


从零开始实现GameBoy模拟器 #6 定时器、串口、调试面板

 銀葉吉祥 

浙江大学 软件工程硕士

已关注

20 人赞同了该文章

目录

收起

- 定时器
 - 0xFF04 - DIV （Divider）
 - 0xFF05 - TIMA （Timer Coun
 - 0xFF06 - TMA （Timer Modul
 - 0xFF07 - TAC （Timer Contro
- 实现定时器
- 串口
 - 0xFF01 - SB （Serial Byte）
 - 0xFF02 - SC （Serial Contro
- 实现串口
- 调试面板
 - CPU调试
 - 串口信息
- CPU单元测试



欢迎来到从零开始实现GameBoy模拟器第六章。本章我们将实现GameBoy的定时器和串口组件，同时在应用窗口添加调试面板，以方便我们检查之前实现的CPU以及之后实现的各种组件是否有程序错误。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-06，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

定时器

定时器（Timer）是SoC中内置的一个简单的电路，其会在每一次时钟信号的时候增加内部计数器的值。由于时钟信号具有一个固定的频率，因此我们只需要读取连续两次计数器值，并进行相减就可以转换为实际的时间差。同时，定时器还提供基于中断的定时功能，我们可以设定一个期望的倒计时，并启动定时器，在定时器时间到的时候，定时器会触发TIMER中断来提示程序。

定时器的行为由以下几个寄存器控制：

0xFF04 - DIV （Divider）

DIV寄存器是一个16位寄存器，但只有其高8位可以通过0xFF04总线地址读取，而低8位只能在计时器内部使用，不可以被外部访问。DIV寄存器的值会在每一个时钟周期加1，如果值溢出，则会自动归零。由于DIV寄存器只有高8位可以访问，因此读取0xFF04时，可以认为该值的自增频率是16384Hz（4194304/256）。对0xFF04地址的任何写入操作都会将该寄存器的值重置为0（包括全部的16位）。

由于DIV的寄存器长度较短，其最多只能记录1/64秒的时间长度，因此在大部分情况下，我们都不会直接使用DIV寄存器的值，而是用DIV寄存器的变化驱动下面所述的定时器数值变化，然后再使用定时器功能来计时。

0xFF05 - TIMA （Timer Counter）

TIMA寄存器是一个8位寄存器，在定时器功能启动时，其保存了当前定时器的时间值，并会随着DIV寄存器的变化，按照定制器控制寄存器中选择的规则增加值。当TIMA寄存器的值溢出（超过0xFF）时，其会触发TIMER中断。

0xFF06 - TMA （Timer Modulo）

TMA寄存器是一个8位寄存器，当TIMA寄存器溢出并触发中断后，其会被重置为TMA的所指定的值。显然，TMA的值越大，代表设置的定时时间越短。实际的定时时间可以按照以下公式计算：

$$T = (256 - TMA) * \Delta T$$

其中 ΔT 代表两次TIMA值增加之间的时间间隔。

0xFF07 - TAC（Timer Control）

该寄存器用于控制计时器的行为，其每个位的含义如下：

7	6	5	4	3	2	1	0
-	-	-	-	-	启动	时钟选择1	时钟选择0

- 启动：该标志位设置是否启动计时器。如果该值为0，则计时器被禁用，TIMA停止更新，也不会产生中断。
- 时钟选择：此处的两个比特位用于设置TIMA更新的频率，每个值的含义如下：
 - 0 (00b)：4096Hz（当DIV的位9从1变为0增加TIMA）
 - 1 (01b)：262144Hz（当DIV的位3从1变为0时增加TIMA）
 - 2 (10b)：65536Hz（当DIV的位5从1变为0时增加TIMA）
 - 3 (11b)：16384Hz（当DIV的位7从1变为0时增加TIMA）

上述所有的位均从0开始计算。

实现定时器

在了解了定时器的工作机制后，我们便可以实现定时器。首先新建Timer.hpp和Timer.cpp来保存我们的定时器代码，Timer.hpp内容如下：

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

struct Emulator;
struct Timer
{
    ///! 0xFF04
    ///! Only the high 8-bit is accessible via bus, thus behaves like increment.
    ///! Writing any value to this resets the value to 0.
    u16 div;
    ///! 0xFF05 Timer counter
    ///! Triggers a INT_TIMER when overflows (exceeds 0xFF).
    u8 tima;
    ///! 0xFF06 Timer modulo
    ///! The value to reset TIMA to if overflows.
    u8 tma;
    ///! 0xFF07 Timer control
    u8 tac;

    u8 read_div() const
    {
        return (u8)(div >> 8);
    }
    u8 clock_select() const { return tac & 0x03; }
    bool tima_enabled() const { return bit_test(&tac, 2); }

    void init()
    {
        div = 0xAC00;
        tima = 0;
        tma = 0;
        tac = 0xF8;
    }
    void tick(Emulator* emu);
    u8 bus_read(u16 addr);
    void bus_write(u16 addr, u8 data);
};
```

我们首先定义了所有寄存器对应的变量，然后编写了几个辅助函数来帮助我们 从寄存器中获取不同的值，例如read_div用于读取0xFF04的值（即DIV寄存器的高8位）、clock_select返回选择的时钟的序号、tima_enabled用于返回当前定时器是否启动。init函数用于初始化定时器的状态；tick函数在每个时钟周期调用一次，用于更新定时器的内部状态；bus_read和bus_write函数则是用于将定时器相关的寄存器接到总线上。

Timer.cpp内容如下:

```
#include "Timer.hpp"
#include "Emulator.hpp"

void Timer::tick(Emulator* emu)
{
    // Increase DIV.
    u16 prev_div = div;
    ++div;
    if(tima_enabled())
    {
        // Check whether we need to increase TIMA in this tick.
        bool tima_update = false;
        switch(clock_select())
        {
            case 0:
                tima_update = (prev_div & (1 << 9)) && (!(div & (1 << 9)));
                break;
            case 1:
                tima_update = (prev_div & (1 << 3)) && (!(div & (1 << 3)));
                break;
            case 2:
                tima_update = (prev_div & (1 << 5)) && (!(div & (1 << 5)));
                break;
            case 3:
                tima_update = (prev_div & (1 << 7)) && (!(div & (1 << 7)));
                break;
        }
        if (tima_update)
        {
            if (tima == 0xFF)
            {
                emu->int_flags |= INT_TIMER;
                tma = tma;
            }
            else
            {
                tima++;
            }
        }
    }
}

u8 Timer::bus_read(u16 addr)
{
    luassert(addr >= 0xFF04 && addr <= 0xFF07);
    if(addr == 0xFF04) return read_div();
    if(addr == 0xFF05) return tima;
    if(addr == 0xFF06) return tma;
    if(addr == 0xFF07) return tac;
}

void Timer::bus_write(u16 addr, u8 data)
{
    luassert(addr >= 0xFF04 && addr <= 0xFF07);
    if(addr == 0xFF04)
    {
        div = 0;
        return;
    }
    if(addr == 0xFF05)
    {
        tima = data;
        return;
    }
    if(addr == 0xFF06)
    {
        tma = data;
        return;
    }
    if(addr == 0xFF07)
    {
        tac = 0xF8 | (data & 0x07);
        return;
    }
}
```

在定时器状态更新的时候，我们首先将div寄存器的值自增1，接着在定时功能打开的情况下，我们根据选择的时钟，检查定时器的3、5、7、9位是否产生了下降信号（值从1变为0），并在条

件符合时增加TIMA的值，最后，当TIMA值溢出时，我们将其重置为TMA，并请求TIMER中断。

最后，我们将定时器接入模拟器。在Emulator.hpp中添加定时器：

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

constexpr u8 INT_VBLANK = 1;
constexpr u8 INT_LCD_STAT = 2;
constexpr u8 INT_TIMER = 4;
constexpr u8 INT_SERIAL = 8;
constexpr u8 INT_JOYPAD = 16;

struct Emulator
{
    /*...*/
    //! 0xFFFF - The interruption enabling flags.
    u8 int_enable_flags;

    Timer timer;
    RV init(const void* cartridge_data, usize cartridge_data_size);
    /*...*/
};
```

