从零开始实现GameBoy模拟器 #5 CB指令、特殊指令、中断

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模拟器第五章。本章将接着上一章的内容，完成CPU剩下的CB指令和特殊指令。然后，我们将会讲解CPU中断机制的基本原理，并在GameBoy中实现CPU中断。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-05，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

CB指令

CB指令并不是单一的一条指令，而是一系列算术指令的总称。由于GameBoy的处理器使用一个字节表示操作码，因此其最多只能表示256个操作码（实际上只使用了244个），无法容纳处理器所需的所有指令，因此部分指令的操作码使用双字节表示，并且其第一个字节的值固定为0xCB。当处理器读取到0xCB操作码时，其会读取紧跟在0xCB之后的一个字节的数据，并根据第二个字节的值来决定真正要执行的操作。

第二个字节的值的**高五位**表示需要进行的操作，这些操作与对应的值如下所示：

操作码	指令	含义
00000b	RLC	将值向左旋转一位。
00001b	RRC	将值向右旋转一位。
00010b	RL	将值与C标志位一起向左旋转一位。
00011b	RR	将值与C标志位一起向右旋转一位。
00100b	SLA	将值左移一位，最低值置0。
00101b	SRA	将值右移一位，保留最高位的值。
00110b	SWAP	将低4位的值与高4位交换。
00111b	SRL	将值右移一位，最高值置0。
01000b	BIT 0	根据第0位的值设置Z标志位。
01001b	BIT 1	根据第1位的值设置Z标志位。
01010b	BIT 2	根据第2位的值设置Z标志位。
01011b	BIT 3	根据第3位的值设置Z标志位。
01100b	BIT 4	根据第4位的值设置Z标志位。

01101b	BIT 5	根据第5位的值设置Z标志位。
01110b	BIT 6	根据第6位的值设置Z标志位。
01111b	BIT 7	根据第7位的值设置Z标志位。
10000b	RES 0	将第0位的值置0。
10001b	RES 1	将第1位的值置0。
10010b	RES 2	将第2位的值置0。
10011b	RES 3	将第3位的值置0。
10100b	RES 4	将第4位的值置0。
10101b	RES 5	将第5位的值置0。
10110b	RES 6	将第6位的值置0。
10111b	RES 7	将第7位的值置0。
11000b	SET 0	将第0位的值置1。
11001b	SET 1	将第1位的值置1。
11010b	SET 2	将第2位的值置1。
11011b	SET 3	将第3位的值置1。
11100b	SET 4	将第4位的值置1。
11101b	SET 5	将第5位的值置1。
11110b	SET 6	将第6位的值置1。
11111b	SET 7	将第7位的值置1。

第二个字节的值的**低三位**表示操作的目标，这些目标与对应的值如下所示：

操作码	目标
000b	B寄存器
001b	C寄存器
010b	D寄存器
011b	E寄存器
100b	H寄存器
101b	L寄存器
110b	HL寄存器所存储的地址
111b	A寄存器

所有的CB指令都只针对单个目标进行操作，因此在实现CB指令时，我们可以在一个函数中实现所有指令，代码如下：

```
//! PREFIX CB : Invokes CB instructions.
void xcb_prefix_cb(Emulator* emu)
{
    u8 op = read_d8(emu);
    emu->tick(1);
    u8 data_bits = op & 0x07;
    u8 data;
    // Load data.
    switch(data_bits)
    {
        case 0: data = emu->cpu.b; break;
        case 1: data = emu->cpu.c; break;
        case 2: data = emu->cpu.d; break;
        case 3: data = emu->cpu.e; break;
        case 4: data = emu->cpu.h; break;
        case 5: data = emu->cpu.l; break;
        case 6: data = emu->bus_read(emu->cpu.hl()); emu->tick(1); break;
        case 7: data = emu->cpu.a; break;
        default: lupanic(); break;
    }
    // Modify data.
    u8 op_bits = (op & 0xF8) >> 3;
    if(op_bits == 0) rlc_8(emu, data);
    else if(op_bits == 1) rrc_8(emu, data);
    else if(op_bits == 2) rl_8(emu, data);
    else if(op_bits == 3) rr_8(emu, data);
    else if(op_bits == 4) sla_8(emu, data);
    else if(op_bits == 5) sra_8(emu, data);
    else if(op_bits == 6) swap_8(emu, data);
    else if(op_bits == 7) srl_8(emu, data);
    else if(op_bits <= 0x0F) bit_8(emu, data, op_bits - 0x08);
    else if(op_bits <= 0x17) res_8(emu, data, op_bits - 0x10);
    else if(op_bits <= 0x1F) set_8(emu, data, op_bits - 0x18);
    else lupanic();
```

```
    // Store data if op is not BIT (which does not modify data).
    if(op_bits <= 0x07 || op_bits >= 0x10)
    {
        switch(data_bits)
        {
            case 0: emu->cpu.b = data; break;
            case 1: emu->cpu.c = data; break;
            case 2: emu->cpu.d = data; break;
            case 3: emu->cpu.e = data; break;
            case 4: emu->cpu.h = data; break;
            case 5: emu->cpu.l = data; break;
            case 6: emu->bus_write(emu->cpu.hl(), data); emu->tick(1); break;
            case 7: emu->cpu.a = data; break;
            default: lupanic(); break;
        }
    }
    emu->tick(1);
}
```

