从零开始实现GameBoy模拟器 #10 按键输入、MBC1卡带

銀葉吉祥
浙江大学 软件工程硕士

欢迎来到从零开始实现GameBoy模拟器第十章。在本章中，我们实现GameBoy的按钮输入逻辑，并扩展卡带读写的功能，从而支持MBC1的卡带。那么，让我们开始吧！

> 本章节代码已经上传至项目仓库，名称为LunaGB-10，已经下载项目代码的读者可以更新代码库来查阅最新代码。同时，欢迎大家加入GameBoy模拟器开发交流群：33854734
> 2024.2.26更新：增加了mbc1_write访问非法地址的检测。
> 2024.3.3更新：修复了卡带非法地址读写时错误日志的格式问题。

按键输入

0xFF00 - P1

GameBoy的所有按键输入检测均通过0xFF00 P1这一个寄存器完成。在GameBoy上，总共有8个游戏可以使用的按键，分别是：

- 方向键（上、下、左、右）
- A键
- B键
- SELECT键
- START键

P1寄存器分为两个区域，低四位为只读区域，用于程序读取按键输出，而第5位和第6位为读写区域，用于选择将哪四个按钮映射到寄存器的低四位上。P1每个位的含义如下表所示：

7	6	5	4	3	2	1	0
-	-	选择功能键	选择方向键	按钮3状态（只读）	按钮2状态（只读）	按钮1状态（只读）	按钮0状态（只读）

P1.4和P1.5都以**0**为选择，**1**为不选择。例如，当选择方向键的值为0，选择功能键的值为1时，P1的低4位会映射到四个方向键的输入状态：

7	6	5	4	3	2	1	0
-	-	1	0	下	上	左	右

当选择方向键的值为1，选择功能键的值为0时，P1的低4位会映射到四个功能键的输入状态：

7	6	5	4	3	2	1	0
-	-	0	1	START	SELECT	B	A

对于上述的8个按键，**当按键按下时，对应的位读取的值为0，否则为1**。当程序需要检测按钮输入时，其首先需要向0xFF00地址写入一个数字，这个数字的第5位和第6位中需要有一位为1，另一位为0，从而选择需要将哪四个按钮的输入状态映射到0xFF00的低四位上。接着，程序需要读取0xFF00的值，并将其与0x0F做按位与操作，从而取出四个按钮的状态，进行进一步的判断。在读取按钮状态时，如果P1.4和P1.5的值均为0，则低四位固定会读取出0xF值，即所有按键都没有按下。P1.4和P1.5的值不可同时为1，否则读取的值无效，实际的游戏会避免将P1.4和P1.5的值同时设置为1。

输入抖动

由于GameBoy按钮的物理性质，当玩家按下按钮时，程序从0xFF00中读取的值并不是立即从1变为0，而是会在1和0之间反复变化若干次，最后才稳定为0；同样，当玩家释放按钮时，值也会在0和1之间变化若干次，最后才稳定为1，该现象被称为输入抖动（switch bounce）。为了避免输入抖动造成的重复输入，大部分游戏在设置0xFF00的按钮选择位以后，都会读取0xFF00的值若干次，并在每两次读取中都加入一定的延迟，从而让按钮的输入数值稳定，然后取最后一次的数值作为按钮的最终状态。

由于每次选中的四个按钮的电平从1变为0时都会由硬件触发INT_JOYPAD中断，而因为输入抖动的存在，玩家在每一次按下按钮时，都会触发多次INT_JOYPAD中断，因此INT_JOYPAD中断在大部分场合下都无法很好地用于检测用户输入。在实践中，几乎所有游戏都采用了轮询法（每隔固定时间读取一次0xFF00的值）而不是INT_JOYPAD中断来响应按键输入。

输入抖动问题在实现模拟器时并不存在，因为正常电脑上的键盘和手柄驱动都会帮我们处理好输入抖动的问题。上述原理只是作为背景资料供读者参考。

实现按键输入

在了解了输入的工作原理后，我们就可以编程实现GameBoy的按键输入了。GameBoy的按键输入实现分为两个部分，首先在模拟器中，我们需要一个按键状态表来记录当前每个按键的状态，当程序读取0xFF00的值时，模拟器会根据该状态表来确定应该返回的值；然后在模拟器外部，我们需要在应用程序的主循环中检测电脑键盘和游戏手柄的输入，并在用户按下键盘按键和手柄按键时更新模拟器的按键状态，从而将程序的输入映射为模拟器的输入。

我们首先实现模拟器中的部分。新建Joypad.hpp和Joypad.cpp来保存按键输入部分的代码，Joypad.hpp的代码如下：

```
#pragma once
#include <Luna/Runtime/MemoryUtils.hpp>
using namespace Luna;

struct Emulator;
struct Joypad
{
    bool a;
    bool b;
    bool select;
    bool start;
    bool right;
    bool left;
    bool up;
    bool down;
    //! 0xFF00
    u8 p1;

    void init();
    u8 get_key_state() const;
    void update(Emulator* emu);
    u8 bus_read();
    void bus_write(u8 v);
};
```

我们定义了一个Joypad类，并在类中为每一个按键定义了一个bool变量，来表示该按键是否按下。除了这些按键状态变量以外，我们还定义了一个p1变量，这个变量用于保存0xFF00 P1寄存器的值。接着我们定义了一系列用于操作Joypad类的方法，例如与其余组件类似的init、bus_read和bus_write方法（由于Joypad总共就一个寄存器，所以这里不需要传入addr参数），以及在下面会介绍的一些新的方法。

Joypad.cpp的代码如下所示：

```
#include "Joypad.hpp"
```

```
#include "Emulator.hpp"

void Joypad::init()
{
    memzero(this);
    p1 = 0xFF;
}
u8 Joypad::get_key_state() const
{
    u8 v = 0xFF;
    if(!bit_test(&p1, 4))
    {
        if(right) bit_reset(&v, 0);
        if(left) bit_reset(&v, 1);
        if(up) bit_reset(&v, 2);
        if(down) bit_reset(&v, 3);
        bit_reset(&v, 4);
    }
    if(!bit_test(&p1, 5))
    {
        if(a) bit_reset(&v, 0);
        if(b) bit_reset(&v, 1);
        if(select) bit_reset(&v, 2);
        if(start) bit_reset(&v, 3);
        bit_reset(&v, 5);
    }
    return v;
}
void Joypad::update(Emulator* emu)
{
    u8 v = get_key_state();
    // Any button is pressed in this update.
    if((bit_test(&p1, 0) && !bit_test(&v, 0)) ||
    (bit_test(&p1, 1) && !bit_test(&v, 1)) ||
    (bit_test(&p1, 2) && !bit_test(&v, 2)) ||
    (bit_test(&p1, 3) && !bit_test(&v, 3)))
    {
        emu->int_flags |= INT_JOYPAD;
    }
    p1 = v;
}
u8 Joypad::bus_read()
{
    return p1;
}
void Joypad::bus_write(u8 v)
{
    // Only update bit 4 and bit 5.
    p1 = (v & 0x30) | (p1 & 0xCF);
    // Refresh key states.
    p1 = get_key_state();
}
```

