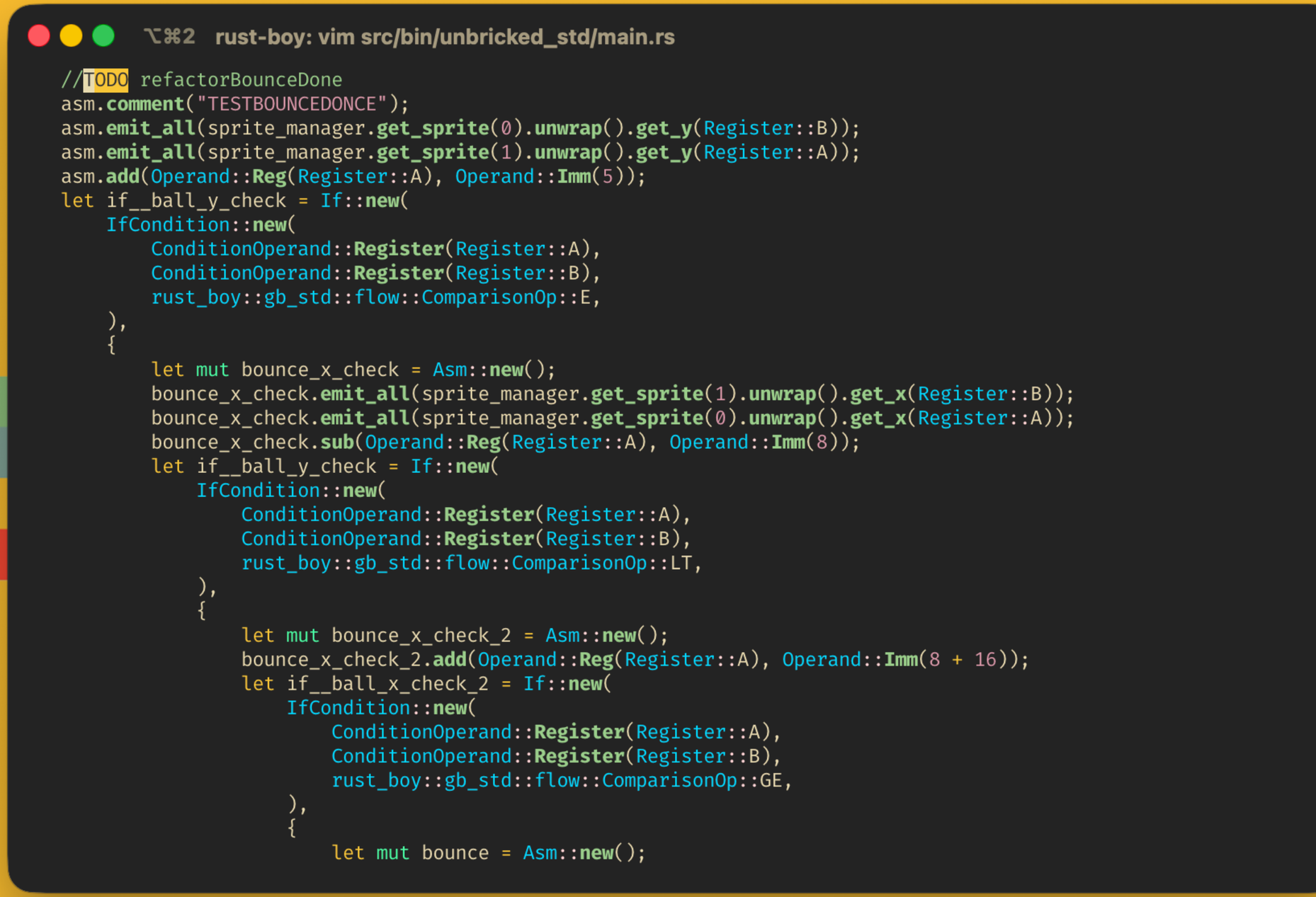
gb_std

Utilities



gb_std

If statement

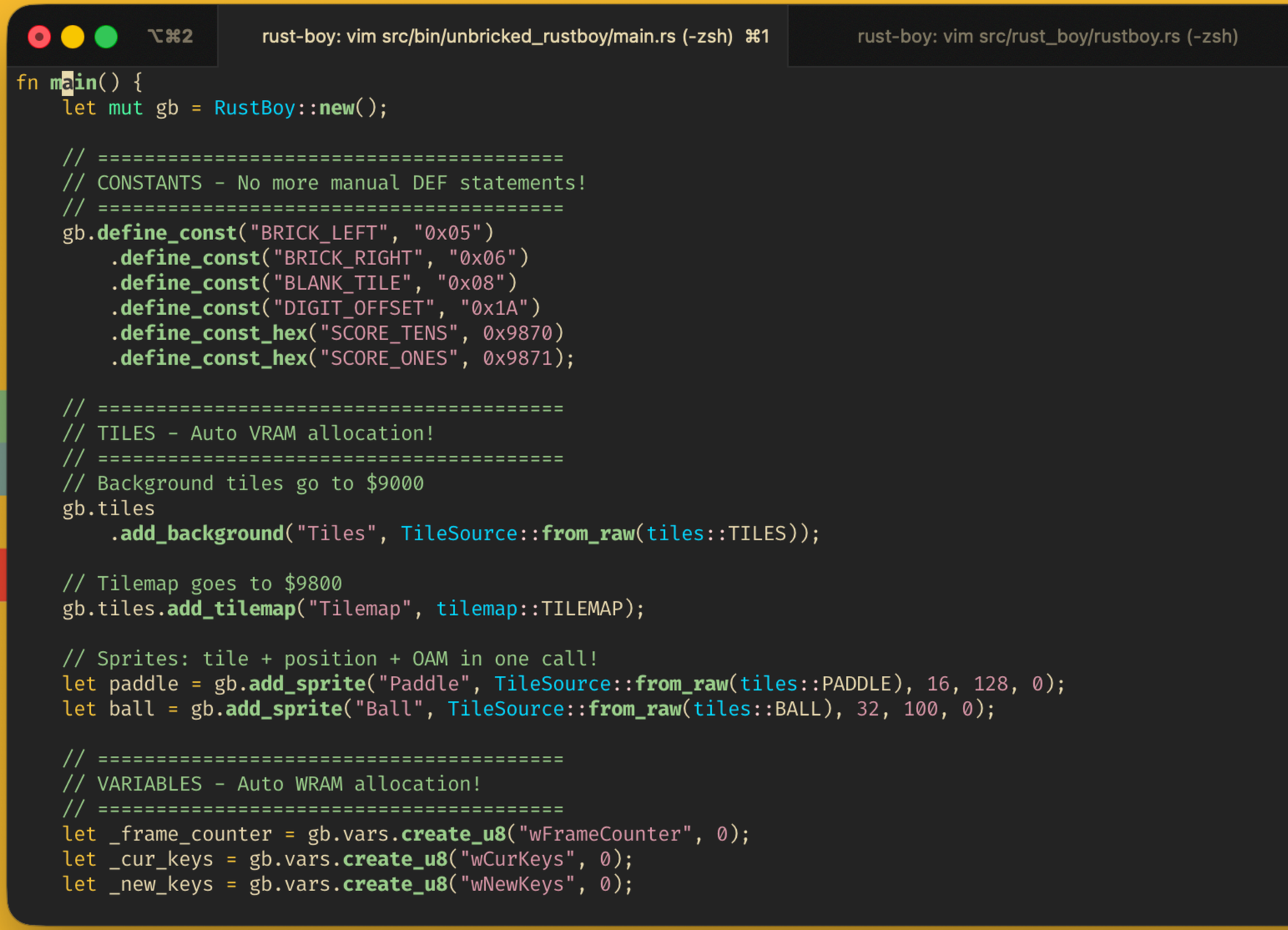


rust_boy

- With gb_std we understood what we can level up:
  - If statements have to be simpler
  - The init instructions must be written in an automatic way
  - There are some functions that can be considered “BuiltIn (UpdateKeys, WaitVBlank, Memcopy, etc.)
  - Make more managers (Tiles, Inputs, etc.)
  - Hide every reference to the memory address

