从零开始实现GameBoy模拟器 #2 CPU、时钟和总线

銀葉吉祥
浙江大学 软件工程硕士

目录 收起

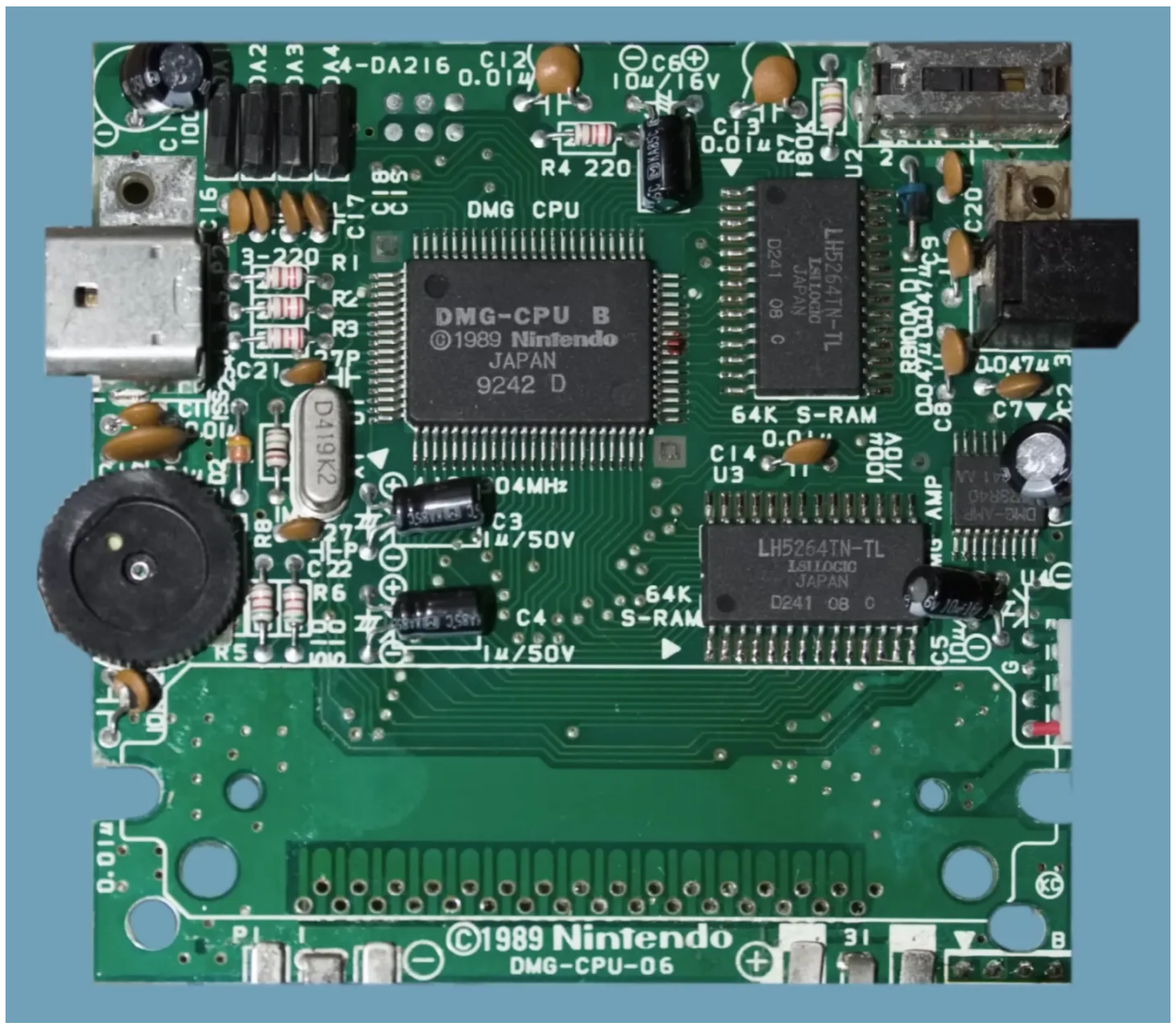
> 2024.1.28更新：修改了时钟的更新方式，由每帧直接更新时钟变为了通过步进CPU来间接更新时钟。新版代码可以更方便地实现正确的总线读写时序，并且不再需要CPU的cycles_countdown来计算指令的时间开销，实现更加简洁。
> 2024.2.4更新：修复一个代码bug：忘了在Emulator中添加high RAM存储空间，已修复。

欢迎来到从零开始实现GameBoy模拟器第二章。从本章开始，我们将开始实现GameBoy的核心元件——CPU。我们首先会大概介绍GameBoy的硬件原理（作者非嵌入式及硬件专业学生，因此只能大概介绍），然后我们将通过编写代码实现GameBoy的时钟、总线和CPU的总体框架，而对于CPU指令的具体实现，我们将放在之后几章中逐一介绍。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-02，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：338547343