首先，我们通过read_d8读取了CB指令的实际操作码，然后分别使用data_bits和op_bits分别提取了操作码的低3位和高5位，以确定我们要执行的操作和数据位置。接着，我们根据data_bits的值从不同的位置中读取数据，然后根据op_bits的值对数据进行操作，最后根据data_bits的值将操作后的数据写回到不同的位置中。

CB指令的时间开销根据指令的不同而定，我们可以在前几章提到的指令表中找到每一个指令消耗的时间。从原理上来说，我们也可以按照以下逻辑理解每个指令的开销：

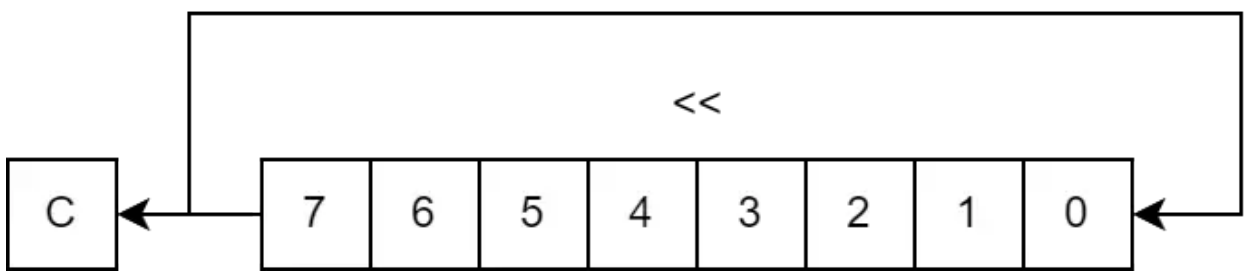
1. 由于CB指令需要在读取0xCB操作码以后额外读取一个字节的操作码，因此有1个机器周期的开销。
2. 如果指令需要读取HL寄存器指向的总线的数据，则需要1个机器周期的额外开销。所有data_bits为001b的指令都有该开销
3. 如果指令需要将数据写入HL寄存器指向的总线，则需要1个机器周期的额外开销。所有data_bits为001b的非BIT指令都有该开销，而BIT指令因为不会修改原先值，因此不需要将数据写回总线，也就没有该开销。
4. 运算操作本身具有1个机器周期的固定开销。

因此一个CB指令的开销在2个机器周期至4个机器周期不等，从上面的代码中我们也可以看出每一个机器周期开销对应的操作。

接下来，我们将逐一了解和实现CB指令中使用到的各个实际操作函数。

RLC指令

RLC（rotate left with carry）指令将8位数据整体左移一位，并将原本最高位的数据设置给移动后的最低位以及C标志位，如图所示：

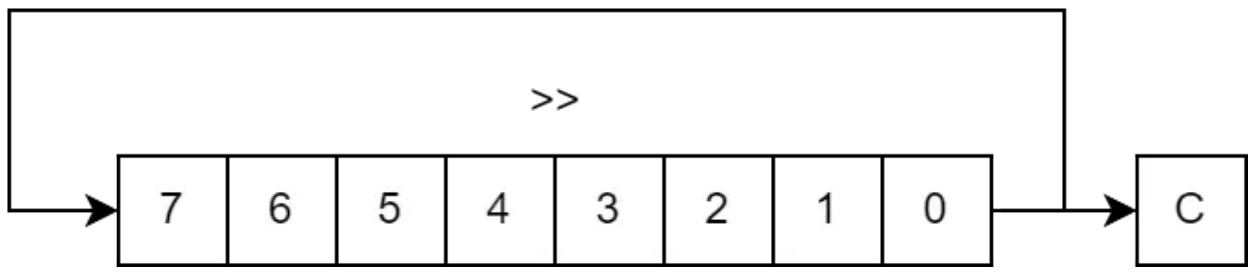


同时，RLC指令根据运算后的值设置Z标志位，并将N和H标志位置0。代码实现如下：

```
inline void rlc_8(Emulator* emu, u8& v)
{
    bool carry = (v & 0x80) != 0;
    v <<= 1;
    if(carry)
    {
        v |= 0x01;
        emu->cpu.set_fc();
    }
    else
    {
        emu->cpu.reset_fc();
    }
    set_zero_flag(emu, v);
    emu->cpu.reset_fn();
    emu->cpu.reset_fh();
}
```