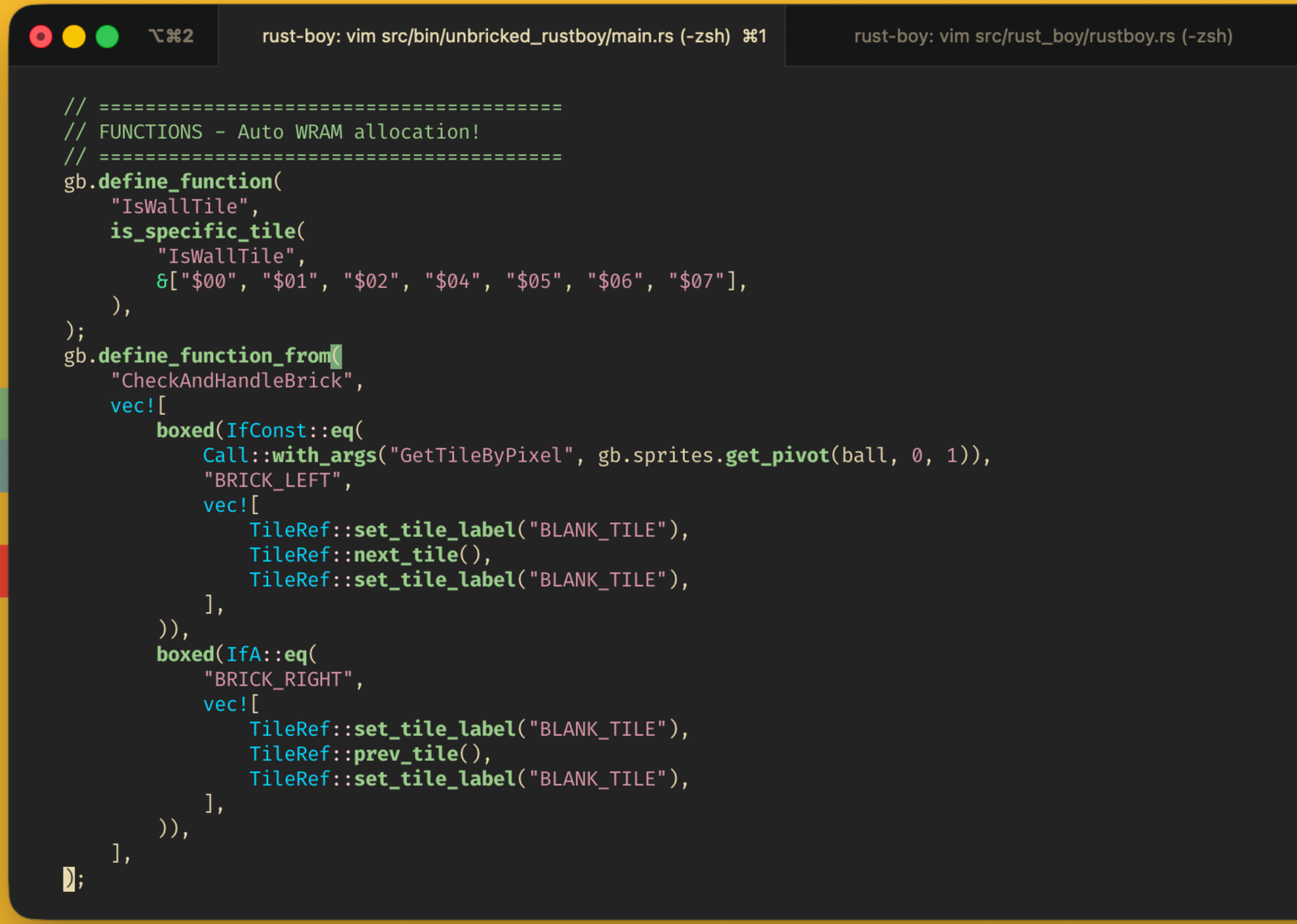
rust_boy

unbricked.rs



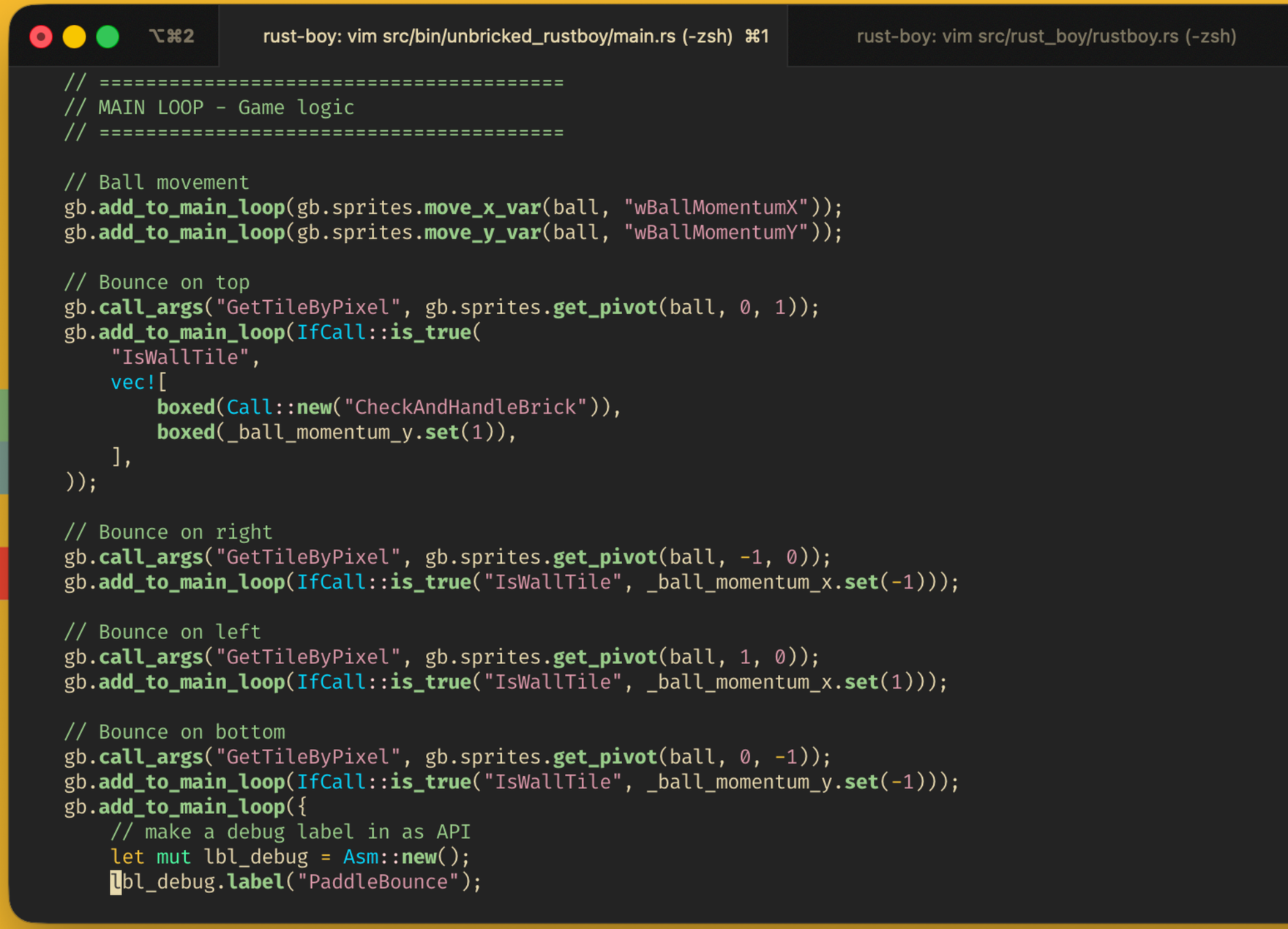
rust_boy

unbricked.rs



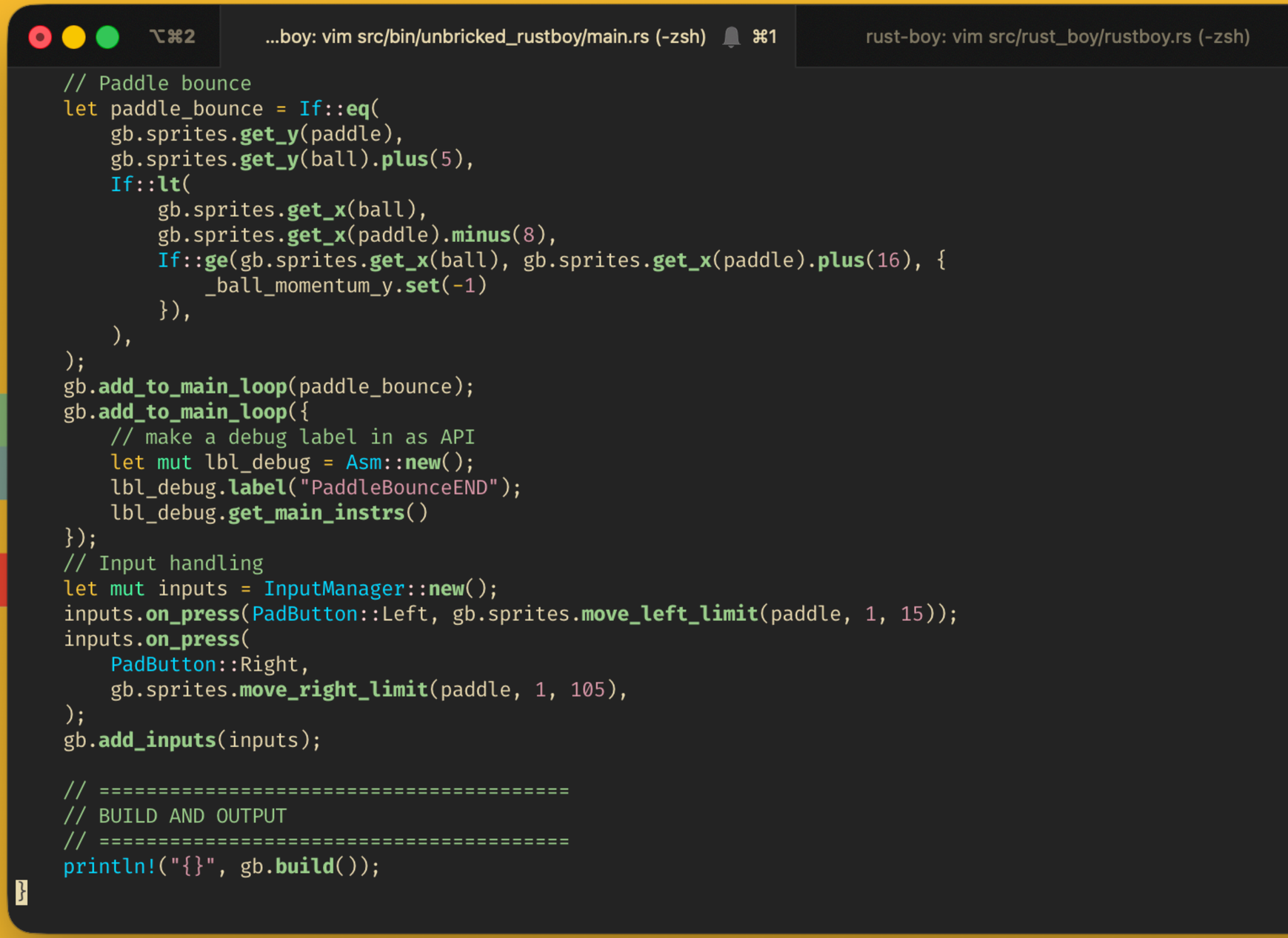
rust_boy

unbricked.rs



rust_boy

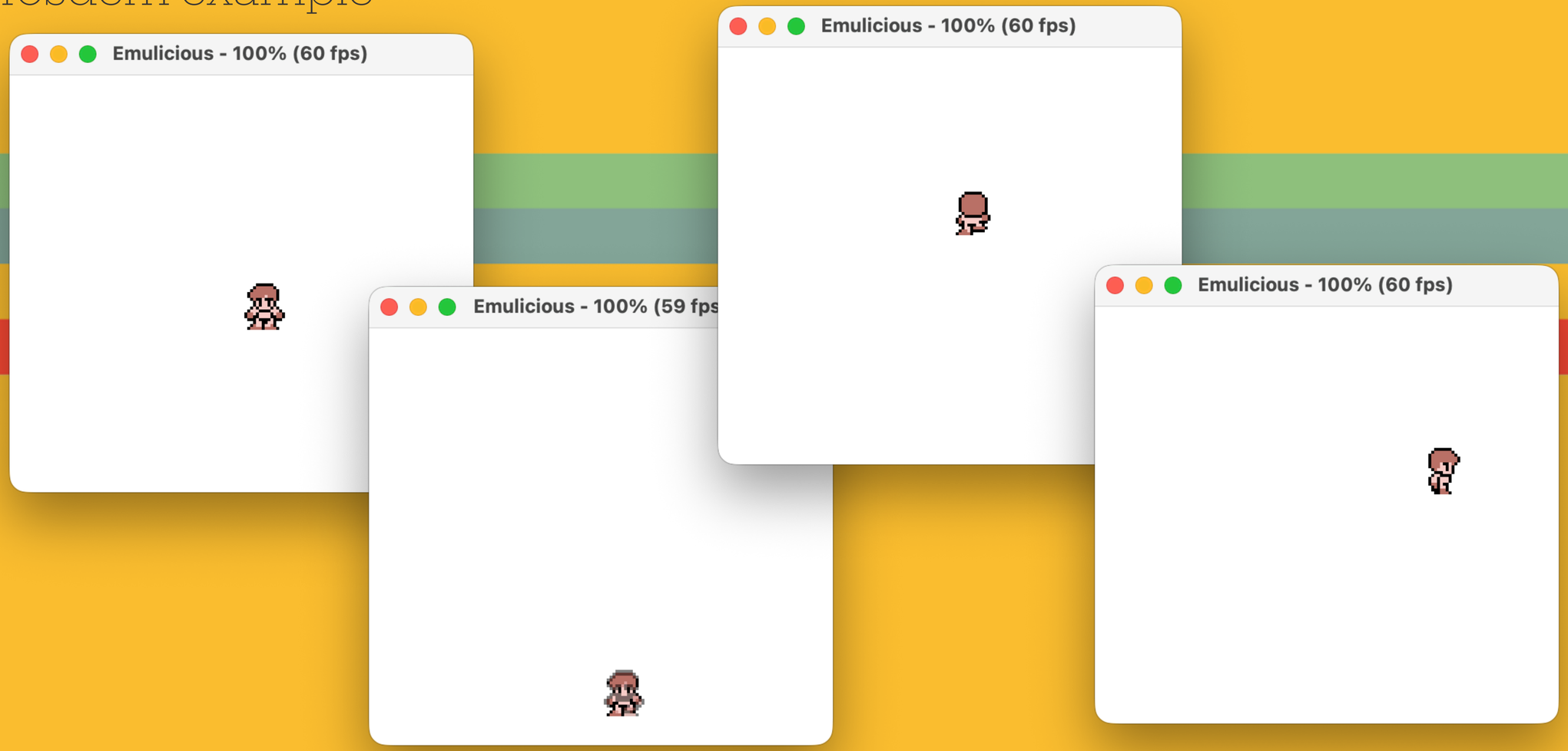
unbricked.rs




```
    // Paddle bounce
    let paddle_bounce = If::eq(
        gb.sprites.get_y(paddle),
        gb.sprites.get_y(ball).plus(5),
        If::lt(
            gb.sprites.get_x(ball),
            gb.sprites.get_x(paddle).minus(8),
            If::ge(gb.sprites.get_x(ball), gb.sprites.get_x(paddle).plus(16), {
                _ball_momentum_y.set(-1)
            }),
        ),
    );
    gb.add_to_main_loop(paddle_bounce);
    gb.add_to_main_loop({
        // make a debug label in as API
        let mut lbl_debug = Asm::new();
        lbl_debug.label("PaddleBounceEND");
        lbl_debug.get_main_instrs()
    });
    // Input handling
    let mut inputs = InputManager::new();
    inputs.on_press(PadButton::Left, gb.sprites.move_left_limit(paddle, 1, 15));
    inputs.on_press(
        PadButton::Right,
        gb.sprites.move_right_limit(paddle, 1, 105),
    );
    gb.add_inputs(inputs);

    // ========================================
    // BUILD AND OUTPUT
    // ========================================
    println!("{}", gb.build());
}
```