get_key_state函数用于根据当前P1.4和P1.5和各个按键变量的状态返回一个新的p1变量值，用于在update和bus_write中更新寄存器。在get_key_state中，我们首先判断P1.4和P1.5的状态，以确定我们需要读取哪四个按键的状态，然后根据P1的选择结果和对应的按键状态将返回值的对应位从1设置为0，并返回设置完毕的结果。由于返回值的初始值是0xFF，因此如果P1.4和P1.5都是1，即都没有选中的话，我们会直接返回0xFF。

update函数在每一帧调用一次，因为我们在一帧中只会改变一次输入状态，因此和别的组件不同，我们不需要在每个时钟周期中都更新Joypad的状态。在update函数中，我们会首先调用get_key_state函数，根据当前的按键状态获取新的P1值，然后将新的P1和之前的P1值比较，以确定是否有任意按钮被按下（对应的值从1变成了0），并在按下的情况下设置JOYPAD中断标志位。最后，我们会用新的P1值更新p1变量。

bus_read和bus_write函数用于读写p1变量。在bus_write中，由于只有P1.4和P1.5是可读写的，因此我们使用位运算屏蔽对其余位的值的修改，并在修改P1变量以后调用get_key_state重新设置P1的值，因此当程序修改了P1.4和P1.5的值后，P1的低四位的值会得到重新映射。

最后，我们修改Emulator类，将Joypad组件加入到模拟器中：

Emulator.hpp：

```
struct Emulator
{
    /*...*/
    PPU ppu;
```

```
    Joypad joypad;

    /*...*/
};
```

Emulator.cpp：

```
RV Emulator::init(const void* cartridge_data, usize cartridge_data_size)
{
    /*...*/
    ppu.init();
    joypad.init();
    return ok;
}
void Emulator::update(f64 delta_time)
{
    joypad.update(this);
    u64 frame_cycles = (u64)((f32)(4194304.0 * delta_time) * clock_speed_scale)
    u64 end_cycles = clock_cycles + frame_cycles;
    while(clock_cycles < end_cycles)
    {
        if(paused) break;
        cpu.step(this);
    }
}
u8 Emulator::bus_read(u16 addr)
{
    /*...*/
    if(addr >= 0xFE00 && addr <= 0xFE9F)
    {
        return oam[addr - 0xFE00];
    }
    if(addr == 0xFF00)
    {
        return joypad.bus_read();
    }
    if(addr >= 0xFF01 && addr <= 0xFF02)
    {
        return serial.bus_read(addr);
    }
    /*...*/
}
void Emulator::bus_write(u16 addr, u8 data)
{
    /*...*/
    if(addr >= 0xFE00 && addr <= 0xFE9F)
    {
        oam[addr - 0xFE00] = data;
        return;
    }
    if(addr == 0xFF00)
    {
        joypad.bus_write(data);
        return;
    }
    if(addr >= 0xFF01 && addr <= 0xFF02)
    {
        serial.bus_write(addr, data);
        return;
    }
    /*...*/
}
```

接着我们需要修改我们的程序框架，以在每一帧更新Joypad类中各个按钮bool变量的值。在App类中添加一个单独的函数来更新模拟器的输入：

```
void update_emulator_input();
```

在App.cpp中，我们首先需要引入两个新的头文件：

```
#include <Luna/HID/Keyboard.hpp>
#include <Luna/HID/Controller.hpp>
```

这两个头文件属于LunaSDK的HID（Human Interface Device）模块，该模块用于读写平台的各种外部设备状态，尤其是输入设备的状态。从头文件的名字不难发现，这两个头文件分别用于获

取键盘和控制器（游戏手柄）的输入。接着我们在App.cpp中实现update_emulator_input函数：

```
void App::update_emulator_input()
{
    auto& joypad = emulator->joypad;
    if(window->is_focused())
    {
        joypad.up = HID::get_key_state(HID::KeyCode::w) || HID::get_key_state(H
        joypad.left = HID::get_key_state(HID::KeyCode::a) || HID::get_key_state
        joypad.down = HID::get_key_state(HID::KeyCode::s) || HID::get_key_state
        joypad.right = HID::get_key_state(HID::KeyCode::d) || HID::get_key_stat
        joypad.a = HID::get_key_state(HID::KeyCode::j);
        joypad.b = HID::get_key_state(HID::KeyCode::k);
        joypad.select = HID::get_key_state(HID::KeyCode::spacebar);
        joypad.start = HID::get_key_state(HID::KeyCode::enter);
        if (HID::supports_controller())
        {
            auto gc_input = HID::get_controller_state(0);
            if (gc_input.connected)
            {
                joypad.up = joypad.up || test_flags(gc_input.buttons, HID::Cont
                joypad.up = joypad.up || (gc_input.axis_ly >= 0.5f);
                joypad.left = joypad.left || test_flags(gc_input.buttons, HID::
                joypad.left = joypad.left || (gc_input.axis_lx <= -0.5f);
                joypad.down = joypad.down || test_flags(gc_input.buttons, HID::
                joypad.down = joypad.down || (gc_input.axis_ly <= -0.5f);
                joypad.right = joypad.right || test_flags(gc_input.buttons, HID
                joypad.right = joypad.right || (gc_input.axis_lx >= 0.5f);
                joypad.a = joypad.a || test_flags(gc_input.buttons, HID::Contro
                joypad.b = joypad.b || test_flags(gc_input.buttons, HID::Contro
                joypad.select = joypad.select || test_flags(gc_input.buttons, H
                joypad.start = joypad.start || test_flags(gc_input.buttons, HID
            }
        }
    }
}
```

window->is_focused用于判断当前窗口是否具有用户焦点，即是否被用户选中。由于HID提供的是全局输入状态，我们不希望用户在操作别的窗口时处理用户的输入，因此需要首先对此进行判断。HID::get_key_state用于判断当前用户是否按下了指定的键盘按键，其在按键处于按下状态时返回true，否则返回false。我们将模拟器的方向键同时映射到键盘的WASD和上下左右键，以方便不同操作习惯的用户使用，同时将A、B、SELECT、START键分别映射为J键、K键、空格键和回车键。读者可以根据自己的喜好自行修改上述键位映射。