然后在Emulator.cpp相应的位置调用计时器的相关函数：

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

RV Emulator::init(const void* cartridge_data, usize cartridge_data_size)
{
    /*...*/
    int_enable_flags = 0;
    timer.init();
    return ok;
}
/*...*/
void Emulator::tick(u32 mcycles)
{
    u32 tick_cycles = mcycles * 4;
    for(u32 i = 0; i < tick_cycles; ++i)
    {
        ++clock_cycles;
        timer.tick(this);
    }
}
/*...*/
u8 Emulator::bus_read(u16 addr)
{
    /*...*/
    if(addr <= 0xDFFF)
    {
        // Working RAM.
        return wram[addr - 0xC000];
    }
    if(addr >= 0xFF04 && addr <= 0xFF07)
    {
        return timer.bus_read(addr);
    }
    if(addr == 0xFF0F)
    {
        // IF
        return int_flags | 0xE0;
    }
    /*...*/
    return 0xFF;
}
void Emulator::bus_write(u16 addr, u8 data)
{
    /*...*/
    if(addr <= 0xDFFF)
    {
        // Working RAM.
        wram[addr - 0xC000] = data;
```

```
        return;
    }
    if(addr >= 0xFF04 && addr <= 0xFF07)
    {
        timer.bus_write(addr, data);
        return;
    }
    if(addr == 0xFF0F)
    {
        // IF
        int_flags = data & 0x1F;
        return;
    }
    /*...*/
}
```

至此我们就完成了定时器组件的实现和接入。

串口

串口（serial）用于在GameBoy主机和其它设备之间通信，从而实现联机游戏的功能。这里的其它设备既可以是别的GameBoy主机，也可以是电脑等设备，这些设备通过一根专用的数据线连接到GameBoy的串口接口上。在互联网还没有完全普及的年代，串口是GameBoy与外界通信的唯一渠道。

GameBoy的串口通信协议较为简单，其连接不需要进行握手，也没有任何连接状态，只需要将待传输的数据写入串口数据寄存器，然后启动传输即可。GameBoy的每一次传输只能发送和接收一个字节的数据，因此如果需要发送多字节的数据，则需要启动多次连接，然后逐一发送。GameBoy的串口协议本身不保证数据的完整性和有效性，也无法校验串口的连接状态，其甚至允许在串口未连接任何设备时启动传输，在这种情况下，输出的数据会被丢弃，而读取到的数据始终为0xFF。

GameBoy的串口协议为非对称协议，也就是说，在连接的时候，有一方扮演主机（master GameBoy）的角色，而另一方则扮演从机（slave GameBoy）的角色。在数据传输时，主机负责提供串口的时钟信号，并能够主动发起传输，而从机的串口使用主机的时钟信号，并且无法主动发起传输，只能将需要传输的数据写入串口数据寄存器，然后等待主机发起传输来发送数据。串口的时钟频率决定了串口传输数据的速度，在每一个时钟周期中，串口能够传输1bit的信号，当GameBoy作为主机通信时，单色GameBoy的串口时钟频率固定为8192Hz，因此其传输速度为1KB/s；当GameBoy作为从机通信时，串口所使用的时钟频率取决于外部设备，且不一定与主机的串口时钟频率相等。根据测试，当作为从机通信时，GameBoy能够支持高达500kHz的时钟频率，并且理论上没有最低时钟频率限制（有人测试了串口在1个月更新一次的时钟频率下仍然能正常工作）。

在本专题中，我们将不会真的实现GameBoy的联机功能，因此我们假定在游戏运行时，串口一直保持断开状态（写入串口的数据被丢弃、读取串口的数据永远返回0xFF）。同时，为了方便在之后的小节中查看CPU单元测试ROM的输出，我们会使用一个字节数组来保存从GameBoy中输出的串口数据，并且在之后的调试面板中显示这些数据。

0xFF01 - SB（Serial Byte）

该寄存器保存串口传输的数据。在传输开始前，该寄存器保存需要发送至串口的数据；在传输结束后，该寄存器保存从串口接收的数据；而在传输进行时，该寄存器的数据是发送数据和接收数据的混合体。当传输进行时，在每一个串口的时钟周期下，SB寄存器中的最高位会被通过串口发送，同时整个寄存器的数据左移一位，并且最低位设置成从串口接收到的数据，如下表所示：

7	6	5	4	3	2	1	0
o.7	o.6	o.5	o.4	o.3	o.2	o.1	o.0
o.6	o.5	o.4	o.3	o.2	o.1	o.0	i.7
o.5	o.4	o.3	o.2	o.1	o.0	i.7	i.6
o.4	o.3	o.2	o.1	o.0	i.7	i.6	i.5
o.3	o.2	o.1	o.0	i.7	i.6	i.5	i.4
o.2	o.1	o.0	i.7	i.6	i.5	i.4	i.3
o.1	o.0	i.7	i.6	i.5	i.4	i.3	i.2
o.0	i.7	i.6	i.5	i.4	i.3	i.2	i.1
i.7	i.6	i.5	i.4	i.3	i.2	i.1	i.0

0xFF02 - SC（Serial Control）

该寄存器控制串口的行为，每个位的含义如下：

--	--	--	--	--	--	--	--

7	6	5	4	3	2	1	0
启动传输	-	-	-	-	-	-	时钟选择

时钟选择：该位控制串口传输使用的时钟，设置为1则使用本机时钟，同时本机会作为主机；设置为0则使用外部时钟，同时本机会作为从机。

启动传输：

- 当时钟选择为1时，设置该位为1会启动数据传输。每一次数据传输会传输1比特的数据，通过SB寄存器指定，在传输结束以后，该位会自动置0，同时请求SERIAL中断。
- 当时钟选择位0时，设置该位为1不起作用，因为传输由主机发起和控制。但是如果本机将该位设置成了1，在传输结束以后，该位会自动置0。

当时钟选择位为1时，GameBoy传输数据的流程如下：

1. 将待传输的数据写入SB寄存器
2. 将SC.7设置为1
3. 传输结束时，总线控制器会将SC.7重置为0，同时请求SERIAL中断
4. 在中断处理函数中读取SB寄存器，然后在SB寄存器中写入下一字节要传输的数据。
5. 重复上述流程。

当时钟选择位为0时，GameBoy传输数据的流程如下：

1. 将待传输的数据写入SB寄存器
2. 可以选择将SC.7设置为1，无论是否设置都不影响实际的传输。
3. 主机（另一台GameBoy）启动传输
4. 传输结束时，总线控制器会将SC.7重置为0，同时请求SERIAL中断
5. 在中断处理函数中读取SB寄存器，然后在SB寄存器中写入下一字节要传输的数据。
6. 重复上述流程。

实现串口

由于我们的实现不会往GameBoy中传输任何数据，因此我们在实现时只需要考虑GameBoy的时钟选择位为1的情况（即GameBoy主动发送数据的情况）。首先新建Serial.hpp和Serial.cpp来保存我们的串口代码，Serial.hpp内容如下：

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

struct Emulator;
struct Serial
{
    ///! 0xFF01 Serial transfer data.
    u8 sb;
    ///! 0xFF02 Serial transfer control.
    u8 sc;

    // Serial internal data.

    // Whether data transferring is in progress.
    bool transferring;

    // GameBoy serial output will be placed in this buffer.
    RingDeque<u8> output_buffer;

    // The current transfer out byte.
    u8 out_byte;
    // The transferring bit index (7 to 0).
    i8 transfer_bit;

    bool is_master() const { return bit_test(&sc, 0); }
    bool transfer_enable() const { return bit_test(&sc, 7); }

    void begin_transfer();
    void process_transfer(Emulator* emu);
    void end_transfer(Emulator* emu);

    void init()
    {
        sb = 0xFF;
        sc = 0x7C;
        transferring = false;
    }
    void tick(Emulator* emu);
```