Try rust_boy!

The fosdem example



rust_boy

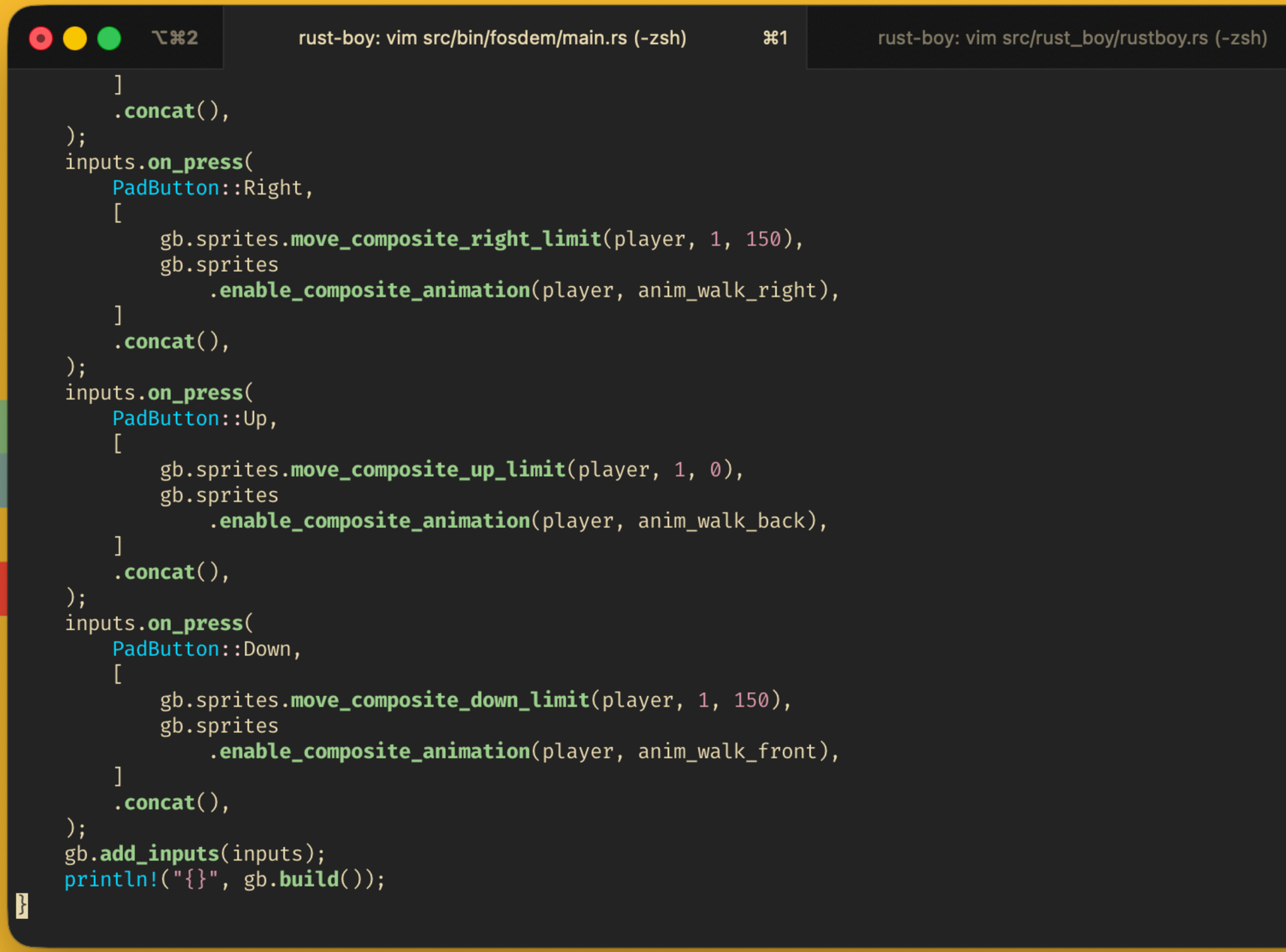
at Fosdem!