HID::supports_controller用于判断当前平台是否支持手柄输入，在当前平台支持手柄输入的情况下，我们可以调用HID::get_controller_state来获取手柄的当前状态，从而让我们的模拟器同样支持手柄操作。HID::get_controller_state的参数用于指定手柄的序号，该参数用于支持多手柄输入，但对于我们的模拟器来说，我们只需要从第一个手柄（序号0）获取输入就可以。HID::get_controller_state返回一个ControllerInputState类型的结构体，该结构体的原型如下：

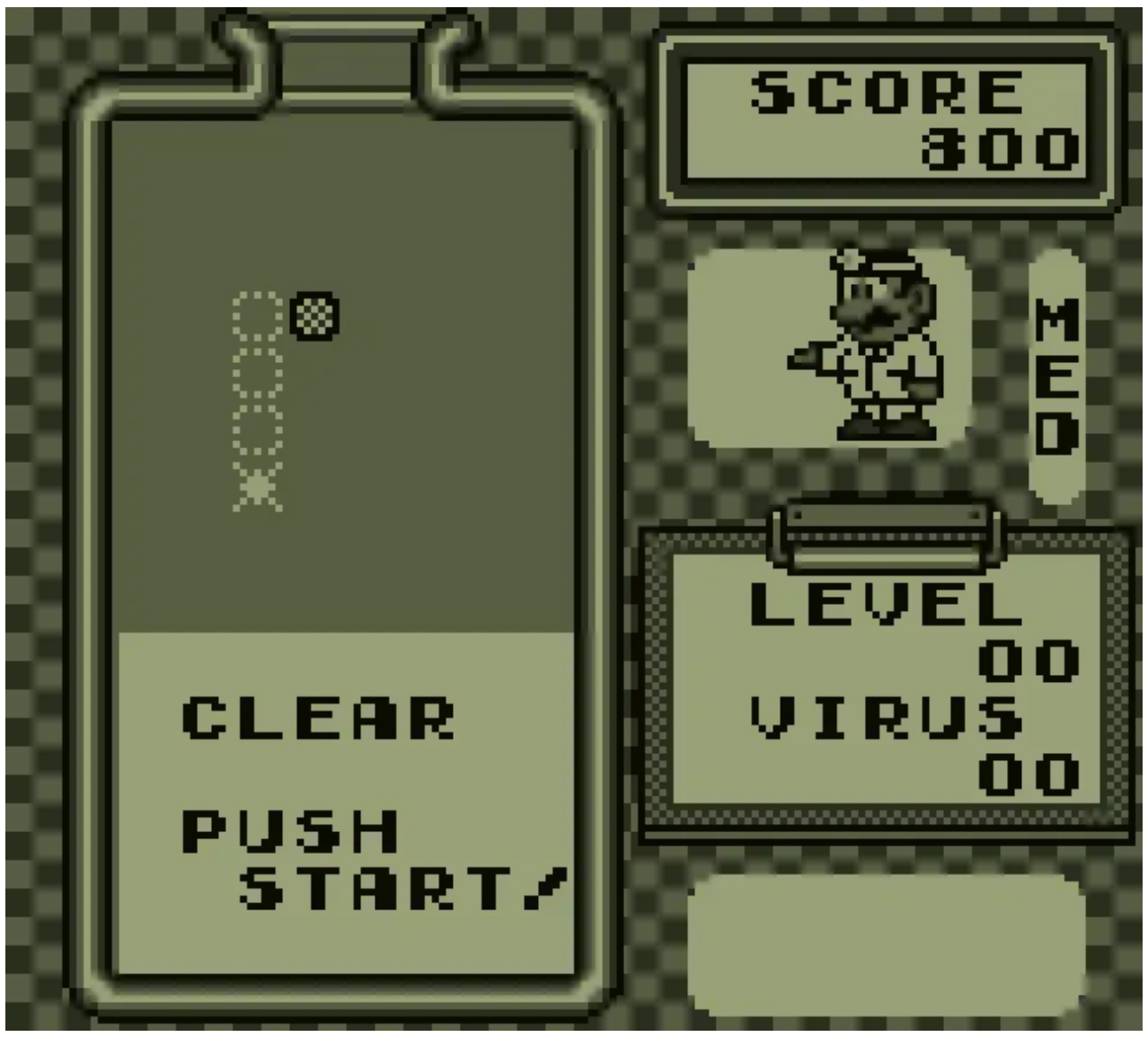
```
struct ControllerInputState
{
    //! Whether this device is connected and the state is valid.
    bool connected;
    //! A combination of bits to represent the pressed state of each button (1
    ControllerButton buttons;
    //! The x axis for left pad, mapped to [-1, 1].
    f32 axis_lx;
    //! The y axis for left pad, mapped to [-1, 1].
    f32 axis_ly;
    //! The x axis for right pad, mapped to [-1, 1].
    f32 axis_rx;
    //! The y axis for right pad, mapped to [-1, 1].
    f32 axis_ry;
    //! The left trigger value, mapped to [0, 1]. For non-linear controllers, t
    f32 axis_lt;
    //! The right trigger value, mapped to [0, 1]. For non-linear controllers,
    f32 axis_rt;
};
```

connected变量用于标识该手柄当前是否处于活跃状态，只有当connected变量的值为true时，下面的所有值才有意义。除了connected变量以外，剩下的所有变量均为从手柄读取到的用户输入，每个变量的含义在上述代码中都有解释。在使用手柄的情况下，我们会将GameBoy的方向键映射为手柄左侧的方向键和左侧的遥感，将A、B键映射为手柄的A、B键，将SELECT键、START键映射为手柄的左功能键和右功能键。

最后，我们在App::update中调用我们的update_emulator_input方法：

```
RV App::update()
{
    lutry
    {
        /*...*/
        if(emulator)
        {
            update_emulator_input();
            emulator->update(delta_time);
        }
        /*...*/
    }
    lucatchret;
    return ok;
}
```