GameBoy硬件介绍

初代GameBoy的核心主板如下图所示：

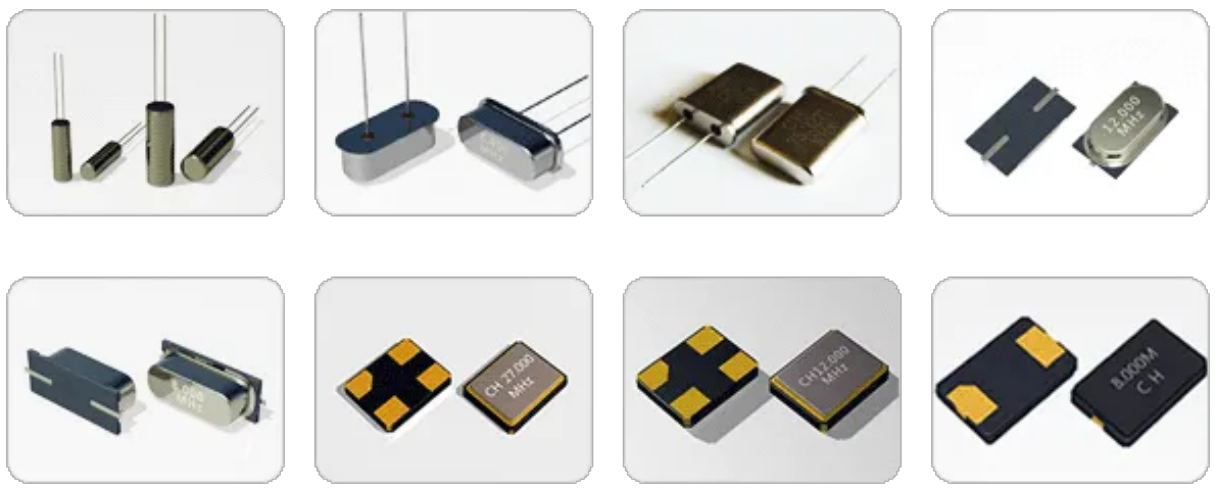


出处：https://www.youtube.com/watch?v=RZUDEaLa5Nw&t=10s

位于主板中央的是一枚印有DMG-CPU B的芯片，该芯片是任天堂为GameBoy定制的SoC（System on Chip），其内置了一个基于ZiLOG Z80架构的CPU、一个用于输出图形的像素处理单元（PPU）、一个用于产生声音的音频处理单元（APU），以及一些额外的辅助电路单元。如果您曾经学过单片机、嵌入式相关的知识，对于这样一个类似于单片机的芯片应该非常熟悉。位于SoC右侧的两枚标有LH5264TN-TL的芯片为GameBoy的内存芯片，两枚芯片均为8KB的SRAM存储器，分别提供了GameBoy的工作内存（WRAM）和显示内存（VRAM）。位于主板下方的是用来连接卡带的32pin金手指，卡带的ROM、RAM芯片通过金手指直接连接到SoC上，以供CPU读取卡带中的数据。主板右侧的黑色接口用于在没有电池的时候为GameBoy提供DC供电，而主板左侧的接口是串口通信接口，在那个互联网还没有普及的年代，这个串口是GameBoy与别的GameBoy通信的唯一渠道。

时钟

时钟是所有现代计算机工作的基础，哪怕读者不了解时钟，也一定在购买CPU和内存的时候听过类似“主频”、“超频”等术语，这里的“频”指的就是计算机的时钟频率。计算机的时钟是一个能够产生规律性脉冲信号的硬件，最常用的时钟硬件被称为“晶体振荡器”或者“晶振”，它的长相类似这样：



时钟硬件直接与SoC的时钟引脚相连，在每一次时钟产生脉冲信号时，SoC的内部状态就会被刷新一次，即完成一次计算或者操作。显然，时钟的频率越高，SoC刷新的速度也会越快，也就代表着相同时间中，SoC能够处理更多的指令，但同时也会导致整机的功耗增加。现代的大部分计算机都具有可变时钟的功能，其通过将一组不同频率的晶振自由组合来根据工作负载来自由调整频率，以便在性能和功耗上达到平衡。然而GameBoy只有一个固定频率的时钟，其安装在上图SoC的左侧，标号为D419K2，频率为4194304Hz，也就是说其能够在一秒钟之内产生4194304次脉冲信号。SoC中的所有电路都由这个晶振驱动，因此它们的时序完全同步，这对我们实现模拟器有很大的帮助。