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

我们首先为SB和SC寄存器分配了对应的变量，然后定义了一系列变量用于表示当前的串口状态。is_master函数返回本机是否为主机（即SC.0是否为1），transfer_enable函数返回当前是否启动传输（即SC.7是否为1）。begin_transfer、process_transfer和end_transfer则用于启动、执行和结束传输。

Serial.cpp内容如下：

```
#include "Serial.hpp"
#include "Emulator.hpp"

void Serial::begin_transfer()
{
    transferring = true;
    out_byte = sb;
    transfer_bit = 7;
}
void Serial::process_transfer(Emulator* emu)
{
    sb <=& 1;
    // Set lowest bit to 1.
    ++sb;
    --transfer_bit;
    if(transfer_bit < 0)
    {
        transfer_bit = 0;
        end_transfer(emu);
    }
}
void Serial::end_transfer(Emulator* emu)
{
    output_buffer.push_back(out_byte);
    bit_reset(&sc, 7);
    transferring = false;
    emu->int_flags |= INT_SERIAL;
}
void Serial::tick(Emulator* emu)
{
    if(!transferring && transfer_enable() && is_master())
    {
        begin_transfer();
    }
    else if(transferring)
    {
        process_transfer(emu);
    }
}
u8 Serial::bus_read(u16 addr)
{
    luassert(addr >= 0xFF01 && addr <= 0xFF02);
    if(addr == 0xFF01) return sb;
    if(addr == 0xFF02) return sc;
}
void Serial::bus_write(u16 addr, u8 data)
{
    luassert(addr >= 0xFF01 && addr <= 0xFF02);
    if(addr == 0xFF01)
    {
        sb = data;
        return;
    }
    if(addr == 0xFF02)
    {
        sc = 0x7C | (data & 0x83);
        return;
    }
}
```

串口组件的更新入口是tick函数。当串口组件更新时，其首先检查当前是否需要启动传输，如果当前设备为主机，且设置了SC.7，则我们调用begin_transfer启动传输。begin_transfer会将transferring标志位设置为true，加载我们需要传输的数据至out_byte变量，然后将transfer_bit设置为7，表示我们下一个传输的比特位是7。

当我们处在传输状态（transferring为true）时，tick函数调用process_transfer函数执行一次传

输。process_transfer函数会将SC的数值左移一位，同时最低位设置为1，因为我们的实现只考虑断开串口的情况，而在断开串口的情况下，读取串口的值永远返回1。然后将transfer_bit减去1，表示下一个需要传输的比特位。process_transfer会被多次调用，直到transfer_bit的值小于0。当transfer_bit的值小于0时，表示我们已经传输完了所有的比特位，此时我们需要调用end_transfer结束传输，该函数将之前加载的out_byte变量放入output_buffer字节FIFO队列中，然后重置SC.7和transferring，并请求CPU的SERIAL中断。

最后我们修改Emulator.hpp和Emulator.cpp，将串口接入模拟器。Emulator.hpp修改如下：

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

constexpr u8 INT_VBLANK = 1;
constexpr u8 INT_LCD_STAT = 2;
constexpr u8 INT_TIMER = 4;
constexpr u8 INT_SERIAL = 8;
constexpr u8 INT_JOYPAD = 16;

struct Emulator
{
    /**...*/
    Timer timer;
    Serial serial;
    /**...*/
};
```

Emulator.cpp修改如下：

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

RV Emulator::init(const void* cartridge_data, usize cartridge_data_size)
{
    /**...*/
    timer.init();
    serial.init();
    return ok;
}
/**...*/
void Emulator::tick(u32 mcycles)
{
    u32 tick_cycles = mcycles * 4;
    for(u32 i = 0; i < tick_cycles; ++i)
    {
        ++clock_cycles;
        timer.tick(this);
        if((clock_cycles % 512) == 0)
        {
            // Serial is ticked at 8192Hz.
            serial.tick(this);
        }
    }
}
/**...*/
u8 Emulator::bus_read(u16 addr)
{
    /**...*/
    if(addr <= 0xDFFF)
    {
        // Working RAM.
        return wram[addr - 0xC000];
    }
    if(addr >= 0xFF01 && addr <= 0xFF02)
    {
        return serial.bus_read(addr);
    }
    if(addr >= 0xFF04 && addr <= 0xFF07)
    {
        return timer.bus_read(addr);
    }
    /**...*/
    return 0xFF;
```



```

}
void Emulator::bus_write(u16 addr, u8 data)
{
    /*...*/
    if(addr <= 0xDFFF)
    {
        // Working RAM.
        wram[addr - 0xC000] = data;
        return;
    }
    if(addr >= 0xFF01 && addr <= 0xFF02)
    {
        serial.bus_write(addr, data);
        return;
    }
    if(addr >= 0xFF04 && addr <= 0xFF07)
    {
        timer.bus_write(addr, data);
        return;
    }
    /*...*/
    return;
}
```

至此我们就完成了串口组件的实现和接入。

调试面板

GameBoy是一个由多个组件构成的，较为复杂的系统，因此在开发GameBoy时，对系统的内部状态进行观察有助于帮助我们及时发现系统中可能的bug，确保各个组件正确实现。在本小节中，我们将实现一个GameBoy的调试面板，用于集中显示GameBoy系统运行时的各种信息，并允许模拟器以调试模式运行游戏，以便我们进行漏洞排查。在本小节中，我们将为调试面板实现以下功能：

- 1. 显示CPU各个寄存器值。
- 2. 控制系统时钟频率的快慢，从而控制CPU和各个组件的运行速度。
- 3. 在暂停模式下，允许步进CPU（单步执行指令），以跟踪CPU的执行情况。
- 4. 记录和保存CPU执行日志，以便事后与参考实现比较，定位差异点。

首先添加DebugWindow.hpp和DebugWindow.cpp文件存档我们的调试面板代码。
DebugWindow.hpp代码如下：

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

struct DebugWindow
{
    bool show = false;

    void gui();
};
```

DebugWindow.cpp代码如下：

```
#include "DebugWindow.hpp"
#include "App.hpp"
#include <Luna/ImGui/ImGui.hpp>
#include <Luna/Window/FileDialog.hpp>
#include <Luna/Runtime/File.hpp>

void DebugWindow::gui()
{
    if(ImGui::Begin("Debug Window", &show))
    {
        // TODO: add debug window gui code.
    }
    ImGui::End();
}
```

然后在App.hpp的App类中加入调试面板类型，同时加入一个g_app的前置定义，方便我们之后在DebugWindow.cpp中引用g_app实例：

```
#pragma once
#include <Luna/RHI/Device.hpp>
#include "Emulator.hpp"
#include <Luna/Runtime/UniquePtr.hpp>
#include "DebugWindow.hpp"
using namespace Luna;

struct App
{
    /*...*/
    //! The ticks for last frame.
    u64 last_frame_ticks;

    //! The debug window context.
    DebugWindow debug_window;

    RV init();
    /*...*/
};

extern App* g_app;
```

在App.cpp中，我们在绘制主菜单时添加一个Debug菜单，允许用户点击后打开调试窗口，并在绘制GUI时绘制调试窗口的GUI：

```
void App::draw_gui()
{
    // Begin GUI.
    ImGuiUtils::update_io();
    ImGui::NewFrame();

    draw_main_menu_bar();

    if(debug_window.show)
    {
        debug_window.gui();
    }

    // End GUI.
    ImGui::Render();
}
void App::draw_main_menu_bar()
{
    if (ImGui::BeginMainMenuBar())
    {
        if (ImGui::BeginMenu("File"))
        {
            if(ImGui::MenuItem("Open"))
            {
                open_cartridge();
            }
            if(ImGui::MenuItem("Close"))
            {
                close_cartridge();
            }
            ImGui::EndMenu();
        }
        if (ImGui::BeginMenu("Debug"))
        {
            if(ImGui::MenuItem("Debug Window"))
            {
                debug_window.show = true;
            }
            ImGui::EndMenu();
        }
        ImGui::EndMainMenuBar();
    }
}
```

此时，当我们运行程序时，就可以在主菜单的Debug菜单下找到Debug Window选项，点击后即可打开调试面板。目前的调试面板只是一个空白的窗口，我们将在下面逐一实现它的各种调试功能。

CPU调试

在DebugWindow类中添加一个cpi_gui函数，用来绘制所有与CPU调试有关的方法：