RRC指令

RRC（rotate right with carry）指令将8位数据整体右移一位，并将原本最低位的数据设置给移动后的最高位以及C标志位，如图所示：

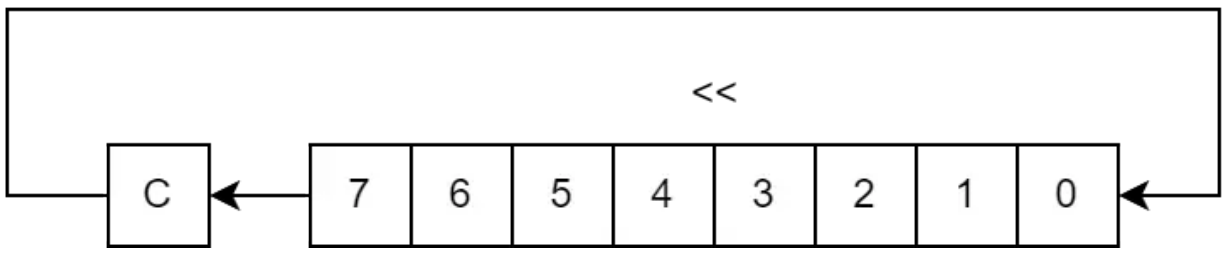


同时，RRC指令根据运算后的值设置Z标志位，并将N和H标志位置0。代码实现如下：

```
inline void rrc_8(Emulator* emu, u8& v)
{
    bool carry = (v & 0x01) != 0;
    v >>= 1;
    if(carry)
    {
        v |= 0x80;
        emu->cpu.set_fc();
    }
    else
    {
        emu->cpu.reset_fc();
    }
    set_zero_flag(emu, v);
    emu->cpu.reset_fn();
    emu->cpu.reset_fh();
}
```

RL指令

RL（rotate left）指令将8位数据整体左移一位，将原本最高位的数据设置给C标志位，并将C标志位原本的值设置给最低位，如图所示：

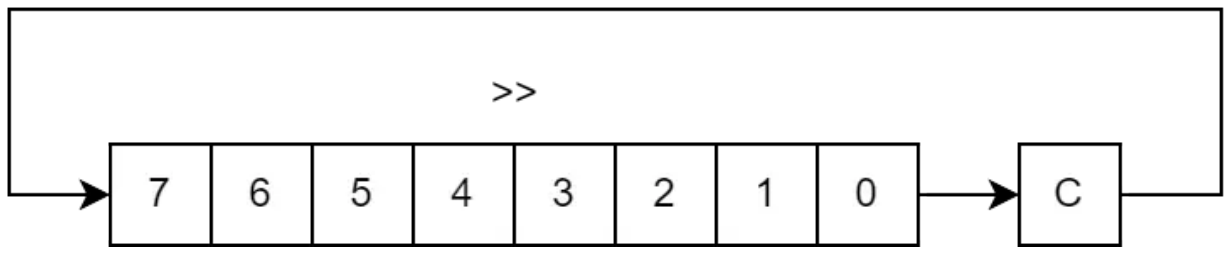


同时，RL指令根据运算后的值设置Z标志位，并将N和H标志位置0。代码实现如下：

```
inline void rl_8(Emulator* emu, u8& v)
{
    bool carry = (v & 0x80) != 0;
    v <<= 1;
    if(emu->cpu.fc())
    {
        v |= 0x01;
    }
    if(carry)
    {
        emu->cpu.set_fc();
    }
    else
    {
        emu->cpu.reset_fc();
    }
    set_zero_flag(emu, v);
    emu->cpu.reset_fn();
    emu->cpu.reset_fh();
}
```

RR指令

RR（rotate right）指令将8位数据整体右移一位，将原本最低位的数据设置给C标志位，并将C标志位原本的值设置给最高位，如图所示：

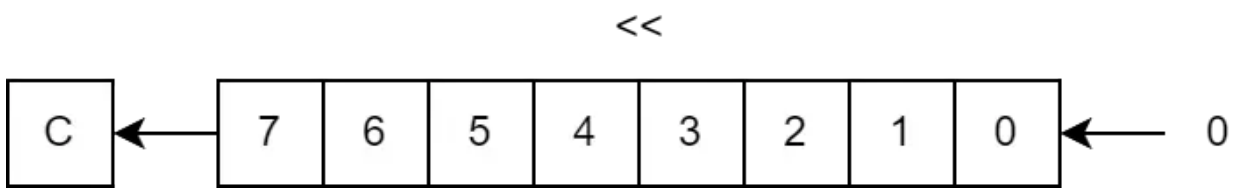


同时，RR指令根据运算后的值设置Z标志位，并将N和H标志位置0。代码实现如下：

```
inline void rr_8(Emulator* emu, u8& v)
{
    bool carry = (v & 0x01) != 0;
    v >>= 1;
    if(emu->cpu.fc())
    {
        v |= 0x80;
    }
    if(carry)
    {
        emu->cpu.set_fc();
    }
    else
    {
        emu->cpu.reset_fc();
    }
    set_zero_flag(emu, v);
    emu->cpu.reset_fn();
    emu->cpu.reset_fh();
}
```

SLA指令

SLA（shift left）指令将8位数据整体左移一位，将原本最高位的数据设置给C标志位，并将最低位设置为0，如图所示：

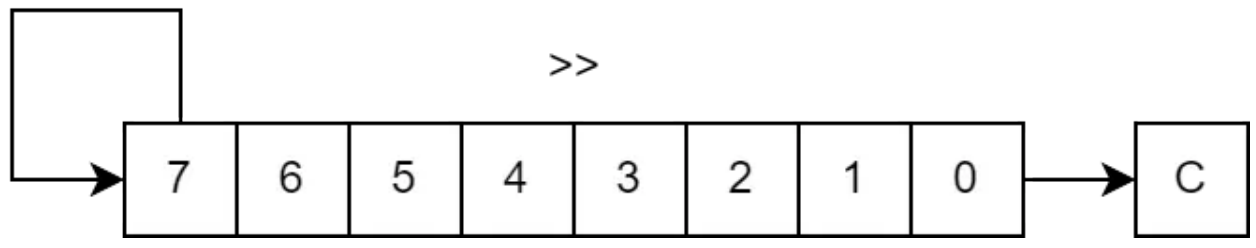


同时，SLA指令根据运算后的值设置Z标志位，并将N和H标志位置0。代码实现如下：

```
inline void sla_8(Emulator* emu, u8& v)
{
    bool carry = (v & 0x80) != 0;
    v <<= 1;
    if(carry)
    {
        emu->cpu.set_fc();
    }
    else
    {
        emu->cpu.reset_fc();
    }
    set_zero_flag(emu, v);
    emu->cpu.reset_fn();
    emu->cpu.reset_fh();
}
```