虽然GameBoy的硬件时钟是固定频率的，但是在模拟GameBoy时，我们实际上可以自由设置模拟器的刷新频率。将刷新频率设置为低于或者高于默认值会使得游戏的模拟速度改变。在实现CPU时，我们可以将刷新频率设置为一个非常低的值，这样可以让我们更容易对我们的CPU进行调试。

时钟周期和机器周期

在之后的开发中，我们将会频繁使用到时钟周期和机器周期这两个概念，因此有必要在此对这两个名词进行说明。

时钟周期（clock cycles，也可以直接称为cycles）指的是两个相邻的时钟脉冲信号之间的时间长度。由于GameBoy的时钟频率为4194304Hz，因此一个时钟周期的长度即为1/4194304秒。在后面描述各个硬件电路行为时，我们经常会说“XX操作需要花费N个时钟周期”，即该操作消耗的时间为N/4194304秒。

机器周期（machine cycles，简称为Mcycles）指的是两次相邻的CPU更新之间的时间长度。GameBoy的CPU每四个时钟周期更新一次，因此**一个机器周期的长度相当于四个时钟周期**，即1/1048576秒。GameBoy所有的CPU指令的时间开销均为机器周期的整数倍，因此在描述CPU指令的时间开销时，我们通常会以机器周期而非时钟周期进行描述。

总线

除了CPU以外，GameBoy有众多的周边硬件，例如工作内存、显示内存、中断和计时器、PPU、APU等。为了让程序更加方便地与这些周边硬件交互，GameBoy使用总线（bus）封装了所有对这些硬件设备的交互操作，所有的操作最终都化为两个操作：从总线读数据和向总线写入数据。

总线的读写操作与我们平时编程时对内存的读写操作类似：每一个总线操作都有一个需要操作的地址，读操作会返回从地址中读取的数据，而写操作则会将数据写入指定的总线地址中。但是与内存读写不同，总线的某些地址并不指向真正的内存，而是指向一些特殊的硬件寄存器，而向这些寄存器写入数据就可以激活硬件去执行特定的工作，而读取这些地址的数据则会返回这些硬件的内部状态。比方说，0xFF00地址映射的是GameBoy的输入按钮（GamePad），当程序向0xFF00的第五位写入1，第六位写入0以后，程序便可以读取0xFF00的数据，其低四位表示了按钮A、B、select、start的按下状态；反之，当程序向0xFF00的第五位写入0，第六位写入1以后，读到的数据的低四位则代表了四个方向键（右、左、上、下）的按下状态。在我们之后的模拟器开发中，有大量类似的使用总线读写来控制硬件行为的地方，但是在实现CPU时，我们只需要使用统一的指令来读写总线，就可以将这些硬件行为封装起来，大大简化了我们的编程难度。

GameBoy的CPU具有16位总线的寻址能力，意味着其最多可以访问0x0000至0xFFFF，一共65536个总线地址。在GameBoy的官方文档中，其对于总线地址每一个区域的用途有详细的说明，以下做简要叙述：

1. 0x0000~0x7FFF映射到卡带的ROM内存区域，一共32KB。对于支持MBC的卡带，后16KB的地址区域（0x4000~0x7FFF）可以按需映射到不同的ROM内存区域上，以读取其中的数据。
2. 0x8000~0x9FFF映射到显示内存（VRAM），一共8KB。
3. 0xA000~0xBFFF映射到卡带的RAM内存区域（CRAM），一共8KB。对于支持MBC+RAM的卡带，该区域可以按需映射到不同的RAM内存区域上，以读写其中的数据。
4. 0xC000~0xDFFF映射到工作内存（WRAM），一共8KB。
5. 0xE000~0xFDFF为禁止使用区域。根据实际硬件检测，其映射到了与0xC000~0xDDFF相同的内存区域，因此有些文档中称其为ECHO RAM。
6. 0xFE00~0xFE9F为对象属性内存（Object Attribute Memory，OAM），其保存了画面上的精灵对象的属性。我们会在实现PPU时详细讲解该内存。
7. 0xFEA0~0xFEFF为另一禁止使用区域。
8. 0xFF00~0xFF7F为I/O寄存器区域，其映射到了大部分GameBoy的外设硬件控制寄存器上，例如PPU的控制寄存器、APU的控制寄存器、按钮输入、串口通信等。
9. 0xFF80~0xFFFE是一块被称为高位RAM（High RAM、HRAM）的特殊内存区域，其直接集成在SoC中，程序可以像使用工作RAM一样自由使用这块区域。在实际游戏中，这块区域通常用于存储程序的函数调用栈。
10. 0xFFFF映射到中断使能寄存器，其控制GameBoy的各种中断是否处于激活状态。