编译并运行模拟器，打开任意单ROM的游戏卡带，可以发现我们已经能够顺利玩游戏了：



卡带RAM

到目前为止，我们的模拟器都只支持加载ROM ONLY类型的卡带，这类卡带的类型编号为0，只包括了一个ROM芯片，直接连接到卡带金手指上，用于提供游戏程序数据。在之前的章节中，我们的主要目的是尽快跑通模拟器的整个流程，而现在既然我们已经实现了模拟器的大部分功能，是时候为模拟器加上卡带RAM读写的支持了。首先我们需要在Emulator类中加入用于存储卡带RAM的变量：

```
struct Emulator
{
    byte_t* rom_data = nullptr;
    usize rom_data_size = 0;

    //! The cartridge RAM.
    byte_t* cram = nullptr;
    //! The cartridge RAM size.
    usize cram_size = 0;

    //! `true` if the emulation is paused.
    bool paused = false;
    /*...*/
};
```

我们用cram（cartilage RAM）表示卡带的RAM区域，由于每个卡带搭载的RAM尺寸并不一致，因此我们需要动态分配CRAM的内存，并使用cram_size表示卡带RAM的尺寸。接着我们在Emulator::init中加入卡带RAM的初始化代码：

```
RV Emulator::init(const void* cartridge_data, usize cartridge_data_size)
{
    /*...*/
    joypad.init();
    switch(header->ram_size)
    {
        case 2: cram_size = 8_kb; break;
        case 3: cram_size = 32_kb; break;
        case 4: cram_size = 128_kb; break;
        case 5: cram_size = 64_kb; break;
        default: break;
    }
    if(cram_size)
    {
        cram = (byte_t*)memalloc(cram_size);
        memzero(cram, cram_size);
    }
    return ok;
}
```

在第一章中，我们曾经讲过CartridgeHeader元信息结构体中每一个字段的作用，其中的ram_size字段就使用枚举的形式标记了卡带的RAM尺寸，因此我们在这里可以直接用ram_size来确定需要分配的卡带RAM的尺寸，并调用memalloc进行分配。同时，我们也别忘了在模拟器关闭时释放这部分数据：

```
void Emulator::close()
{
    if(cram)
    {
        memfree(cram);
        cram = nullptr;
        cram_size = 0;
    }
    if(rom_data)
    {
        memfree(rom_data);
        rom_data = nullptr;
        rom_data_size = 0;
        log_info("LunaGB", "Cartridge Unloaded.");
    }
}
```

接着我们需要修改卡带的总线读写函数，在卡带RAM存在的情况下，我们允许程序读写卡带RAM数据：

```
u8 cartridge_read(Emulator* emu, u16 addr)
{
    if(addr <= 0x7FFF)
    {
        return emu->rom_data[addr];
    }
    if(addr >= 0xA000 && addr <= 0xBFFF && emu->cram)
    {
        return emu->cram[addr - 0xA000];
    }
    log_error("LunaGB", "Unsupported cartridge read address: 0x%04X", (u32)addr
    return 0xFF;
}
void cartridge_write(Emulator* emu, u16 addr, u8 data)
{
    if(addr >= 0xA000 && addr <= 0xBFFF && emu->cram)
    {
        emu->cram[addr - 0xA000] = data;
        return;
    }
    log_error("LunaGB", "Unsupported cartridge write address: 0x%04X", (u32)add
}
```

可以看到，在默认情况下，我们最多只能读取8KB的卡带RAM地址范围。如果卡带RAM的尺寸超过了8KB，就需要使用下面将会介绍的MBC来切换卡带RAM的映射区域，从而让程序能够读写完整的卡带数据。

RAM存档

在有些卡带里，除了RAM芯片之外还会包含一颗纽扣电池，用于给RAM芯片供电，从而支持持久存储RAM芯片内的数据，在游戏机关闭甚至卡带被拔出以后数据都不会丢失。这种类型的RAM芯片一般用于保存玩家的游戏进度，从而支持玩家在下一次打开游戏以后接着之前的进度继续游玩，即现代游戏普遍使用的游戏存档功能。游戏存档技术的出现显著提高了游戏的深度上限，使得制作长剧情的沉浸式游戏成为了可能。为了在模拟器中支持类似的功能，我们可以在每次卸载卡带的时候将卡带RAM中的数据以二进制的形式保存成一个文件，并在下次加载同一个卡带时读取该文件，将数据拷贝至卡带RAM内存中，从而让游戏“觉得”数据一直保存在卡带的RAM内存里。我们首先在Cartridge.hpp中添加一个帮助函数：

```
inline bool is_cart_battery(u8 cartridge_type)
{
    return cartridge_type == 3 || // MBC1+RAM+BATTERY
    cartridge_type == 6 || // MBC2+BATTERY
    cartridge_type == 9 || // ROM+RAM+BATTERY 1
    cartridge_type == 13 || // MMM01+RAM+BATTERY
    cartridge_type == 15 || // MBC3+TIMER+BATTERY
    cartridge_type == 16 || // MBC3+TIMER+RAM+BATTERY 2
    cartridge_type == 19 || // MBC3+RAM+BATTERY 2
    cartridge_type == 27 || // MBC5+RAM+BATTERY
    cartridge_type == 30 || // MBC5+RUMBLE+RAM+BATTERY
    cartridge_type == 34; // MBC7+SENSOR+RUMBLE+RAM+BATTERY
}
```

该函数用于判断我们的游戏卡带是否为具有电池，从而确定我们是否需要启用卡带RAM数据存储功能。然后为Emulator类添加两个函数来分别对应卡带RAM内存数据的保存和加载功能：

```
struct Emulator
{
    /*...*/
    u8 bus_read(u16 addr);
    void bus_write(u16 addr, u8 data);
    void load_cartridge_ram_data();
    void save_cartridge_ram_data();
};
```

在Emulator初始化并创建的时候，我们需要尝试加载RAM卡带的数据，而在Emulator销毁的时候，我们需要保存卡带数据，代码如下：

```
RV Emulator::init(const void* cartridge_data, usize cartridge_data_size)
{
    /*...*/
    if(cram_size)
    {
        cram = (byte_t*)memalloc(cram_size);
        memzero(cram, cram_size);
        if(is_cart_battery(header->cartridge_type))
        {
            load_cartridge_ram_data();
        }
    }
    return ok;
}
void Emulator::close()
{
    if(cram)
    {
        CartridgeHeader* header = get_cartridge_header(rom_data);
        if(is_cart_battery(header->cartridge_type))
        {
            save_cartridge_ram_data();
        }
        memfree(cram);
        cram = nullptr;
        cram_size = 0;
    }
    /*...*/
}
```