SRA指令

SRA（shift right）指令将8位数据整体右移一位，将原本最低位的数据设置给C标志位，并**保留**最高位的值，如图所示：



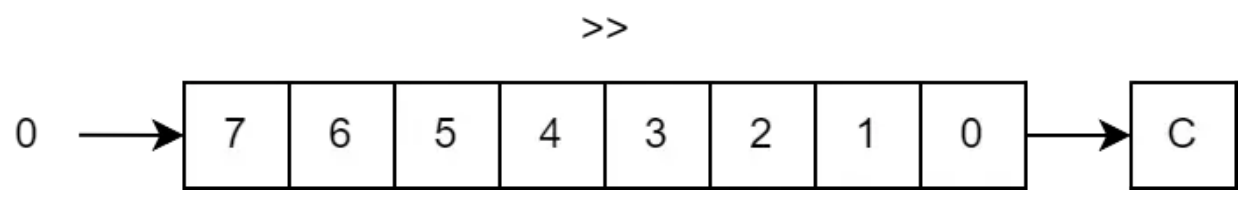
同时，SRA指令根据运算后的值设置Z标志位，并将N和H标志位置0。代码实现如下：

```
inline void sra_8(Emulator* emu, u8& v)
```

```
{
    bool carry = (v & 0x01) != 0;
    v = (v & 0x80) | ((v >> 1) & 0x7F);
    if(carry)
    {
        emu->cpu.set_fc();
    }
    else
    {
        emu->cpu.reset_fc();
    }
    set_zero_flag(emu, v);
    emu->cpu.reset_fn();
    emu->cpu.reset_fh();
}
```

SRL指令

SRL（shift right）指令将指令将8位数据整体右移一位，将原本最低位的数据设置给C标志位，并将最高位设置为0，如图所示：

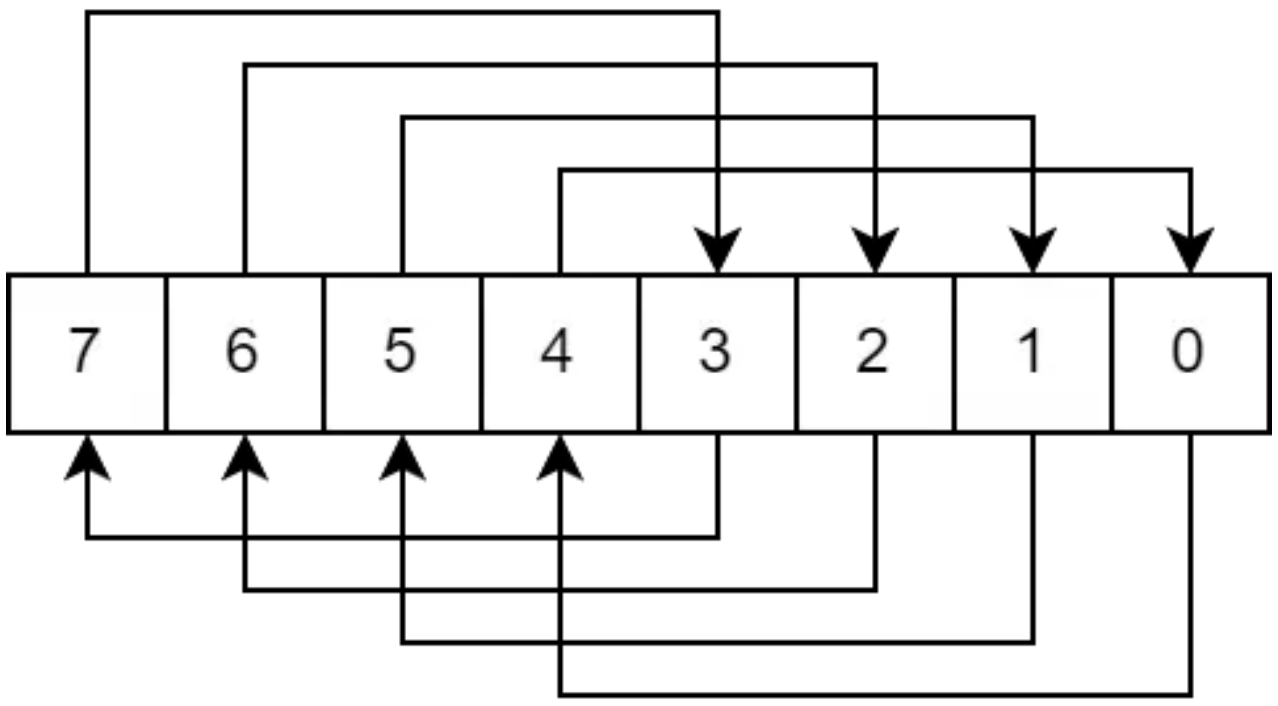


同时，SRL指令根据运算后的值设置Z标志位，并将N和H标志位置0。代码实现如下：

```
inline void srl_8(Emulator* emu, u8& v)
{
    bool carry = (v & 0x01) != 0;
    v = (v >> 1) & 0x7F;
    if(carry)
    {
        emu->cpu.set_fc();
    }
    else
    {
        emu->cpu.reset_fc();
    }
    set_zero_flag(emu, v);
    emu->cpu.reset_fn();
    emu->cpu.reset_fh();
}
```