对于以上每一个区域，我们会在实际编写对应硬件的模拟代码时详细叙述，此处可以仅了解就行。所有对非法总线地址（访问被拒绝的区域，或者不存在实际硬件的区域，例如访问不带有RAM的卡带的CRAM地址区域）的读操作均会返回0xFF，而所有对非法总线地址的写操作会被直接忽略。

> 访问非法地址会返回0xFF的原因是GameBoy SoC的非连通引脚始终具有高电平，所以当读取按钮的输入状态时，1表示的是未按下，而0表示的是按下，因为按下按钮会使得对应的引脚连通接地。同样的道理，当我们不插入卡带就启动GameBoy时，画面上原本出现任天堂LOGO的地方会出现一块黑色的矩形，就是因为GameBoy在读取卡带里的任天堂LOGO数据时读取到了全是0xFF的值。

来源：https://youtube.com/shorts/oEJ3W77atFA

CPU

大部分的中央处理单元（CPU）都遵循类似的架构设计，因此此处的CPU原理介绍也可以引用在分析别的CPU的核心结构上。

寄存器

寄存器（register）是内嵌在CPU中的数据存储单元，用于存储CPU在运行过程中的临时数据。GameBoy CPU的寄存器包括A、F、B、C、D、E、H、L，一共8个8位寄存器，以及PC和SP两个16位寄存器。同时，在某些指令中，8个8位寄存器可以当成4个16位寄存器（AF、BC、DE、HL）使用。在8位寄存器中，F（flags）寄存器主要用于存储CPU指令产生的各种标志位（例如进位标志位、减法标志位等），不能直接写入数据，而其余的寄存器都可以通过LD指令直接写入数据。

并不是所有的寄存器都具有相同的功能。某些寄存器具有特殊的含义，例如：

1. A寄存器被定义为累加器（accumulator），几乎所有的数学计算都只能针对A寄存器里的数据进行。
2. H（high）和L（low）寄存器用于存储一个16位地址的高8位和低8位，这两个寄存器用于在总线读写指令中提供地址，而别的大部分寄存器不能直接存储地址用于读写。
3. PC（program counter）寄存器存储CPU下一条要执行的指令的总线地址，CPU使用该地址从内存中读取指令，并会在执行指令时自动增加该寄存器的值。该寄存器的值无法直接修改，需要使用JP、JR等专门的跳转指令进行跳转。
4. SP（stack pointer）寄存器存储当前的函数调用栈的栈顶的内存地址，并会在call、ret指令或者触发中断时被CPU自动操作。

F（flags）寄存器用于存储CPU在运行过程中产生的各种位，其只有高位的4个比特有效，低位的4个比特永远是0。高位的四个比特对应的标志位分别为：

1. 比特7：Z标志位（zero flag），当运算结果为0时设置为1，否则设置为0。
2. 比特6：N标志位（negative flag），当运算为减法时设置为1，否则设置为0。
3. 比特5：H标志位（half-carry flag），当运算出现了比特3与比特4之间的进位或退位时设置为1，否则设置为0。
4. 比特4：C标志位（carry flag），当运算出现了向上或者向下溢出时设置为1，否则设置为0。

上述比特均从最低位为0开始计数。

执行流程

如上文所述，GameBoy的CPU每四个时钟周期（或者一个机器周期）更新一次。为了执行一条指令，CPU需要完成两个步骤：

1. 获取指令
2. 执行指令

在获取指令阶段，CPU会读取PC寄存器的值，并通过总线读取指令获取该地址下存储的操作码数据，最后将PC寄存器的值加1并写回。所有对于总线的读写操作在一个机器周期内最多只会进行一次，因此获取指令阶段花费一个机器周期。GameBoy所使用基于ZiLOG Z80定制的CPU是一个8位的CPU，这意味着其每一个指令的操作码长度均为1个字节（8位），且其数据引脚的宽度为8位，即该CPU可以在一个机器周期内一次性读写8位的内存数据。

在执行阶段，CPU会对读取到的操作码进行译码，并执行操作码对应的指令。如果对应的操作需要操作16位数据，或者需要对总线进行读写，则通常该操作需要额外的机器周期来完成，因此执行阶段视具体的指令不同，需要花费1至多个机器周期不等。

GameBoy的CPU采用了流水线设计，下一个指令的获取阶段可以和当前指令的执行阶段并行运行，因此在实际表现上，我们可以认为一个指令只在执行阶段花费时间，并以一个指令在执行阶段的耗时来作为指令的时间开销。此处的例外是跳转指令，由于跳转指令会导致流水线停顿（因为PC值的改变会导致从原先的PC+1地址读取的指令无效），因此在发生跳转以后的下一个指令需要重新执行获取+执行的整个流程。在这种情况下，我们将额外产生的1个机器周期的开销添加到跳转指令中，因此跳转指令是否真正执行跳转会影响该指令时间开销。