```
void cpu_gui();
```

然后在主绘制方法中调用它：

```
void DebugWindow::gui()
{
    if(ImGui::Begin("Debug Window", &show))
    {
        cpu_gui();
    }
    ImGui::End();
}
```

首先我们为CPU调试面板添加寄存器值显示的功能：

```
void DebugWindow::cpu_gui()
{
    if(g_app->emulator)
    {
        if(ImGui::CollapsingHeader("CPU Info"))
        {
            auto& cpu = g_app->emulator->cpu;
            if(ImGui::BeginTable("Byte registers", 6, ImGuiTableFlags_Borders))
            {
                ImGui::TableNextRow();
                ImGui::TableSetColumnIndex(0);
                ImGui::Text("AF");
                ImGui::TableNextColumn();
                ImGui::Text("%4.4X", (u32)cpu.af());
                ImGui::TableNextColumn();
                ImGui::Text("A");
                ImGui::TableNextColumn();
                ImGui::Text("%2.2X", (u32)cpu.a);
                ImGui::TableNextColumn();
                ImGui::Text("F");
                ImGui::TableNextColumn();
                ImGui::Text("%2.2X", (u32)cpu.f);

                ImGui::TableNextRow();
                ImGui::TableSetColumnIndex(0);
                ImGui::Text("BC");
                ImGui::TableNextColumn();
                ImGui::Text("%4.4X", (u32)cpu.bc());
                ImGui::TableNextColumn();
                ImGui::Text("B");
                ImGui::TableNextColumn();
                ImGui::Text("%2.2X", (u32)cpu.b);
                ImGui::TableNextColumn();
                ImGui::Text("C");
                ImGui::TableNextColumn();
                ImGui::Text("%2.2X", (u32)cpu.c);

                ImGui::TableNextRow();
                ImGui::TableSetColumnIndex(0);
                ImGui::Text("DE");
                ImGui::TableNextColumn();
                ImGui::Text("%4.4X", (u32)cpu.de());
                ImGui::TableNextColumn();
                ImGui::Text("D");
                ImGui::TableNextColumn();
                ImGui::Text("%2.2X", (u32)cpu.d);
                ImGui::TableNextColumn();
                ImGui::Text("E");
                ImGui::TableNextColumn();
                ImGui::Text("%2.2X", (u32)cpu.e);

                ImGui::TableNextRow();
                ImGui::TableSetColumnIndex(0);
                ImGui::Text("HL");
                ImGui::TableNextColumn();
                ImGui::Text("%4.4X", (u32)cpu.hl());
                ImGui::TableNextColumn();
                ImGui::Text("H");
                ImGui::TableNextColumn();
                ImGui::Text("%2.2X", (u32)cpu.h);
                ImGui::TableNextColumn();
                ImGui::Text("L");
```

```
        ImGui::TableNextColumn();
        ImGui::Text("%2.2X", (u32)cpu.l);

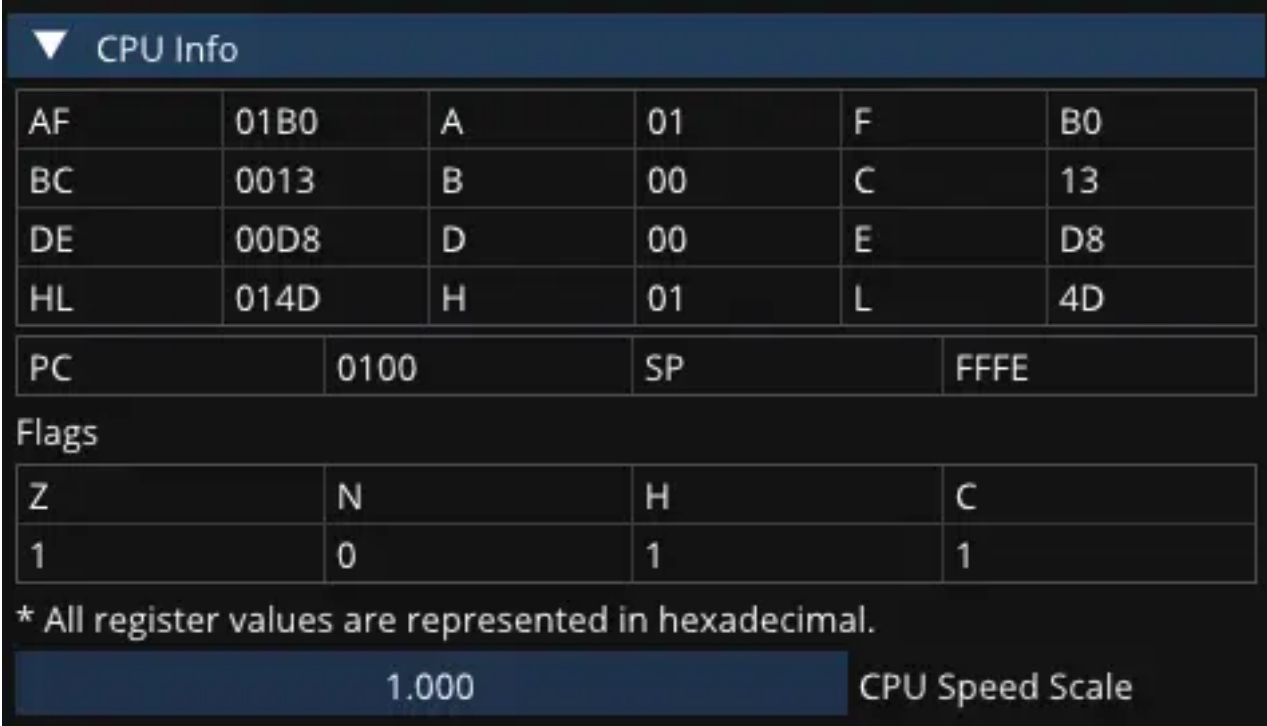
        ImGui::EndTable();
    }
    if(ImGui::BeginTable("Word registers", 4, ImGuiTableFlags_Borders))
    {
        ImGui::TableNextRow();
        ImGui::TableSetColumnIndex(0);
        ImGui::Text("PC");
        ImGui::TableNextColumn();
        ImGui::Text("%4.4X", (u32)cpu.pc);
        ImGui::TableNextColumn();
        ImGui::Text("SP");
        ImGui::TableNextColumn();
        ImGui::Text("%4.4X", (u32)cpu.sp);

        ImGui::EndTable();
    }
    ImGui::Text("Flags");
    if(ImGui::BeginTable("Flags", 4, ImGuiTableFlags_Borders))
    {
        ImGui::TableNextRow();
        ImGui::TableSetColumnIndex(0);
        ImGui::Text("Z");
        ImGui::TableNextColumn();
        ImGui::Text("N");
        ImGui::TableNextColumn();
        ImGui::Text("H");
        ImGui::TableNextColumn();
        ImGui::Text("C");

        ImGui::TableNextRow();
        ImGui::TableSetColumnIndex(0);
        ImGui::Text(cpu.fz() ? "1" : "0");
        ImGui::TableNextColumn();
        ImGui::Text(cpu.fn() ? "1" : "0");
        ImGui::TableNextColumn();
        ImGui::Text(cpu.fh() ? "1" : "0");
        ImGui::TableNextColumn();
        ImGui::Text(cpu.fc() ? "1" : "0");