SWAP指令

SWAP指令交换值低四位和高四位的数据，如图所示：



同时，SWAP指令根据运算后的值设置Z标志位，并将N、H和C标志位置0。代码实现如下：

```
inline void swap_8(Emulator* emu, u8& v)
{
    v = ((v >> 4) & 0x0F) + ((v << 4) & 0xF0);
    set_zero_flag(emu, v);
    emu->cpu.reset_fc();
```

```
    emu->cpu.reset_fn();
    emu->cpu.reset_fh();
}
```

BIT指令

BIT（bit test）指令用于检查值对应的位是否为1，如果值为1，则设置CPU标志位Z的值为1，否则设置为0。同时，BIT指令会将N标志位置0，H标志位置1，代码实现如下：

```
inline void bit_8(Emulator* emu, u8 v, u8 bit)
{
    if(bit_test(&v, bit))
    {
        emu->cpu.reset_fz();
    }
    else
    {
        emu->cpu.set_fz();
    }
    emu->cpu.reset_fn();
    emu->cpu.set_fh();
}
```

RES指令

RES（reset）指令用于将值对应的位设置为0，不对CPU标志位做任何修改，代码实现如下：

```
inline void res_8(Emulator* emu, u8& v, u8 bit)
{
    bit_reset(&v, bit);
}
```

SET指令

SET指令用于将值对应的位设置为1，不对CPU标志位做任何修改，代码实现如下：

```
inline void set_8(Emulator* emu, u8& v, u8 bit)
{
    bit_set(&v, bit);
}
```

RLCA指令

0x07 RLCA指令与**0xCB07 RLC A**指令行为一致，除了其会将Z标志位固定设置为0，而不是根据结果来设置，并且只花费一个机器周期。

```
//! RLCA : Rotates A left.
void x07_rlca(Emulator* emu)
{
    rlc_8(emu, emu->cpu.a);
    emu->cpu.reset_fz();
    emu->tick(1);
}
```

RRCA指令

0x0F RRCA指令与**0xCB0F RRC A**指令行为一致，除了其会将Z标志位固定设置为0，而不是根据结果来设置，并且只花费一个机器周期。

```
//! RRCA : Rotates A right.
void x0f_rrca(Emulator* emu)
{
    rrc_8(emu, emu->cpu.a);
    emu->cpu.reset_fz();
    emu->tick(1);
}
```

RLA指令

0x17 RLA指令与**0xCB17 RL A**指令行为一致，除了其会将Z标志位固定设置为0，而不是根据结果来设置，并且只花费一个机器周期。

```
//! RLA : Rotates A left through carry.
void x17_rla(Emulator* emu)
{
    rl_8(emu, emu->cpu.a);
    emu->cpu.reset_fz();
    emu->tick(1);
}
```

RRA指令

0x1F RRA指令与**0xCB1F RR A**指令行为一致，除了其会将Z标志位固定设置为0，而不是根据结果来设置，并且只花费一个机器周期。

```
//! RRA : Rotates A right through carry.
void x1f_rra(Emulator* emu)
{
    rr_8(emu, emu->cpu.a);
    emu->cpu.reset_fz();
    emu->tick(1);
}
```

HALT指令

0x76 HALT指令会使得CPU进入停止状态，但是保持系统时钟以及所有外围电路（包括PPU、APU等）的运行。进入停止状态的CPU不会继续指令任何指令，直到外部中断恢复CPU的运行为止。由于我们已经在CPU::step中判断了halt状态，因此在实现该指令时，我们只需要简单讲halt变量设置为true就行：

```
//! HALT : Pauses the CPU.
void x76_halt(Emulator* emu)
{
    emu->cpu.halted = true;
    emu->tick(1);
}
```

STOP指令

0x1000 STOP指令会停止CPU的运行。与HALT不同的是，STOP指令会停止系统时钟，停止所有外围电路的运行，同时也会关闭所有中断和LCD控制器，因此STOP指令执行后CPU无法通过中断恢复运行，只能通过RESET按键重启机器来恢复CPU运行，并按照RESET的逻辑重启整个系统。在实际表现上，STOP指令类似于直接关闭GameBoy，但是GameBoy仍然处于通电状态，因此内存和卡带RAM中的数据在RESET机器以后仍然会保留。

STOP指令具有两个字节长度，第一个字节为0x10，第二个字节为0x00，但是由于0x10操作码没有任何别的作用，因此我们可以认为STOP指令的操作码是0x10，并且直接读取并丢弃第二个字节。至于STOP的操作，由于我们不会真的关闭GameBoy，所以我们会将模拟器的paused变量设置为true，从而暂停模拟器的运行。

```
//! STOP : Stops CPU.
void x10_stop(Emulator* emu)
{
    read_d8(emu); // always 0x00.
    emu->paused = true;
    emu->tick(1);
}
```