由于模拟器需要知道当前我们正在运行哪一个卡带，才能够正确读写对应的存档文件，因此我们在模拟器中添加一个新的变量来存储卡带的文件路径，然后修改Emulator::init函数，将卡带的文件路径作为参数传入。首先在Emulator.hpp中添加新的头文件：

```
#include <Luna/Runtime/Path.hpp>
```

然后修改Emulator类，添加对应的路径变量：

```
struct Emulator
{
    //! The cartridge file path. Used for saving cartridge RAM data if any.
    Path cartridge_path;

    byte_t* rom_data = nullptr;
    /*...*/

    RV init(Path cartridge_path, const void* cartridge_data, usize cartridge_da
    /*...*/
};
```

并在Emulator::init中设置：

```
RV Emulator::init(Path cartridge_path, const void* cartridge_data, usize cartri
{
    luassert(cartridge_data && cartridge_data_size);
    this->cartridge_path = cartridge_path;
    rom_data = (byte_t*)memalloc(cartridge_data_size);
    /*...*/
}
```

最后修改App::open_cartridge，将卡带的路径传入对模拟器的init函数的调用中：

```
void App::open_cartridge()
{
    lutry
    {
        /*...*/
        if(succeeded(result) && !result.get().empty())
        {
            /*...*/
            UniquePtr<Emulator> emu(memnew<Emulator>());
            luexp(emu->init(path, rom_data.data(), rom_data.size()));
            emulator = move(emu);
        }
    }
    /*...*/
}
```

最后我们需要在Emulator.cpp中实现卡带RAM数据的加载和保存函数：

```
void Emulator::load_cartridge_ram_data()
{
    lutry
    {
        auto path = cartridge_path;
        if (path.empty()) return;
        path.replace_extension("sav");
        lulet(f, open_file(path.encode().c_str(), FileOpenFlag::read, FileCreat
        luexp(f->read(cram, cram_size));
        log_info("LunaGB", "cartridge RAM data loaded: %s", path.encode().c_str
    }
    lucatch {}
    return;
}

void Emulator::save_cartridge_ram_data()
{
    lutry
    {
        if (cartridge_path.empty() || !cram) return;
        Path path = cartridge_path;
        path.replace_extension("sav");
        lulet(f, open_file(path.encode().c_str(), FileOpenFlag::write, FileCrea
        luexp(f->write(cram, cram_size));
        log_info("LunaGB", "Save cartridge RAM data to %s.", path.encode().c_st
    }
    lucatch
    {
        log_error("LunaGB", "Failed to save cartridge RAM data. Game progress n
    }
}
```

可以看到对于需要存储RAM数据的游戏，我们会在游戏卡带ROM文件的同一目录下创建一个与游戏卡带文件名相同，但是以.sav后缀结尾的文件，然后直接将卡带RAM中的二进制数据写入文件。而在加载卡带时，我们会查找该文件，并在文件存在时加载文件中的数据至卡带RAM对应的内存中。如果加载卡带时我们没有找到对应的文件，则我们直接跳过加载卡带RAM数据的步骤，此时卡带RAM会以数据全部是0开始运行。

MBC1

MBC为Multiple Bank Controller的缩写，搭载了MBC的卡带允许程序在运行时将卡带ROM和RAM的不同分区映射到GameBoy的地址总线上，供程序读写。在第一章中，我们已经详细讲解了MBC卡带的硬件构造，读者可以重新阅读相关内容进行温习。在本章中，我们将以MBC1卡带为例，讲解MBC卡带的工作原理，以及我们如何编写代码模拟MBC卡带的工作。至于MBC2和MBC3卡带，我们将放到下一章中介绍和实现。

MBC卡带与普通卡带的最大差别在于其搭载了MBC（Multiple Bank Controller）芯片，其一端连接卡带的金手指，另一端连接卡带内置的ROM和RAM芯片，从而充当卡带数据读写的中间商角色。MBC的功能类似于一个多路开关，其可以受程序控制，将ROM芯片和RAM芯片的不同区域映射到GameBoy的卡带ROM和卡带RAM总线地址区域内，从而让游戏读写其中的数据。GameBoy总共具有MBC1、MBC2、MBC3、MBC5、MBC6、MBC7这六种MBC卡带类型，不同的MBC卡带类型在最高支持的ROM/RAM尺寸、映射规则、控制协议等方面有所不同，但是其工作原理是一样的，都是通过动态将总线地址区域映射到存储芯片的不同区域来允许程序读写更大范围的数据。单色GameBoy的卡带以MBC1~MBC3居多，而彩色GameBoy的卡带则囊括了从MBC1到MBC7的所有MBC卡带类型。由于本专题只涉及单色GameBoy的开发，因此在本专题中，我们将只实现MBC1~MBC3类型的卡带读写功能，而其余卡带类型，读者可以参考GameBoy的pandocs文档自行扩展实现。

MBC1是最早推出的MBC芯片，其具有5+2个分块选择引脚，并支持两种连接配置：

1. 将5个引脚用于选择ROM分块，2个引脚用于选择RAM分块，因此支持最多32个ROM分块和4个RAM分块，即512KB的ROM和32KB的RAM。
2. 将7个引脚全部用于选择ROM分块，因此支持最多128个ROM分块，即2MB的ROM。在这种配置下，RAM将无法通过MBC切换分块，因此最高只支持8KB的RAM。

> 在所有的MBC方案中，单个ROM分块的尺寸均为16KB，单个RAM分块的尺寸均为8KB。

第一种连接配置为MBC1的出厂设计配置，而第二种配置实际上是游戏厂商为了支持大尺寸ROM而自己魔改的配置。在下文中，为了区分这两种卡带，我们将第一张配置称为标准卡带，第二章配置称为2MB卡带。

在映射分块时，MBC1将卡带的总线地址分为三个区域：

1. 0x0000~0x3FFF：ROM0区域（16KB）
2. 0x4000~0x7FFF：ROM1区域（16KB）
3. 0xA000~0xBFFF：RAM区域（8KB）

每一个地址区域都可以被完整映射到一个ROM或者RAM分块上，但是ROM0和ROM1地址区域只能被映射到ROM分块上，RAM地址区域只能被映射到RAM分块上。MBC1具有两个工作模式：默认（default）模式和高级（advanced）模式，下表展示了两个模式下每一个地址区域可以映射的分块范围：