        ImGui::EndTable();
    }
    ImGui::Text("* All register values are represented in hexadecimal.");
    if(cpu.halted)
    {
        ImGui::Text("CPU Halted.");
    }
    ImGui::DragFloat("CPU Speed Scale", &g_app->emulator->clock_speed, 0.001f);
}
}
```

这段代码看似冗长，实际上就是绘制了两个表格，用于显示CPU的各个寄存器的值，以及其是否处于停止状态。我们同时提供了一个CPU速度缩放的功能，通过调节时钟的缩放倍率来调节CPU的运行速度。该界面绘制出来类似下图所示：



接着我们需要为CPU添加模拟器暂停状态下的单步执行指令功能。为了实现该功能，首先我们需要能够让模拟器处于暂停状态，因此我们在App::draw_main_menu_bar中添加一些额外的指令：

```
void App::draw_main_menu_bar()
{
    if (ImGui::BeginMainMenuBar())
    {
        if (ImGui::BeginMenu("File"))
        {
            if(ImGui::MenuItem("Open"))
            {
                open_cartridge();
            }
            if(ImGui::MenuItem("Open without playing"))
            {
                open_cartridge();
                if(emulator)
                {
                    emulator->paused = true;
                }
            }
            /*...*/
            ImGui::EndMenu();
        }
        if (ImGui::BeginMenu("Play"))
        {
            if(ImGui::MenuItem("Play"))
            {
                if(emulator)
                {
                    emulator->paused = false;
                }
            }
            if(ImGui::MenuItem("Pause"))
            {
                if(emulator)
                {
                    emulator->paused = true;
                }
            }
            ImGui::EndMenu();
        }
        if (ImGui::BeginMenu("Debug"))
        {
            /*...*/
        }
        ImGui::EndMainMenuBar();
    }
}
```

我们在File菜单中添加了一个Open without playing的选项，该选项与常规的Open逻辑类似，只不过其在加载ROM文件以后会立即暂停模拟器的运行，以便我们可以开始调试模拟器。同时，我们添加了一个Play菜单，其中有两个选项：Play和Pause，以便我们在加载卡带以后可以手动暂停和恢复模拟器运行。

接下来，我们在DebugWindow::cpu_gui中添加CPU调试的功能：

```
void DebugWindow::cpu_gui()
{
    if(g_app->emulator)
    {
        if(ImGui::CollapsingHeader("CPU Info"))
        {
            /*...*/
        }
        if(ImGui::CollapsingHeader("CPU Stepping"))
        {
            bool cpu_stepping_enabled = g_app->emulator->paused;
            if(!cpu_stepping_enabled)
            {
                ImGui::Text("CPU stepping is enabled only when the game is paused");
            }
            else
            {
                ImGui::Text("Next instruction: %s", get_opcode_name(g_app->emulator->cpu.get()));
                if(ImGui::Button("Step CPU"))
                {
                    g_app->emulator->cpu.step(g_app->emulator.get());
                }
            }
        }
    }
}
```



```
    }  
}
```

当模拟器在运行时，CPU步进功能被禁用，因此其会显示一行文本，提示用户先暂停模拟器。在模拟器暂停的情况下，我们可以点击Step CPU按钮来手动让CPU执行下一条指令，而按钮上方的文本则会提示用户下一条指令是什么。我们使用一个get_opcode_name函数来根据操作码返回具体的指令，如下所示：