由于STOP指令会暂停时钟且无法恢复，因此在大部分文档中，其并没有一个明确的时间开销。我们在此使用默认的1个机器周期作为STOP指令的时间开销。

中断

在实现了上述所有指令后，我们的指令函数表中只剩下两个指令没有实现了：**0xF3 DI**和**0xFB EI**。这两个指令的作用是禁用和启动中断。在实现这两个指令之前，我们需要了解中断的基础概念，以及中断在模拟器中的具体实现。

首先让我们了解一下GameBoy的CPU，或者现代的大部分处理器提供中断机制的背景。当我们在

执行一段程序时，GameBoy内部实际上还有别的硬件电路在并行工作。例如，像素处理单元（PPU）可能在读取显存的数据，并将它们转换成像素显示在屏幕上；计时器在跟着时钟信号更新自己的时间；串口控制芯片在接收另一台设备的数据，并将自己的数据发送给对方。在大部分情况下，这些硬件电路都互不干扰的可以独自工作，但是在某些时候，CPU需要与这些硬件进行**同步**，以确保仅在某些条件符合的时候才会执行接下来的操作，例如：

1. CPU需要在PPU完成一帧图片的绘制后才能更新显示屏的控制寄存器以及显示内存中的数据，不然会导致数据无法被正确写入显存中，甚至可能会损坏硬件电路。
2. 当串口完成一个字节数据的处理后，CPU需要读取接收到的数据，同时向串口写入下一个需要发送的字节，以保证串口通信能够连续工作。
3. 当计时器的时间到达一定数值时，CPU需要执行某些特定的操作，例如实现游戏中的倒计时功能。

而在程序中，如果要实现这种同步，我们一般会使用两种设计模式：轮询（polling）和事件（event）。在轮询模式下，程序会每隔一段时间检查一次硬件电路的工作状态，以检测其状态是否满足条件，并在条件满足的情况下执行特定的操作；而在事件模式下，程序并不需要主动去查询硬件的状态，其只需要标记那些当前感兴趣的事件（即硬件电路的状态改变，且改变后的状态满足特定条件），当这些事件发生的时候，硬件会主动通知程序去处理这些事件。

轮询模式不需要额外的硬件支持，在GameBoy中，程序正是通过每隔一段时间检查一次硬件电路的状态来读取用户是否按下了游戏机的操作按钮的。然而，在大多数情况下，轮询模式具有以下一些问题：

1. 性能问题：轮询模式需要CPU不停地查询硬件状态，哪怕大部分情况下硬件并没有进入程序预期的状态，这相当于CPU一直在和硬件进行同步，会导致CPU一直在执行重复且无效的指令，白白浪费GameBoy本就不多的处理器性能，并且也会导致游戏耗电量的增加。
2. 时效问题：轮询模式下，CPU处理硬件事件的时间点取决于程序下一次读取硬件状态的时间，而并非该事件发生的时间。如果事件发生时CPU正在执行一些别的任务，就可能会导致事件发生了很久以后CPU才去处理事件。对于GameBoy这类对实时性要求很高的系统来说，一个事件如果在几十个时钟周期中得不到处理，就可能产生严重的问题（例如串口接收到的数据如果没有及时读取，就会被下一次通信的数据覆盖，从而导致数据错误）。

上述两个问题通过事件机制就可以得到解决，而中断正是处理器在硬件层面实现的事件机制。中断通常通过一个特定的CPU引脚与外部电路连接，当外部电路发生某些特定事件时，其可以将该引脚的电平置为高电平（或者低电平）来通知CPU产生中断，而如果程序又通过寄存器指定了监听该中断事件，则CPU将会执行一段特殊的中断服务程序（interrupt service routine）来响应中断。对于GameBoy来说，该程序执行以下操作：

1. 禁用中断，相当于调用DI指令。
2. 将中断标志位寄存器（IF）中的对应中断标志位重置为0。
3. 等待两个机器周期，相当于调用两次NOP指令。
4. 将PC寄存器的值入栈，然后将PC寄存器设置为中断处理函数的入口地址，相当于调用一次CALL a16指令。

整个中断处理流程累计花费5个机器周期。

中断处理函数是一段游戏自定义的函数，其用于处理对应的中断，然后调用RET或者RETI指令返回到正常程序继续执行，即完成一次中断响应。每一个中断类型只能有一个中断处理函数，其入口地址为固定值，如下表所示：

中断标识符	中断条件	中断处理程序入口地址
VBLANK	PPU完成了一帧画面的绘制	0x40
LCD_STAT	LCD运行状态改变	0x48
TIMER	计时器时间到	0x50
SERIAL	串口传输了一个字节的数据	0x58
JOYPAD	选择的四个按钮中任意按钮按下	0x60

在GameBoy中，中断的行为主要由以下几个寄存器控制：

0xFFFF——中断激活（IE）

IE（interrupt enable）寄存器是8位可读写寄存器，通过总线地址0xFFFF访问。该寄存器的低5位表示五种类型的中断是否可以被CPU所检测到，它们的映射关系分别是：