实现时钟

在了解了本章节的基础概念以后，就让我们进入到实际的编程中吧。首先，我们需要在模拟器中添加一个稳定的时钟信号，并在每一帧根据帧时间来更新时钟，确保模拟器的各个元件能够按照时钟信号进行更新。首先，我们需要测量每一帧花费的时间，因此我们在App类中加入一个新的变量来保存上一帧的时间：

```
u64 last_frame_ticks;
```

然后在App::update函数中加入测量时间的代码，修改的部分已使用粗体标出：

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

RV App::update()
{
    lutry
    {
        // Update window events.
        Window::poll_events();
        // Exit the program if the window is closed.
        if (window->is_closed())
        {
            is_exiting = true;
            return ok;
        }
        u64 ticks = get_ticks();
        u64 delta_ticks = ticks - last_frame_ticks;
        f64 delta_time = (f64)delta_ticks / get_ticks_per_second();
        delta_time = min(delta_time, 0.125);
        if(emulator)
        {
            emulator->update(delta_time);
        }
        last_frame_ticks = ticks;
        // Draw GUI.
        draw_gui();
        /*...*/
    }
    lucatchret;
    return ok;
}
```

get_ticks()函数会返回当前操作系统的高精度时钟的当前数值，该数值使用64位无符号整数表示，从系统启动开始递增，通常具有纳秒级的精度。get_ticks_per_second()返回高精度时钟的频率，即一秒钟递增多少次，我们通过将两帧之间的时钟数值差除以时钟频率就可以获得按秒表示的帧时间。需要注意的是该时间值需要使用64位浮点表示，而非32位，不然浮点数的精度误差会显著影响时间的计算结果。

在获取了帧时间以后，我们便可以通过帧时间来更新模拟器。在Emulator类中添加如下变量和函数：

```
//! `true` if the emulation is paused.
bool paused = false;
//! The cycles counter.
u64 clock_cycles = 0;
//! The clock speed scale value.
f32 clock_speed_scale = 1.0;
void update(f64 delta_time);
//! Advances clock and updates all hardware states (except CPU).
//! This is called from CPU instructions.
//! @param[in] mcycles The number of machine cycles to tick.
void tick(u32 mcycles);
```

其中：

1. paused变量表示了当前的模拟器是否被暂停，我们添加这一功能，以便在某些时刻可以停止CPU的运行进行调试操作。
2. clock_cycles变量记录了当前模拟器累计运行的时钟周期数，我们可以用该计数器来确定何时更新GameBoy的各个组件。
3. clock_speed_scale可以用于缩放模拟器的更新速度，以在必要时候放慢模拟器的模拟速度来观察中间过程。
4. update函数每帧调用一次，其接受一个delta_time参数，该参数指定了上一渲染帧所花费的时间，以用于计算我们在这一帧中需要模拟的时钟周期数。
5. tick函数用于步进模拟器中除CPU外各个组件的状态，同时增加clock_cycles计数器的值。tick函数接受一个mcycles参数，该参数指定了tick函数需要步进的**机器周期**个数。

然后我们在Emulator.cpp中实现模拟器的update函数和tick函数：