MBC工作模式	分块序号范围（默认模式）	分块序号范围（高级模式）
ROM0区域（标准卡带）	0x00	0x00
ROM1区域（标准卡带）	0x01~0x1F	0x01~0x1F
RAM区域（标准卡带）	0x00	0x00~0x03
ROM0区域（2MB卡带）	0x00	0x00, 0x20, 0x40, 0x60
ROM1区域（2MB卡带）	0x01~0x1F	0x01~0x1F, 0x21~0x3F, 0x41~0x5F, 0x61~0x7F
RAM区域（2MB卡带）	0x00	0x00

可以看到，在默认模式下，MBC1只有5个引脚用于控制ROM1区域的内存块切换，ROM0区域和RAM区域均固定映射到第一个ROM和RAM分块上；在高级模式下，MBC1的剩下2个引脚也可以用于选择RAM分块（标准卡带），或者作为ROM分块索引的高2位选择ROM分块（2MB卡带）。

在没有MBC的卡带中，卡带的ROM区域（0x0000~0x7FFF）是只读的，游戏程序不应该向这些地址写入数据；但是在MBC卡带中，由于程序向卡带写入数据会首先经过MBC，因此MBC利用了这一地址区间作为向MBC发送控制指令的地址区间。具体来说，在MBC卡带中，当程序向0x0000~0x7FFF区域写入数据时，MBC控制器并不是真的将这些数据写入RAM，而是根据程序指定的地址和写入的值，执行相应的MBC操作。下表列出了不同操作所对应的地址区间：

地址区间	操作	默认值
0x0000~0x1FFF	启用/禁用RAM	禁用

0x2000~0x3FFF	设置ROM分块序号	1
0x4000~0x5FFF	设置RAM分块序号	0
0x6000~0x7FFF	设置工作模式	默认

启用/禁用RAM

该操作用于启用/禁用卡带RAM，对该地址写入任意低四位等于0xA的数据会启用RAM，写入除此之外的任意值则会禁用RAM。当卡带RAM被禁用时，程序对卡带RAM的所有写操作会被丢弃，所有读操作会返回0xFF。

根据GameBoy开发手册，程序只应在确实需要读写RAM数据（例如保存或者加载存档时）才启用RAM，并应在读写完毕以后立即禁用RAM，以避免由于卡带连接不稳定、意外关机或者拔出卡带导致的数据损坏。

设置ROM分块序号

该操作用于选择ROM1的映射分块，写入的值将作为分块序号使用。分块序号的范围是0x00~0x1F，因此只有写入值的低五位会作为分块序号使用，而高三位的值会被忽略。在2MB卡带中，该操作用于指定分块序号的低5位，而高2位则通过设置RAM分块序号操作来指定。

在设置分块序号时，如果实际的ROM芯片并没有32个分块（即尺寸不到512KB），则当设置的分块序号大于ROM芯片实际拥有的分块数时，MBC会逐步丢弃序号的高位数据，直到剩下的数据在ROM芯片的有效序号范围内。例如，如果ROM芯片的实际尺寸是256KB，则只有写入值的低四位会被使用，因此分块序号的有效范围是0x00~0x0F。

需要注意的是，该操作无法将ROM芯片的第一个分块（bank 0）映射到ROM1上，当程序写入0x00时，MBC会将其等同于写入0x01处理，并将ROM芯片的第二个分块（bank 1）映射到ROM1上。

设置RAM分块序号

在标准卡带上，该操作用于设置RAM的映射分块，写入的值将作为分块序号使用。分块序号的范围是0x00~0x03，因此只有写入值的低2位会作为分块序号使用。与ROM分块序号类似，当RAM芯片的实际分块数量小于4时，MBC会逐步丢弃序号的高位数据，直到剩下的数据在RAM芯片的有效序号范围内。

在2MB卡带上，该操作用于设置ROM1的映射分块的高2位，其同时也会改变ROM0的映射分块，规则如下：

1. 写入值为0x00时，ROM0映射到0x00分块，ROM1映射到0x01~0x1F分块。
2. 写入值为0x01时，ROM0映射到0x20分块，ROM1映射到0x21~0x3F分块。
3. 写入值为0x02时，ROM0映射到0x40分块，ROM1映射到0x41~0x5F分块。
4. 写入值为0x03时，ROM0映射到0x60分块，ROM1映射到0x61~0x7F分块。

需要注意的是，RAM分块序号只有当MBC工作模式为高级时才生效。当MBC工作模式为默认时，设置RAM分块序号操作会修改对应内部寄存器的值，但是不会使该值真正用于控制分块映射，而是需要在之后设置MBC工作模式为高级时才生效。

设置工作模式

该操作用于设置MBC的工作模式，当写入值的最低位为0时，MBC的工作模式将设置为默认，否则设置为高级。

实现MBC1

在了解了MBC1的工作原理和控制方法后，是时候编写代码实现MBC1卡带了！首先我们在Cartridge.hpp中添加用于检测当前卡带是否为MBC1卡带的辅助函数：

```
inline bool is_cart_mbc1(u8 cartridge_type)
{
    return cartridge_type >= 1 && cartridge_type <= 3;
}
```

然后修改cartridge_read和cartridge_write函数，加入MBC1卡带支持：

```
u8 cartridge_read(Emulator* emu, u16 addr)
{
    u8 cartridge_type = get_cartridge_header(emu->rom_data)->cartridge_type;
    if(is_cart_mbc1(cartridge_type))
    {
        return mbc1_read(emu, addr);
    }
    else
    {
```

```
        if(addr <= 0x7FFF)
        {
            return emu->rom_data[addr];
        }
        if(addr >= 0xA000 && addr <= 0xBFFF && emu->cram)
        {
            return emu->cram[addr - 0xA000];
        }
    }
    log_error("LunaGB", "Unsupported cartridge read address: 0x%04X", (u32)addr
    return 0xFF;
}
void cartridge_write(Emulator* emu, u16 addr, u8 data)
{
    u8 cartridge_type = get_cartridge_header(emu->rom_data)->cartridge_type;
    if(is_cart_mbc1(cartridge_type))
    {
        mbc1_write(emu, addr, data);
        return;
    }
    else
    {
        if(addr >= 0xA000 && addr <= 0xBFFF && emu->cram)
        {
            emu->cram[addr - 0xA000] = data;
            return;
        }
    }
    log_error("LunaGB", "Unsupported cartridge write address: 0x%04X", (u32)add
}
```

可以看到，我们在cartridge_read和cartridge_write中首先对当前的卡带类型进行了判断，如果卡带为MBC1类型的卡带，我们就调用mbc1_read和mbc1_write函数（将在下文中实现）执行实际的读写操作，否则才使用之前非MBC卡带的逻辑读写数据。