```
const c8* instruction_names[256] = {  
    "x00 NOP",  
    "x01 LD BC, d16",  
    "x02 LD (BC), A",  
    "x03 INC BC",  
    "x04 INC B",  
    "x05 DEC B",  
    "x06 LD B, d8",  
    "x07 RLCA",  
    "x08 LD (a16), SP",  
    "x09 ADD HL, BC",  
    "x0A LD A, (BC)",  
    "x0B DEC BC",  
    "x0C INC C",  
    "x0D DEC C",  
    "x0E LD C, d8",  
    "x0F RRCA",  
    "x10 STOP 0",  
    "x11 LD DE, d16",  
    "x12 LD (DE), A",  
    "x13 INC DE",  
    "x14 INC D",  
    "x15 DEC D",  
    "x16 LD D, d8",  
    "x17 RLA",  
    "x18 JR r8",  
    "x19 ADD HL, DE",  
    "x1A LD A, (DE)",  
    "x1B DEC DE",  
    "x1C INC E",  
    "x1D DEC E",  
    "x1E LD E, d8",  
    "x1F RRA",  
    "x20 JR NZ, r8",  
    "x21 LD HL, d16",  
    "x22 LD (HL+), A",  
    "x23 INC HL",  
    "x24 INC H",  
    "x25 DEC H",  
    "x26 LD H, d8",  
    "x27 DAA",  
    "x28 JR Z, r8",  
    "x29 ADD HL, HL",  
    "x2A LD A, (HL+)",  
    "x2B DEC HL",  
    "x2C INC L",  
    "x2D DEC L",  
    "x2E LD L, d8",  
    "x2F CPL",  
    "x30 JR NC, r8",  
    "x31 LD SP, d16",  
    "x32 LD (HL-), A",  
    "x33 INC SP",  
    "x34 INC (HL)",  
    "x35 DEC (HL)",  
    "x36 LD (HL), d8",  
    "x37 SCF",  
    "x38 JR C, r8",  
    "x39 ADD HL, SP",  
    "x3A LD A, (HL-)",  
    "x3B DEC SP",  
    "x3C INC A",  
    "x3D DEC A",  
    "x3E LD A, d8",  
    "x3F CCF",  
    "x40 LD B, B",  
    "x41 LD B, C",  
    "x42 LD B, D",  
    "x43 LD B, E",  
    "x44 LD B, H",  
    "x45 LD B, L",  
    "x46 LD B, (HL)",  
    "x47 LD B, (DE)",  
    "x48 LD B, (BC)",  
    "x49 LD B, (IX)",  
    "x4A LD B, (IY)",  
    "x4B LD B, (a16)",  
    "x4C LD B, (a16)",  
    "x4D LD B, (a16)",  
    "x4E LD B, (a16)",  
    "x4F LD B, (a16)",  
    "x50 LD C, B",  
    "x51 LD C, C",  
    "x52 LD C, D",  
    "x53 LD C, E",  
    "x54 LD C, H",  
    "x55 LD C, L",  
    "x56 LD C, (HL)",  
    "x57 LD C, (DE)",  
    "x58 LD C, (BC)",  
    "x59 LD C, (IX)",  
    "x5A LD C, (IY)",  
    "x5B LD C, (a16)",  
    "x5C LD C, (a16)",  
    "x5D LD C, (a16)",  
    "x5E LD C, (a16)",  
    "x5F LD C, (a16)",  
    "x60 LD D, B",  
    "x61 LD D, C",  
    "x62 LD D, D",  
    "x63 LD D, E",  
    "x64 LD D, H",  
    "x65 LD D, L",  
    "x66 LD D, (HL)",  
    "x67 LD D, (DE)",  
    "x68 LD D, (BC)",  
    "x69 LD D, (IX)",  
    "x6A LD D, (IY)",  
    "x6B LD D, (a16)",  
    "x6C LD D, (a16)",  
    "x6D LD D, (a16)",  
    "x6E LD D, (a16)",  
    "x6F LD D, (a16)",  
    "x70 LD E, B",  
    "x71 LD E, C",  
    "x72 LD E, D",  
    "x73 LD E, E",  
    "x74 LD E, H",  
    "x75 LD E, L",  
    "x76 LD E, (HL)",  
    "x77 LD E, (DE)",  
    "x78 LD E, (BC)",  
    "x79 LD E, (IX)",  
    "x7A LD E, (IY)",  
    "x7B LD E, (a16)",  
    "x7C LD E, (a16)",  
    "x7D LD E, (a16)",  
    "x7E LD E, (a16)",  
    "x7F LD E, (a16)",  
    "x80 LD H, B",  
    "x81 LD H, C",  
    "x82 LD H, D",  
    "x83 LD H, E",  
    "x84 LD H, H",  
    "x85 LD H, L",  
    "x86 LD H, (HL)",  
    "x87 LD H, (DE)",  
    "x88 LD H, (BC)",  
    "x89 LD H, (IX)",  
    "x8A LD H, (IY)",  
    "x8B LD H, (a16)",  
    "x8C LD H, (a16)",  
    "x8D LD H, (a16)",  
    "x8E LD H, (a16)",  
    "x8F LD H, (a16)",  
    "x90 LD L, B",  
    "x91 LD L, C",  
    "x92 LD L, D",  
    "x93 LD L, E",  
    "x94 LD L, H",  
    "x95 LD L, L",  
    "x96 LD L, (HL)",  
    "x97 LD L, (DE)",  
    "x98 LD L, (BC)",  
    "x99 LD L, (IX)",  
    "x9A LD L, (IY)",  
    "x9B LD L, (a16)",  
    "x9C LD L, (a16)",  
    "x9D LD L, (a16)",  
    "x9E LD L, (a16)",  
    "x9F LD L, (a16)",  
    "xA0 LD B, B",  
    "xA1 LD B, C",  
    "xA2 LD B, D",  
    "xA3 LD B, E",  
    "xA4 LD B, H",  
    "xA5 LD B, L",  
    "xA6 LD B, (HL)",  
    "xA7 LD B, (DE)",  
    "xA8 LD B, (BC)",  
    "xA9 LD B, (IX)",  
    "xAA LD B, (IY)",  
    "xAB LD B, (a16)",  
    "xAC LD B, (a16)",  
    "xAD LD B, (a16)",  
    "xAE LD B, (a16)",  
    "xAF LD B, (a16)",  
    "xB0 LD C, B",  
    "xB1 LD C, C",  
    "xB2 LD C, D",  
    "xB3 LD C, E",  
    "xB4 LD C, H",  
    "xB5 LD C, L",  
    "xB6 LD C, (HL)",  
    "xB7 LD C, (DE)",  
    "xB8 LD C, (BC)",  
    "xB9 LD C, (IX)",  
    "xBA LD C, (IY)",  
    "xBB LD C, (a16)",  
    "xBC LD C, (a16)",  
    "xBD LD C, (a16)",  
    "xBE LD C, (a16)",  
    "xBF LD C, (a16)",  
    "xC0 LD D, B",  
    "xC1 LD D, C",  
    "xC2 LD D, D",  
    "xC3 LD D, E",  
    "xC4 LD D, H",  
    "xC5 LD D, L",  
    "xC6 LD D, (HL)",  
    "xC7 LD D, (DE)",  
    "xC8 LD D, (BC)",  
    "xC9 LD D, (IX)",  
    "xCA LD D, (IY)",  
    "xCB LD D, (a16)",  
    "xCC LD D, (a16)",  
    "xCD LD D, (a16)",  
    "xCE LD D, (a16)",  
    "xCF LD D, (a16)",  
    "xD0 LD E, B",  
    "xD1 LD E, C",  
    "xD2 LD E, D",  
    "xD3 LD E, E",  
    "xD4 LD E, H",  
    "xD5 LD E, L",  
    "xD6 LD E, (HL)",  
    "xD7 LD E, (DE)",  
    "xD8 LD E, (BC)",  
    "xD9 LD E, (IX)",  
    "xDA LD E, (IY)",  
    "xDB LD E, (a16)",  
    "xDC LD E, (a16)",  
    "xDD LD E, (a16)",  
    "xDE LD E, (a16)",  
    "xDF LD E, (a16)",  
    "xE0 LD H, B",  
    "xE1 LD H, C",  
    "xE2 LD H, D",  
    "xE3 LD H, E",  
    "xE4 LD H, H",  
    "xE5 LD H, L",  
    "xE6 LD H, (HL)",  
    "xE7 LD H, (DE)",  
    "xE8 LD H, (BC)",  
    "xE9 LD H, (IX)",  
    "xEA LD H, (IY)",  
    "xEB LD H, (a16)",  
    "xEC LD H, (a16)",  
    "xED LD H, (a16)",  
    "xEE LD H, (a16)",  
    "xEF LD H, (a16)",  
    "xF0 LD L, B",  
    "xF1 LD L, C",  
    "xF2 LD L, D",  
    "xF3 LD L, E",  
    "xF4 LD L, H",  
    "xF5 LD L, L",  
    "xF6 LD L, (HL)",  
    "xF7 LD L, (DE)",  
    "xF8 LD L, (BC)",  
    "xF9 LD L, (IX)",  
    "xFA LD L, (IY)",  
    "xFB LD L, (a16)",  
    "xFC LD L, (a16)",  
    "xFD LD L, (a16)",  
    "xFE LD L, (a16)",  
    "xFF LD L, (a16)"  
}
```

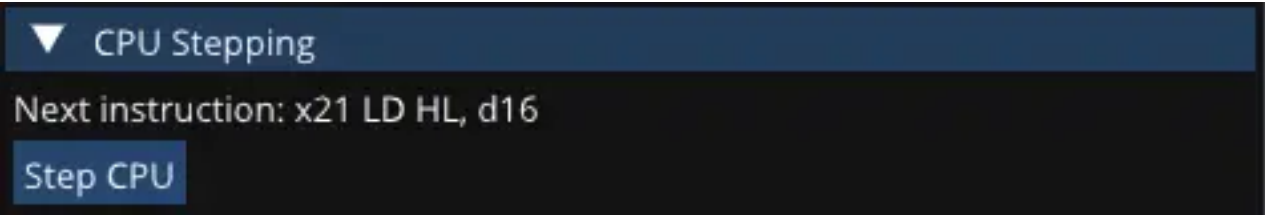
```
"x44 LD B, H",
"x45 LD B, L",
"x46 LD B, (HL)",
"x47 LD B, A",
"x48 LD C, B",
"x49 LD C, C",
"x4A LD C, D",
"x4B LD C, E",
"x4C LD C, H",
"x4D LD C, L",
"x4E LD C, (HL)",
"x4F LD C, A",
"x50 LD D, B",
"x51 LD D, C",
"x52 LD D, D",
"x53 LD D, E",
"x54 LD D, H",
"x55 LD D, L",
"x56 LD D, (HL)",
"x57 LD D, A",
"x58 LD E, B",
"x59 LD E, C",
"x5A LD E, D",
"x5B LD E, E",
"x5C LD E, H",
"x5D LD E, L",
"x5E LD E, (HL)",
"x5F LD E, A",
"x60 LD H, B",
"x61 LD H, C",
"x62 LD H, D",
"x63 LD H, E",
"x64 LD H, H",
"x65 LD H, L",
"x66 LD H, (HL)",
"x67 LD H, A",
"x68 LD L, B",
"x69 LD L, C",
"x6A LD L, D",
"x6B LD L, E",
"x6C LD L, H",
"x6D LD L, L",
"x6E LD L, (HL)",
"x6F LD L, A",
"x70 LD (HL), B",
"x71 LD (HL), C",
"x72 LD (HL), D",
"x73 LD (HL), E",
"x74 LD (HL), H",
"x75 LD (HL), L",
"x76 HALT",
"x77 LD (HL), A",
"x78 LD A, B",
"x79 LD A, C",
"x7A LD A, D",
"x7B LD A, E",
"x7C LD A, H",
"x7D LD A, L",
"x7E LD A, (HL)",
"x7F LD A, A",
"x80 ADD A, B",
"x81 ADD A, C",
"x82 ADD A, D",
"x83 ADD A, E",
"x84 ADD A, H",
"x85 ADD A, L",
"x86 ADD A, (HL)",
"x87 ADD A, A",
"x88 ADC A, B",
"x89 ADC A, C",
"x8A ADC A, D",
"x8B ADC A, E",
"x8C ADC A, H",
"x8D ADC A, L",
"x8E ADC A, (HL)",
"x8F ADC A, A",
"x90 SUB B",
"x91 SUB C",
"x92 SUB D",
```