rust-boy: vim src/bin/fosdem/main.rs (-zsh) ⌘1 | rust-boy: vim src/rust_boy/rustboy.rs (-zsh)

```rust
fn main() {
    let mut gb = RustBoy::new();

    // Add 16x16 composite sprite (two 8x16 sprites side by side)
    let player = gb.add_sprite_16x16(
        "player",
        TileSource::from_file("char.2bpp", 64),
        TileSource::from_file("char-dx.2bpp", 64),
        80,
        72,
        0,
    );

    // Add looping animations to the composite sprite (applies to both halves)
    // add_composite_animation returns the animation index
    // Animation order: front, back, left, right (frames 0-3, 4-7, 8-11, 12-15)
    let anim_walk_front =
        gb.sprites
            .add_composite_animation(player, "playerWalkFront", 0, 3, AnimationType::Loop);
    let anim_walk_back =
        gb.sprites
            .add_composite_animation(player, "playerWalkBack", 4, 7, AnimationType::Loop);
    let anim_walk_left =
        gb.sprites
            .add_composite_animation(player, "playerWalkLeft", 8, 11, AnimationType::Loop);
    let anim_walk_right =
        gb.sprites
            .add_composite_animation(player, "playerWalkRight", 12, 15, AnimationType::Loop);

    // Start with no animation (disabled)
    gb.sprites
        .set_composite_initial_animation(player, ANIM_DISABLED);
```

rust_boy

at Fosdem!



What's next?

Improvment

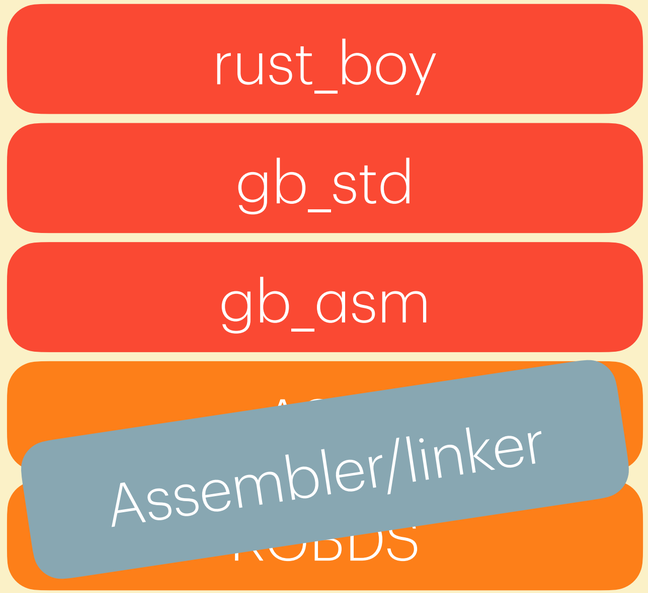
- Refactor the code:
  - It is written in a cumulative way so there are some optimization to do.
  - Try to write new examples to find new built-in functions
    - Example: implement the sound manager.

What's next?