接着在Emulator中加入以下变量来支持MBC1卡带：

```
struct Emulator
{
    /*...*/
    //! The cartridge RAM size.
    usize cram_size = 0;

    //! The number of ROM banks. 16KB per bank.
    usize num_rom_banks = 0;
    //! MBC1: The cartridge RAM is enabled for reading / writing.
    bool cram_enable = false;
    //! MBC1: The ROM bank number controlling which rom bank is mapped to 0x400
    u8 rom_bank_number = 1;
    //! MBC1: The RAM bank number register controlling which ram bank is mapped
    //! If the cartridge ROM size is larger than 512KB (32 banks), this is used
    //! high 2 bits of rom bank number, enabling the game to use at most 2MB of
    u8 ram_bank_number = 0;
    //! MBC1: The banking mode.
    //! 0: 0000–3FFF and A000–BFFF are locked to bank 0 of ROM and SRAM respect
    //! 1: 0000–3FFF and A000–BFFF can be bank-switched via the 4000–5FFF regis
    u8 banking_mode = 0;

    //! `true` if the emulation is paused.
    bool paused = false;
    /*...*/
};
```

其中：

- num_rom_banks用于记录当前的卡带有多少个ROM分块，我们用该变量来判断当前卡带是否为尺寸超过512KB的卡带，从而确定MBC1的工作模式。
- cram_enable用于记录卡带RAM当前是否被启用。
- rom_bank_number用于记录ROM的卡带分块序号的低5位。
- ram_bank_number用于记录RAM的卡带分块序号，或者ROM的卡带分块序号的高2位。
- banking_mode用于记录MBC1的工作模式，0为默认模式，1为高级模式。

然后我们需要在Emulator::init中初始化num_rom_banks的值：

```
RV Emulator::init(Path cartridge_path, const void* cartridge_data, usize cartri
```

```
{
    /*...*/
    if(checksum != header->checksum)
    {
        return set_error(BasicError::bad_data(), "The cartridge checksum dismat
    }
    num_rom_banks = (((usize)32) << header->rom_size) / 16;
    // Print cartridge load info.
    /*...*/
}
```

有了这些变量以后，我们就可以实现mbc1_read和mbc1_write函数了。我们首先来实现mbc1_read函数，在Cartridge.cpp中实现如下：

```
u8 mbc1_read(Emulator* emu, u16 addr)
{
    if(addr <= 0x3FFF)
    {
        if(emu->banking_mode && emu->num_rom_banks > 32)
        {
            usize bank_index = emu->ram_bank_number;
            usize bank_offset = bank_index * 32 * 16_kb;
            return emu->rom_data[bank_offset + addr];
        }
        else
        {
            return emu->rom_data[addr];
        }
    }
    if(addr >= 0x4000 && addr <= 0x7FFF)
    {
        // Cartridge ROM bank 01-7F.
        if(emu->banking_mode && emu->num_rom_banks > 32)
        {
            usize bank_index = emu->rom_bank_number + (emu->ram_bank_number <<
            usize bank_offset = bank_index * 16_kb;
            return emu->rom_data[bank_offset + (addr - 0x4000)];
        }
        else
        {
            usize bank_index = emu->rom_bank_number;
            usize bank_offset = bank_index * 16_kb;
            return emu->rom_data[bank_offset + (addr - 0x4000)];
        }
    }
    if(addr >= 0xA000 && addr <= 0xBFFF)
    {
        if(emu->cram)
        {
            if(!emu->cram_enable) return 0xFF;
            if(emu->num_rom_banks <= 32)
            {
                if(emu->banking_mode)
                {
                    // Advanced banking mode.
                    usize bank_offset = emu->ram_bank_number * 8_kb;
                    luassert(bank_offset + (addr - 0xA000) <= emu->cram_size);
                    return emu->cram[bank_offset + (addr - 0xA000)];
                }
                else
                {
                    // Simple banking mode.
                    return emu->cram[addr - 0xA000];
                }
            }
            else
            {
                // ram_bank_number is used for switching ROM banks, use 1 ram p
                return emu->cram[addr - 0xA000];
            }
        }
    }
    log_error("LunaGB", "Unsupported MBC1 cartridge read address: 0x%04X", (u32
    return 0xFF;
}
```

mbc1_read首先判断程序需要读取数据的地址范围，分为ROM0、ROM1和RAM三个部分，并根据当前的MBC1寄存器状态，将ROM和RAM内存地址加上合适的偏移值后返回读取的结果。虽然代码的部分较长，但是都是上述MBC1特性说明的具体实现，读者结合上文的描述应当不难看懂这部分代码。