```
void Emulator::update(f64 delta_time)
{
    u64 frame_cycles = (u64)((f32)(4194304.0 * delta_time) * clock_speed_scale)
    u64 end_cycles = clock_cycles + frame_cycles;
    while(clock_cycles < end_cycles)
    {
        /*TODO: step emulator and advance clock_cycles*/
    }
}
void Emulator::tick(u32 mcycles)
{
    u32 tick_cycles = mcycles * 4;
    for(u32 i = 0; i < tick_cycles; ++i)
    {
        ++clock_cycles;
    }
}
```

在Emulator::update函数中，我们首先使用delta_time乘以4194304来计算当前帧需要步进的时钟次数，然后乘以clock_speed_scale来缩放时钟的频率。然后，我们计算end_cycles，其表示该帧结束时clock_cycles计数器应该到达的值。最后，我们使用一个条件循环，如果当前clock_cycles计数器的值小于end_cycles，我们便可以反复步进模拟器，直到clock_cycles达到end_cycles，完成该帧的模拟器更新。在循环步进模拟器时，实现代码会调用Emulator::tick来增加clock_cycles计数器的值，并更新时钟、PPU、APU等各个组件的状态若干次，以实现各个组件之间的时序同步。在现阶段，我们只需要在Emulator::tick中对clock_cycles计数器进行更新，而对于模拟器其余组件的更新，我们将会在之后实现各个组件时将代码添加到该函数中。

实现CPU

新建两个新的文件：CPU.hpp和CPU.cpp来存放CPU相关的代码。CPU.hpp的代码如下：

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

struct Emulator;
struct CPU
{
    //! A register.
    u8 a;
    //! F register.
    u8 f;
    //! B register.
    u8 b;
    //! C register.
    u8 c;
    //! D register.
```

```
    u8 d;
    //! E register.
    u8 e;
    //! H register.
    u8 h;
    //! L register.
    u8 l;
    //! SP register (Stack Pointer).
    u16 sp;
    //! PC register (Program Counter/Pointer).
    u16 pc;
    //! CPU is halted;
    bool halted;

    u16 af() const { return (((u16)a) << 8) + (u16)f; }
    u16 bc() const { return (((u16)b) << 8) + (u16)c; }
    u16 de() const { return (((u16)d) << 8) + (u16)e; }
    u16 hl() const { return (((u16)h) << 8) + (u16)l; }
    void af(u16 v) { a = (u8)(v >> 8); f = (u8)(v & 0xF0); } // The lower 4 bi
    void bc(u16 v) { b = (u8)(v >> 8); c = (u8)(v & 0xFF); }
    void de(u16 v) { d = (u8)(v >> 8); e = (u8)(v & 0xFF); }
    void hl(u16 v) { h = (u8)(v >> 8); l = (u8)(v & 0xFF); }

    bool fz() const { return (f & 0x80) != 0; }
    bool fn() const { return (f & 0x40) != 0; }
    bool fh() const { return (f & 0x20) != 0; }
    bool fc() const { return (f & 0x10) != 0; }
    void set_fz() { f |= 0x80; }
    void reset_fz() { f &= 0x7F; }
    void set_fn() { f |= 0x40; }
    void reset_fn() { f &= 0xBF; }
    void set_fh() { f |= 0x20; }
    void reset_fh() { f &= 0xDF; }
    void set_fc() { f |= 0x10; }
    void reset_fc() { f &= 0xEF; }

    void init();
    void step(Emulator* emu);
};
```

我们创建了一个CPU类型来存储所有CPU相关的状态。在类中，我们为每个寄存器定义了一个变量，这样就可以通过变量存储各个寄存器的值，然后我们还定义了一系列方法，方便后续读写由两个8位寄存器合并成的16位寄存器的值。同时，我们也为F寄存器的每一个标志位定义了各自的读取和设置函数。halted用于判断当前CPU是否通过HALT指令进入了挂起模式，在挂起模式下，我们需要直接跳过CPU的执行。

CPU.cpp代码如下：

```
#include "CPU.hpp"
#include "Emulator.hpp"
void CPU::init()
{
    // ref: https://github.com/rockytriton/LLD_gbemu/raw/main/docs/The%20Cycle-
    af(0x01B0);
    bc(0x0013);
    de(0x00D8);
    hl(0x014D);
    sp = 0xFFFE;
    pc = 0x0100;
    halted = false;
}
void CPU::step(Emulator* emu)
{
    if(!halted)
    {
        // fetch opcode.
        u8 opcode = emu->bus_read(pc);
        // increase counter.
        ++pc;
        // execute opcode.
        // TODO...
    }
    else
    {
        emu->tick(1);
    }
}
```

在CPU::init中，我们初始化了CPU的各个寄存器的值。GameBoy的每个寄存器都有一个特定的值，这些值根据设备的不同而不同，此处我们使用的是初版GameBoy（DMG）中的寄存器初始值。CPU::step负责步进一次CPU，即执行一条CPU指令。根据上文的描述，当CPU没有被暂停时，其首先通过bus_read来调起总线读取PC寄存器指向的地址的值，然后将PC寄存器的值增加1，随后根据操作码来执行对应的操作；而当CPU被暂停时，我们仍然需要调用Emulator::tick(1)来步进一次模拟器，以便其余组件能够更新状态，并在合适的时候通过触发中断来唤醒CPU。

接下来我们需要实现CPU的执行指令部分。我们使用一个256个元素的函数数组来表示CPU的所有指令，每一个指令对应的函数都由其操作码作为数组下标来索引。新建两个文件：Instructions.hpp和Instructions.cpp来存储所有的指令函数。Instructions.hpp内容如下：

```
#pragma once

struct Emulator;
using instruction_func_t = void(Emulator* emu);

//! A map of all instruction functions by their opcodes.
extern instruction_func_t* instructions_map[256];
```

Instructions.cpp内容如下：

```
#include "Instructions.hpp"

instruction_func_t* instructions_map[256] =
{
    nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nul
    nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nul
    nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nul
    nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nul
    nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nul
    nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nul
    nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nul
    nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nul
    nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nul
    nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nul
    nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nul
    nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nul
    nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nul
    nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nul
    nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nul
    nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nul
};
```