7	6	5	4	3	2	1	0
-	-	-	JOYPAD	SERIAL	TIMER	LCD_STAT	VBLANK

将对应位设置为0，则CPU不会处理该类型的中断，也不会因为该类型的中断被唤醒。

0xFF0F——中断标志位（IF）

IF（interrupt flags）寄存器是8位可读写寄存器，通过总线地址0xFF0F访问。该寄存器的低5位表示是否有对应的中断正在等待被处理，如果其值为1，则代表有一个该类型的中断正在等待被处理。IF寄存器上不同类型中断的映射关系与IE寄存器一致，我们可以使用IE & IF != 0来判断当前是否有激活且需要被处理的中断请求。

由于IF寄存器是可读写寄存器，CPU也可以通过将1写入IF寄存器的指定位来主动触发中断，其效果与外部硬件触发的中断一致。

中断总控（IME）

中断总控（interrupt master enable，IME）是一个内置在CPU中的标志位寄存器（只有1个bit的寄存器）。该寄存器只可写不可读，其值通过DI（disable interrupt）指令设置为0，通过EI（enable interrupt）指令设置为1。当IME的值为1时，程序允许被激活且被等待被处理器的中断触发中断服务程序来打断程序的运行，以跳转到中断处理程序。而当IME的值为0时，等待被处理的中断会被保存在IF寄存器中，但是不会触发中断服务程序，直到中断总控被重新打开。在IME为0时，如果程序手动将0写入IF寄存器的对应标志位，就可以丢弃对应的中断事件。

DI指令会立即生效，而EI指令的生效需要推迟一个指令，因此当EI指令的下一条指令执行时，如果中断总控之前处于关闭状态，则仍然会保持关闭，直到这条指令执行完毕后才会打开。该行为导致如果在EI指令之后立即调用DI指令，则EI的操作会被撤销，中断总控会保持关闭状态，没有任何中断会被处理。

需要注意的是，在CPU使用HALT指令停止之后，只要在任一时刻IE & IF != 0，则CPU都会被唤醒，即使IME处在关闭状态。然而，中断服务程序只有当IME打开后才会运行，因此如果CPU因为IE & IF != 0而唤醒，但是IME处在关闭状态，则CPU会继续执行HALT指令之后的代码，而不会处理中断。

实现中断

在了解了中断的原理后，就让我们开始在模拟器中实现中断机制吧！我们首先实现CPU的IME寄存器以及控制指令。在CPU类中添加以下变量：

```
//! Interrupt master enable flag.
bool interrupt_master_enabled;
//! Interrupt master enableing countdown.
u8 interrupt_master_enabling_countdown;
```

IME在CPU启动时默认处于关闭状态，因此我们在CPU::init中添加这两个变量的初始化代码：

```
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;
    interrupt_master_enabled = false;
    interrupt_master_enabling_countdown = 0;
}
```

interrupt_master_enabled存储了当前IME寄存器的状态，interrupt_master_enabling_countdown用于记录在多少条指令之后激活中断总线，以实现延迟启动的功能。然后我们在CPU类中添加两个函数用于开启和关闭中断总控：

```
void enable_interrupt_master()
{
    interrupt_master_enabling_countdown = 2;
}
void disable_interrupt_master()
{
    interrupt_master_enabled = false;
    interrupt_master_enabling_countdown = 0;
}
```

在调用enable_interrupt_master时，我们将interrupt_master_enabling_countdown设置为2，意味着我们需要等待两个指令（包括EI本身）完成以后启动中断总控；在调用disable_interrupt_master时，我们可以立即将中断关闭，同时将interrupt_master_enabling_countdown清零以撤销任何还未生效的激活中断总控操作。有了这两个函数后，实现DI和EI指令只需要简单调用这两个函数就行：

```
//! DI : Disable interrupts.
void xf3_di(Emulator* emu)
{
    emu->cpu.disable_interrupt_master();
    emu->tick(1);
}
//! EI : Enable interruption.
void xfb_ei(Emulator* emu)
{
    emu->cpu.enable_interrupt_master();
    emu->tick(1);
}
```

接着我们实现中断的触发和跳转流程。首先我们需要在Emulator类中添加两个8位的整数变量来代表IE和IF寄存器：

```
//! 0xFF0F - The interruption flags.
u8 int_flags;
//! 0xFFFF - The interruption enabling flags.
u8 int_enable_flags;
```

IE和IF寄存器初始值均为0x00，即屏蔽所有中断，因此我们在Emulator::init中初始化两个寄存器的值：

```
RV Emulator::init(const void* cartridge_data, usize cartridge_data_size)
{
    /*...*/
    memzero(wram, 8_kb);
    memzero(vram, 8_kb);
    int_flags = 0;
    int_enable_flags = 0;
    return ok;
}
```

在之后实现PPU等外围组件时，我们只需要将int_flags中对应的标志位置1，就可以向CPU申请中断。

然后我们修改CPU::step的代码，添加CPU检查和处理中断的逻辑：

```
void CPU::step(Emulator* emu)
{
    if(!halted)
    {
        // Handle interruptions.
        if(interrupt_master_enabled && (emu->int_flags & emu->int_enable_flags)
        {
            service_interrupt(emu);
        }
        else
        {
            // 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)opc
                emu->paused = true;
            }
            else
            {
                instruction(emu);
            }
        }
    }
    else
    {
        emu->tick(1);
        // Wake up CPU if any interruption is pending.
        // This happens even if IME is disabled (interruption_enabled == false)
        if(emu->int_flags & emu->int_enable_flags)
        {
            halted = false;
        }
```

```
    }
    // Enable interrupt master when countdown reaches 0.
    if(interrupt_master_enabling_countdown)
    {
        --interrupt_master_enabling_countdown;
        if(!interrupt_master_enabling_countdown)
        {
            interrupt_master_enabled = true;
        }
    }
}
```