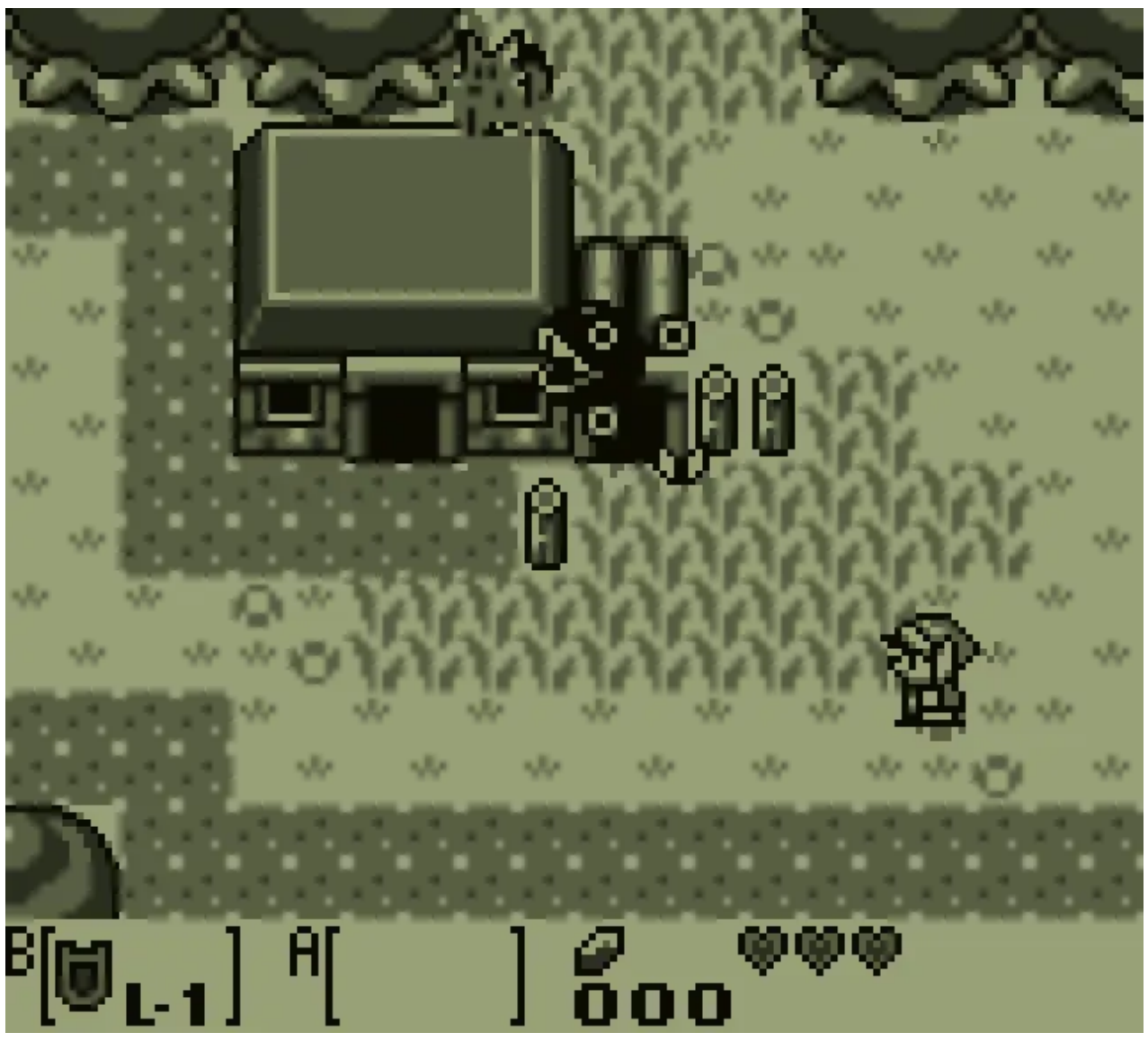
mbc1_write函数实现如下：

```
void mbc1_write(Emulator* emu, u16 addr, u8 data)
{
    if(addr <= 0x1FFF)
    {
        // Enable/disable cartridge RAM.
        if(emu->cram)
        {
            if((data & 0x0F) == 0x0A)
            {
                emu->cram_enable = true;
            }
            else
            {
                emu->cram_enable = false;
            }
            return;
        }
    }
    if(addr >= 0x2000 && addr <= 0x3FFF)
    {
        // Set ROM bank number.
        emu->rom_bank_number = data & 0x1F;
        if(emu->rom_bank_number == 0)
        {
            emu->rom_bank_number = 1;
        }
        if(emu->num_rom_banks <= 2)
        {
            emu->rom_bank_number = emu->rom_bank_number & 0x01;
        }
        else if(emu->num_rom_banks <= 4)
        {
            emu->rom_bank_number = emu->rom_bank_number & 0x03;
        }
        else if(emu->num_rom_banks <= 8)
        {
            emu->rom_bank_number = emu->rom_bank_number & 0x07;
        }
        else if(emu->num_rom_banks <= 16)
        {
            emu->rom_bank_number = emu->rom_bank_number & 0x0F;
        }
        return;
    }
    if(addr >= 0x4000 && addr <= 0x5FFF)
    {
        // Set RAM bank number.
        emu->ram_bank_number = data & 0x03;
        // Discards unsupported banks.
        if(emu->num_rom_banks > 32)
        {
            if(emu->num_rom_banks <= 64)
            {
                emu->ram_bank_number &= 0x01;
            }
        }
        else
        {
            if(emu->cram_size <= 8_kb)
            {
                emu->ram_bank_number = 0;
            }
            else if(emu->cram_size <= 16_kb)
            {
                emu->ram_bank_number &= 0x01;
            }
        }
        return;
    }
    if(addr >= 0x6000 && addr <= 0x7FFF)
    {
```

```
        // Set banking mode.
        if(emu->num_rom_banks > 32 || emu->cram_size > 8_kb)
        {
            emu->banking_mode = data & 0x01;
        }
        return;
    }
    if(addr >= 0xA000 && addr <= 0xBFFF)
    {
        if(emu->cram)
        {
            if(!emu->cram_enable) return;
            if(emu->num_rom_banks <= 32)
            {
                if(emu->banking_mode)
                {
                    // Advanced banking mode.
                    usize bank_offset = emu->ram_bank_number * 8_kb;
                    luassert(bank_offset + (addr - 0xA000) <= emu->cram_size);
                    emu->cram[bank_offset + (addr - 0xA000)] = data;
                }
                else
                {
                    // Simple banking mode.
                    emu->cram[addr - 0xA000] = data;
                }
            }
            else
            {
                // ram_bank_number is used for switching ROM banks, use 1 ram
                emu->cram[addr - 0xA000] = data;
            }
            return;
        }
    }
    log_error("LunaGB", "Unsupported MBC1 cartridge write address: 0x%04X", (u
}
```

mbc1_write首先判断程序需要写入数据的地址范围，当范围在0x0000~0x7FFF时，其根据每一个范围对应的功能，将程序需要写入的值保存到各个寄存器变量中，而当范围在0xA000~0xBFFF时，其根据当前的banking_mode和ram_bank_number将数据写入卡带RAM的不同分块中。

此时编译并运行模拟器，打开一个MBC1类型的卡带，例如《塞尔达传说：织梦岛》，可以看到我们已经能够正常游玩该游戏了：



以上就是本章节的全部内容了。在下一章中，我们将接着这一章的内容，介绍和实现MBC2和MBC3两种MBC卡带的读写。那么，下一章见！

銀葉吉祥：从零开始实现GameBoy模拟器 #11 MBC2、MBC3卡带
14 赞同 · 0 评论 文章

编辑于 2024-03-03 16:29 · IP 属地上海

Game Boy（GB） 游戏机

欢迎参与讨论

还没有评论，发表第一个评论吧

文章被以下专栏收录

吉祥的游戏制作笔记
游戏制作中的技术、艺术和设计灵感记录

推荐阅读

开始Chiptune之旅，为LSDJ准备模拟器
蓬岸 Dr... 发表于古董电脑室

为什么盗版FC卡带上总有一个黑疙瘩?
杉果Son... 发表于杉果游戏

【掌机玩家必看】Steam、ROG掌机怎么玩才够爽？一文
绯闻男友

梦开始的地方——FC游戏开发指南（5）导入并显示图片
皮皮关 发表于游戏开发入...

▲ 赞同 13 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···