可以看到，目前我们的函数数组只是一个全部为空指针的数组。我们将在后面几章中逐步实现各个指令，并将对应的函数填入数组中。有了这个数组以后，我们便可以如下实现CPU::step：

```
#include "CPU.hpp"
#include "Emulator.hpp"
#include <Luna/Runtime/Log.hpp>
#include "Instructions.hpp"
void CPU::init()
{
    // ref: https://github.com/rockytriton/LLD_gbemu/raw/main/docs/The%20Cycle-
    af(0x01B0);
    bc(0x0013);
    de(0x00D8);
    hl(0x014D);
    sp = 0xFFFE;
    pc = 0x0100;
    halted = false;
}
void CPU::step(Emulator* emu)
{
    if(!halted)
    {
        // fetch opcode.
        u8 opcode = emu->bus_read(pc);
        // increase counter.
        ++pc;
        // execute opcode.
        instruction_func_t* instruction = instructions_map[opcode];
        if(!instruction)
        {
            log_error("LunaGB", "Instruction 0x%02X not present.", (u32)opcode
```

```
            emu->paused = true;
        }
        else
        {
            instruction(emu);
        }
    }
    else
    {
        emu->tick(1);
    }
}
```

在上面的代码中，我们使用操作码作为下标来索引对应的指令函数并执行。如果找不到对应的函数，则输出错误信息并暂停模拟器，以便我们能够排查错误。在实际的游戏机中，执行不存在的指令会锁死机器（暂停所有部件的运行），直到机器断电并再次通电才会恢复。

最后，让我们修改Emulator.hpp和Emulator.cpp，将CPU部分的代码接入模拟器：

Emulator.hpp：

```
#pragma once
#include <Luna/Runtime/Result.hpp>
#include "CPU.hpp"
using namespace Luna;

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

    //! `true` if the emulation is paused.
    bool paused = false;
    //! The cycles counter.
    u64 clock_cycles = 0;
    //! The clock speed scale value.
    f32 clock_speed_scale = 1.0;

    CPU cpu;

    RV init(const void* cartridge_data, usize cartridge_data_size);
    void update(f64 delta_time);
    //! Advances clock and updates all hardware states (except CPU).
    //! This is called from CPU instructions.
    //! @param[in] mcycles The number of machine cycles to tick.
    void tick(u32 mcycles);
    void close();
    ~Emulator()
    {
        close();
    }
};
```

Emulator.cpp：

```
#include "Emulator.hpp"
#include "Cartridge.hpp"
#include <Luna/Runtime/Log.hpp>

RV Emulator::init(const void* cartridge_data, usize cartridge_data_size)
{
    /*...*/
    cpu.init();
    return ok;
}
void Emulator::update(f64 delta_time)
{
    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);
    }
}
```

在Emulator::init中，我们添加了代码以初始化CPU的状态。在Emulator::update中，我们反复执行CPU::step来步进CPU，直到当前clock_cycles计数器的值大于等于帧结束时clock_cycles计数器应该到达的值。CPU::step执行的每一条指令都会在内部调用Emulator::tick来增加clock_cycles计数器的值，其参数等于该指令花费的机器周期数。

实现总线

上述的CPU代码中调用了bus_read函数以读取总线数据，接下来就让我们实现总线吧！在Emulator类中添加以下两个方法：

```
u8 bus_read(u16 addr);
void bus_write(u16 addr, u8 data);
```

bus_read用于从指定的地址中读取数据并返回，而bus_write则用于将给定的数据写入指定地址。对于总线读写需要的工作内存和显示内存，我们可以直接在Emulator类中以字节数组的形式定义：

```
byte_t vram[8_kb];
byte_t wram[8_kb];
byte_t hram[128];
```

接下来，我们在Emulator.cpp中实现总线读写函数：

```
u8 Emulator::bus_read(u16 addr)
{
    if(addr <= 0x7FFF)
    {
        // Cartridge ROM.
        return cartridge_read(this, addr);
    }
    if(addr <= 0x9FFF)
    {
        // VRAM.
        return vram[addr - 0x8000];
    }
    if(addr <= 0xBFFF)
    {
        // Cartridge RAM.
        return cartridge_read(this, addr);
    }
    if(addr <= 0xDFFF)
    {
        // Working RAM.
        return wram[addr - 0xC000];
    }
    if(addr >= 0xFF80 && addr <= 0xFFFE)
    {
        return hram[addr - 0xFF80];
    }
    log_error("LunaGB", "Unsupported bus read address: 0x%04X", (u32)addr);
    return 0xFF;
}
void Emulator::bus_write(u16 addr, u8 data)
{
    if(addr <= 0x7FFF)
    {
        // Cartridge ROM.
        cartridge_write(this, addr, data);
        return;
    }
    if(addr <= 0x9FFF)
    {
        // VRAM.
        vram[addr - 0x8000] = data;
        return;
    }
    if(addr <= 0xBFFF)
    {
        // Cartridge RAM.
        cartridge_write(this, addr, data);
        return;
    }
    if(addr <= 0xDFFF)
    {
        // Working RAM.
```

```
        wram[addr - 0xC000] = data;
        return;
    }
    if(addr >= 0xFF80 && addr <= 0xFFFE)
    {
        // High RAM.
        hram[addr - 0xFF80] = data;
        return;
    }
    log_error("LunaGB", "Unsupported bus write address: 0x%04X", (u32)addr);
    return;
}
```