```
"x93 SUB E",
"x94 SUB H",
"x95 SUB L",
"x96 SUB (HL)",
"x97 SUB A",
"x98 SBC A, B",
"x99 SBC A, C",
"x9A SBC A, D",
"x9B SBC A, E",
"x9C SBC A, H",
"x9D SBC A, L",
"x9E SBC A, (HL)",
"x9F SBC A, A",
"xA0 AND B",
"xA1 AND C",
"xA2 AND D",
"xA3 AND E",
"xA4 AND H",
"xA5 AND L",
"xA6 AND (HL)",
"xA7 AND A",
"xA8 XOR B",
"xA9 XOR C",
"xAA XOR D",
"xAB XOR E",
"xAC XOR H",
"xAD XOR L",
"xAE XOR (HL)",
"xAF XOR A",
"xB0 OR B",
"xB1 OR C",
"xB2 OR D",
"xB3 OR E",
"xB4 OR H",
"xB5 OR L",
"xB6 OR (HL)",
"xB7 OR A",
"xB8 CP B",
"xB9 CP C",
"xBA CP D",
"xBB CP E",
"xBC CP H",
"xBD CP L",
"xBE CP (HL)",
"xBF CP A",
"xC0 RET NZ",
"xC1 POP BC",
"xC2 JP NZ, a16",
"xC3 JP a16",
"xC4 CALL NZ, a16",
"xC5 PUSH BC",
"xC6 ADD A, d8",
"xC7 RST 00H",
"xC8 RET Z",
"xC9 RET",
"xCA JP Z, a16",
"xCB PREFIX CB",
"xCC CALL Z, a16",
"xCD CALL a16",
"xCE ADC A, d8",
"xCF RST 08H",
"xD0 RET NC",
"xD1 POP DE",
"xD2 JP NC, a16",
"xD3 NULL",
"xD4 CALL NC, a16",
"xD5 PUSH DE",
"xD6 SUB d8",
"xD7 RST 10H",
"xD8 RET C",
"xD9 RETI",
"xDA JP C, a16",
"xDB NULL",
"xDC CALL C, a16",
"xDD NULL",
"xDE SBC A, d8",
"xDF RST 18H",
"xE0 LDH (a8), A",
"xE1 POP HL",
```

```
    "xE2 LD (C), A",
    "xE3 NULL",
    "xE4 NULL",
    "xE5 PUSH HL",
    "xE6 AND d8",
    "xE7 RST 20H",
    "xE8 ADD SP, r8",
    "xE9 JP (HL)",
    "xEA LD (a16), A",
    "xEB NULL",
    "xEC NULL",
    "xED NULL",
    "xEE XOR d8",
    "xEF RST 28H",
    "xF0 LDH A, (a8)",
    "xF1 POP AF",
    "xF2 LD A, (C)",
    "xF3 DI",
    "xF4 NULL",
    "xF5 PUSH AF",
    "xF6 OR d8",
    "xF7 RST 30H",
    "xF8 LD HL, SP+r8",
    "xF9 LD SP, HL",
    "xFA LD A, (a16)",
    "xFB EI",
    "xFC NULL",
    "xFD NULL",
    "xFE CP d8",
    "xFF RST 38H"
};

const c8* get_opcode_name(u8 opcode)
{
    return instruction_names[opcode];
}
```

CPU调试面板显示效果如下：



最后，我们需要实现CPU的日志打印功能。该功能会让CPU在每执行一条语句前都将CPU相关的状态作为日志输出到一个字符串中，然后我们可以将字符串中的日志保存成文本文件。在之后的开发过程中，当我们发现自己的模拟器与参考实现的行为不一致时，我们可以将同一个ROM文件在两个模拟器中分别运行一遍，记录并导出日志数据，再使用诸如Beyond Compare等软件比较两者的日志来定位错误发生的具体指令和内存地址。

我们先在DebugWindow中添加两个变量来存储当前的日志系统状态：

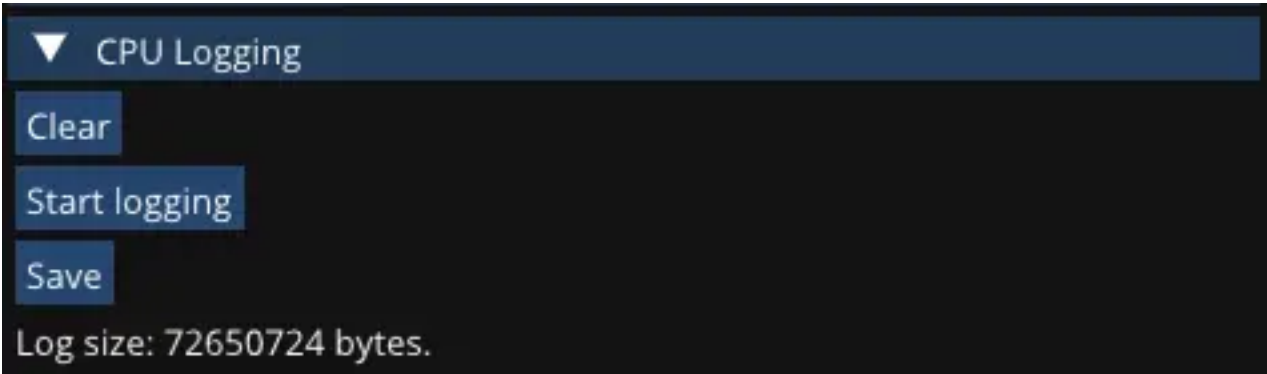
```
// CPU log.
String cpu_log;
bool cpu_logging = false;
```

接着在DebugWindow::cpu_gui中添加CPU日志的相关代码：

```
void DebugWindow::cpu_gui()
{
    if(g_app->emulator)
    {
        if(ImGui::CollapsingHeader("CPU Info"))
        {
            /*...*/
        }
        if(ImGui::CollapsingHeader("CPU Stepping"))
        {
            /*...*/
        }
        if(ImGui::CollapsingHeader("CPU Logging"))
        {
            if(ImGui::Button("Clear"))
            {
                cpu_log.clear();
            }
            if(cpu_logging)
```

```
        {
            ImGui::Button("Stop logging")
            {
                cpu_logging = false;
            }
        }
        else
        {
            ImGui::Button("Start logging")
            {
                cpu_logging = true;
            }
        }
        ImGui::Button("Save")
        {
            Window::FileDialogFilter filter;
            filter.name = "Text file";
            const c8* extension = "txt";
            filter.extensions = {&extension, 1};
            auto rpath = Window::save_file_dialog("Save", {&filter, 1});
            if (succeeded(rpath) && !rpath.get().empty())
            {
                if(rpath.get().extension() == Name())
                {
                    rpath.get().replace_extension("txt");
                }
                auto f = open_file(rpath.get().encode().c_str(), FileOpenF
                if(succeeded(f))
                {
                    auto _ = f.get()->write(cpu_log.c_str(), cpu_log.size()
                }
            }
        }
        ImGui::Text("Log size: %llu bytes.", (u64)cpu_log.size());
    }
}
```

在UI中，我们提供了一个Clear按钮，供用户清楚现有的日志数据，以及一个开启日志记录和停止日志记录的按钮，根据当前是否正在记录日志而切换。最后，我们提供了一个保存日志的按钮，当用户点击Save时，程序会通过Window::save_file_dialog函数弹出一个保存文件对话框，供用户选择保存的位置以及文件名。当用户选择有效的路径和文件名后，程序会使用open_file打开文件并写入数据。最后，我们会显示当前日志占用的内存容量，避免用户无意间记录了过多的数据。当模拟器以正常速度运行时，只需几秒钟就可以使得日志数据量增加100MB。日志记录功能UI类似如下所示：



最后，我们修改CPU类的代码，以添加日志记录支持。在CPU类中添加一个log函数：