At low-level

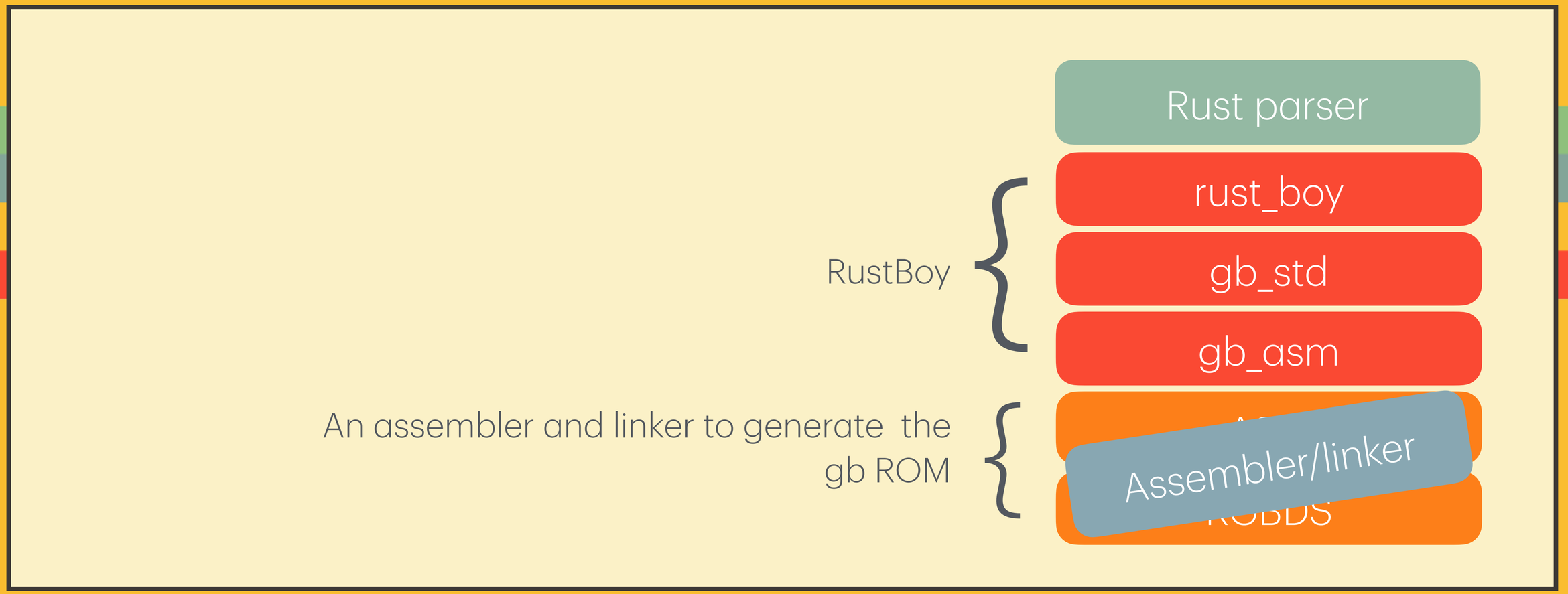
RustBoy

An assembler and linker to generate the gb ROM

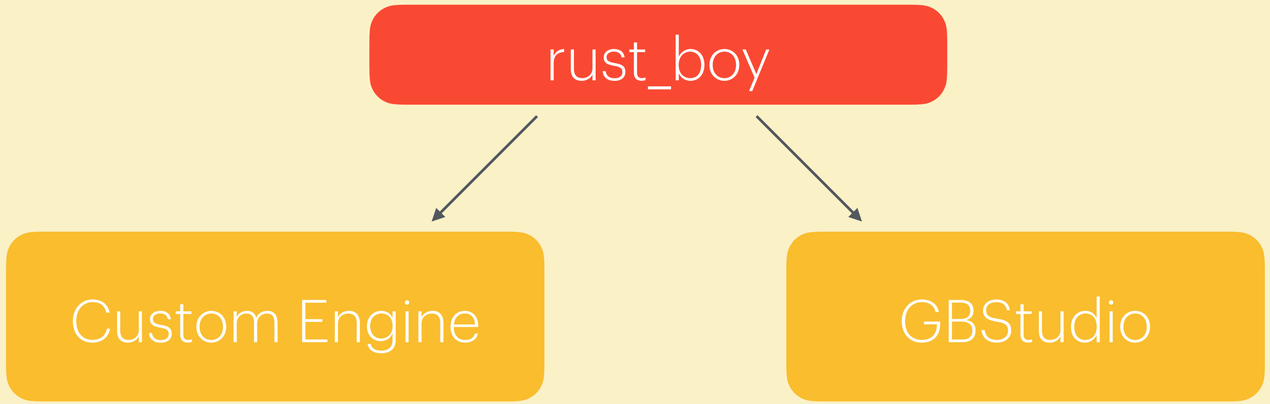


What's next?

High-level



One more thing…



Thank you

https://github.com/ffex/rust-boy

https://github.com/ffex

https://www.linkedin.com/in/federico-bassini/

https://mastodon.social/@ffex

https://www.instagram.com/ffex_tech/