总线读写函数的实现非常简单，我们只需要根据上文所说的映射规则，将对不同地址的读写映射到不同的内存区域上。对于目前还没实现的地址区域，我们简单地输出一个错误信息，然后当作对非法地址的访问处理（读操作返回0xFF，写操作直接忽略）。由于之后我们需要为卡带实现MBC功能，所以这里我们使用cartridge_read和cartridge_write两个函数来封装对卡带数据的读写操作。我们在Cartridge.hpp中添加以下声明：

```
struct Emulator;
u8 cartridge_read(Emulator* emu, u16 addr);
void cartridge_write(Emulator* emu, u16 addr, u8 data);
```

然后在Cartridge.cpp中添加以下实现：

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

由于目前我们只支持单ROM芯片的卡带，因此cartridge_write实际上不执行任何操作，只会记录一个错误然后就返回，而cartridge_read会检查输入的地址范围，如果地址在卡带的ROM数据区域，就返回对应的数据，否则就输出错误并返回0xFF。

最后，别忘了在模拟器初始化的时候将工作内存和显示内存清零：

```
RV Emulator::init(const void* cartridge_data, usize cartridge_data_size)
{
    /*...*/
    cpu.init();
    memzero(wram, 8_kb);
    memzero(vram, 8_kb);
    return ok;
}
```

此时我们可以编译并且运行程序。在加载了卡带以后，读者应该会看到类似如下的输出：

```
[LunaGB]Info: Cartridge Loaded.
[LunaGB]Info: Title    : TENNIS
[LunaGB]Info: Type     : 00 (ROM ONLY)
[LunaGB]Info: ROM Size : 32 KB
[LunaGB]Info: RAM Size : 00 (0)
[LunaGB]Info: LIC Code : 01 (Nintendo R&D1)
[LunaGB]Info: ROM Ver. : 00
[LunaGB]Error: Instruction 0x00 not present.
```

还记得上一章节我们讨论过的卡带数据头吗，我们曾经讲过，从0x100开始的四个字节是卡带程序的入口，并且第一个字节的数据通常为0x00，即一个NOP指令。此处我们加载的卡带《网球》正好就满足这个情况，因此CPU读进了第一个程序的指令码0x00，并且弹出了指令未实现的错误。

以上就是本章节的全部内容了。从下一章开始，我们将开始实现CPU的各个指令集，让CPU能够真正开始运行GameBoy的游戏程序。那么，下一章见！

銀葉吉祥：从零开始实现GameBoy模拟器 #3 加载，比较和跳转指令
22 赞同 · 2 评论 文章

编辑于 2024-02-04 23:32 · IP 属地浙江

中央处理器 (CPU) 游戏机模拟器 Game Boy（GB）

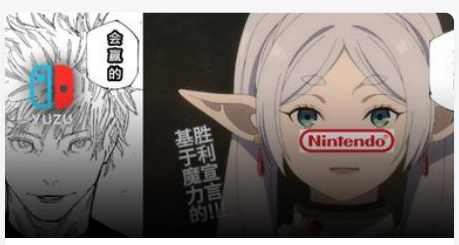
文章被以下专栏收录

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

yuzu模拟器上的shader逆向，流程+工具开发（笔记）
最近买了塞尔达《王国之泪》，在switch上玩过后，想学习一下它的渲染技术，于是从网上下载了柚子模拟器的游戏版本，模拟器是可以使用renderdoc截帧的，但是反出来的shader代码实在是太难懂...
一片枫叶 发表于DSL元语...

游戏主机模拟器的一般设计模式
satur... 发表于计算机技术



一周干废沦落收费的NS模拟器，招来14岁开发者带头冲塔
情报姬

腾讯手游助手、网易MUMU模拟器、雷电模拟器、逍遥模拟
电脑上使用安卓模拟器玩手游的用户都想知道哪个模拟器好用节省电脑资源？现在常用的安卓模拟器就只有腾讯手游助手、MUMU模拟器、雷电模拟器、逍遥模拟器、蓝叠模拟器和夜神模拟器。这几个
走着走着就累了