```
void log(Emulator* emu);
```

然后在CPU.cpp中实现该函数：

```
void CPU::log(Emulator* emu)
{
    c8 buf[256];
    c8 flags[16];
    snprintf(flags, 16, "%C%C%C%C",
        fz() ? 'Z' : '-',
        fn() ? 'N' : '-',
        fh() ? 'H' : '-',
        fc() ? 'C' : '-'
    );
    snprintf(buf, 256, "%02X %02X %02X A: %02X F: %s BC: %04X DE: %04X HL: %04X",
        //get_opcode_name(emu->bus_read(emu->cpu.pc)),
        emu->bus_read(emu->cpu.pc),
        emu->bus_read(emu->cpu.pc + 1),
```



```
        emu->bus_read(emu->cpu.pc + 2),
        (u32)emu->cpu.a,
        flags,
        (u32)emu->cpu.bc(),
        (u32)emu->cpu.de(),
        (u32)emu->cpu.hl(),
        (u32)emu->cpu.pc,
        (u32)emu->cpu.sp
    );
    g_app->debug_window.cpu_log.append(buf);
}
```

可以看到我们在一行日志中记录了CPU各个寄存器的值，以及PC寄存器指向的地址处三个字节的数
据，以便我们检查接下来需要执行的指令，然后将日志行添加到DebugWindow的cpu_log字
符串中。最后，我们在CPU::step中调用log函数，完成日志的记录：

```
void CPU::step(Emulator* emu)
{
    if(!halted)
    {
        // Handle interruptions.
        if(interrupt_master_enabled && (emu->int_flags & emu->int_enable_flags))
        {
            service_interrupt(emu);
        }
        else
        {
            if(g_app->debug_window.cpu_logging)
            {
                log(emu);
            }
            // fetch opcode.
            u8 opcode = emu->bus_read(pc);
            /*...*/
        }
    }
    /*...*/
}
```

串口信息

串口信息输出面板会以ASCII的形式显示GameBoy程序通过串口输出的数据，这主要用于在后面的CPU单元测试中查看程序输出的信息。我们首先为DebugWindow添加相应的变量和函数：

```
// Serial inspector.
Vector<u8> serial_data;
void serial_gui();
```

接着在DebugWindow.cpp中实现并接入函数：

```
void DebugWindow::gui()
{
    if(ImGui::Begin("Debug Window", &show))
    {
        cpu_gui();
        serial_gui();
    }
    ImGui::End();
}
void DebugWindow::serial_gui()
{
    if(g_app->emulator)
    {
        // Read serial data.
        while(!g_app->emulator->serial.output_buffer.empty())
        {
            u8 data = g_app->emulator->serial.output_buffer.front();
            g_app->emulator->serial.output_buffer.pop_front();
            serial_data.push_back(data);
        }
    }
    if(ImGui::CollapsingHeader("Serial data"))
    {
        ImGui::Text("Serial Data:");
```

```
        if(ImGui::BeginChild("Serial Data", ImVec2(500.0f, 100.0f), ImGuiChildFlagsNone))
        {
            ImGui::TextUnformatted((c8*)serial_data.begin(), (c8*)serial_data.end());
        }
        ImGui::EndChild();
        if(ImGui::Button("Clear"))
        {
            serial_data.clear();
        }
    }
}
```

DebugWindow::serial_gui会在调用时读取模拟器Serial::output_buffer中的程序输出数据，并将它们添加到DebugWindow的serial_data中。在显示时，该函数会绘制一个固定高度的子窗口，并在窗口中展示从总线读取的数据。

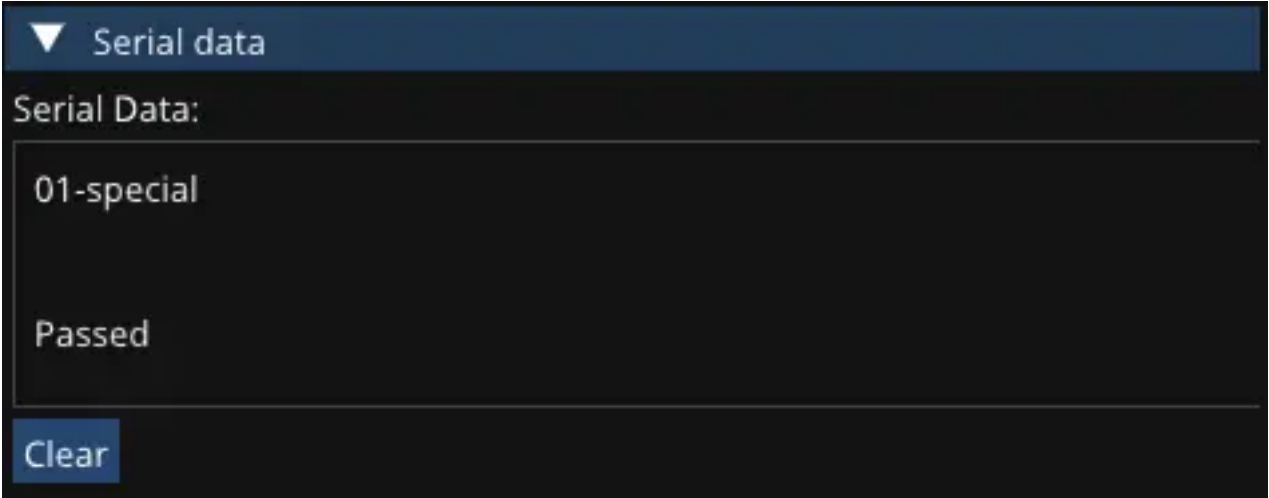
CPU单元测试

在完成了上述所有工作以后，我们就可以开始加载用于CPU单元测试的ROM文件了。本专题使用的CPU单元测试ROM文件由Blargg制作，并在GameBoy模拟器开发者中广泛使用。ROM文件最初上传的网站已经失效，读者可以在此处下载到这些文件的拷贝：

https://github.com/retrio/gb-test-roms

github.com/retrio/gb-test-roms

在下载这些ROM文件后，使用我们的模拟器打开cpu_instrs/individual目录下的文件，例如01-special.gb。在调试面板中，我们可以看到测试ROM文件输出了如下信息：



如果看到Passed字样，则说明模拟器通过了该单元测试。而如果单元测试没有通过，则输出信息中会包含具体失败的CPU指令操作码，以及Failed字样。需要注意的是，由于某些指令也用于运行单元测试自身的框架代码，因此单元测试在失败时的输出信息可能不一定准确。如果您确认单元测试输出的错误指令并没有问题，可以使用上文中提到的日志输出功能保存虚拟机CPU的运行日志，并与参考实现的日志进行比较，以定位问题。参考实现中的代码应当能够通过cpu_instrs/individual目录下的所有单元测试。

以上就是本章节的全部内容了。本章节的内容比较细碎，算是对之前几章实现CPU时一直忽略的一些方面进行了查漏补缺，希望读者能够在进入接下来的章节之前先使用单元测试ROM文件和调试功能确保CPU的所有指令和功能正确实现。从下一章开始，我们将进入像素处理单元（PPU）的学习和编程，这也是整个GameBoy模拟器之旅中最有意思的部分。那么，下一章见！

銀葉吉祥：从零开始实现GameBoy模拟器 #7 PPU

37 赞同 · 0 评论 [文章](#)

编辑于 2024-02-09 14:58 · IP 属地浙江

串口通信

游戏机模拟器

Game Boy（GB）



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

安卓模拟器运行卡顿原因七大解决方法

模拟器VT设置教程帮助很多大佬解决了在使用模拟器玩游戏时的卡顿和闪退等问题。但是并不是所有的卡顿和闪退都能把原因丢到电脑是否开启VT这个事情上的，比如目前发现装有360安全卫士的电脑...

雷电模拟器

安卓模拟器换IP

随着手机的普及越来越多的手机工作室诞生，也促进了模拟器发展电脑端玩手游，挂机，多开等等，减少了手机，话费，电费一系列成本，只需要一台电脑、一个模拟器（网上可以免费下载：建议雷电...

770785839



安卓逆向之模拟器检测

甜甜



【NAS娱乐回本】重返童年之使用Docker搭建FC-web游戏

Stark-C

赞同 20



添加评论

分享

喜欢

收藏

申请转载