中断的处理主要包括三部分：

1. 中断会打断正常CPU的执行流程，因此在每次CPU执行一条新的指令前，其都会检查当前是否需要处理中断（IME打开，且IF & IE != 0），如果当前有待处理的中断，则调用service_interrupt函数处理中断，否则才按照正常逻辑执行指令。
2. 当CPU被暂停（halted == true）时，如果IF & IE != 0，则CPU需要被唤醒继续执行指令。
3. 当interrupt_master_enabling_countdown不为0时，说明IME已经由EI指令申请激活，因此我们在每个指令周期将该计数器值减1，并在值等于0的时候激活IME。

最后我们需要实现的便是实际处理中断的service_interrupt函数了。首先我们先在Emulator.hpp中定义每个中断的标志位：

```
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;
```

然后在CPU类中添加该函数的声明：

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

最后在CPU.cpp中添加函数实现：

```
inline void push_16(Emulator* emu, u16 v)
{
    emu->cpu.sp -= 2;
    emu->bus_write(emu->cpu.sp + 1, (u8)((v >> 8) & 0xFF));
    emu->bus_write(emu->cpu.sp, (u8)(v & 0xFF));
}
void CPU::service_interrupt(Emulator* emu)
{
    u8 int_flags = emu->int_flags & emu->int_enable_flags;
    u8 service_int = 0;
    if(int_flags & INT_VBLANK) service_int = INT_VBLANK;
    else if(int_flags & INT_LCD_STAT) service_int = INT_LCD_STAT;
    else if(int_flags & INT_TIMER) service_int = INT_TIMER;
    else if(int_flags & INT_SERIAL) service_int = INT_SERIAL;
    else if(int_flags & INT_JOYPAD) service_int = INT_JOYPAD;
    emu->int_flags &= ~service_int;
    emu->cpu.disable_interrupt_master();
    emu->tick(2);
    push_16(emu, emu->cpu.pc);
    emu->tick(2);
    switch(service_int)
    {
        case INT_VBLANK: emu->cpu.pc = 0x40; break;
        case INT_LCD_STAT: emu->cpu.pc = 0x48; break;
        case INT_TIMER: emu->cpu.pc = 0x50; break;
        case INT_SERIAL: emu->cpu.pc = 0x58; break;
        case INT_JOYPAD: emu->cpu.pc = 0x60; break;
    }
    emu->tick(1);
}
```

CPU在一次中断处理中只能响应一个中断事件，如果在处理时发现有多个中断事件同时发生，则CPU会处理优先级最高的中断。中断的优先级按照其在IF寄存器中的位顺序从高到低排列，即VBLANK具有最高优先级，JOYPAD具有最低优先级，因此我们使用多个if-else语句块选择第一个需要被处理的中断，保存在service_int中。

在确定了需要处理的中断以后，我们需要清除IF中对应的中断标志位，作为该中断已经处理的响应，该操作通过emu->int_flags &=~service_int实现；接着我们需要关闭中断总控，以避免在处

理中断时被别的中断打断。然后我们需要跳转到中断处理程序入口地址，该跳转指令与正常的CALL d16并无二致：我们先将PC寄存器的值压入栈中，然后根据service_int中存储的中断类型跳转到对应的地址就行。

中断处理程序在执行的时候与正常的程序没有区别，因此在跳转以后，我们只需要按照正常逻辑继续让CPU执行指令即可。中断程序在执行完成后一般通过RETI指令返回之前的程序位置，其在返回的时候会同时启动中断总控，以允许程序被再一次中断。

以上就是本章节的全部内容了。自此我们已经实现了GameBoy CPU部分的所有功能，在下一章节中，我们将实现GameBoy的串口通信和计时器组件，并为模拟器添加CPU的单步调试和日志输出功能，方便读者检查自己CPU实现是否正确。那么，下一章见！

銀葉吉祥：从零开始实现GameBoy模拟器 #6 定时器、串口、调试面板
20 赞同 · 0 评论 文章

编辑于 2024-02-09 14:58 · IP 属地浙江

游戏机模拟器 中央处理器 (CPU)

欢迎参与讨论

2 条评论 默认 最新

小离 ···
已经四天了！快更新呀，要不然我会以为你弃坑了😭一定要更新下去啊
02-06 · IP 属地辽宁 回复 喜欢

銀葉吉祥 作者 ···
一般周二和周五更新
02-06 · IP 属地浙江 回复 喜欢

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

小白入门安卓（2）Genymotion模拟器下载安装运
前面已经对AS进行了基本的配置，现在觉得下一步应该是把模拟器先弄上去先。这里觉得AVD不好用，所以根据大佬的建议，装Genymotion模拟器。 1.4 Genymotion模拟器安装 跟着这个
享受生活 发表于项目

折腾树莓派3跑retropie玩模拟器的几点感受和吐槽
过年前买了块树莓派3打算玩模拟器用。 树莓派上可以选的主流模拟器前端有retropie和recalbox可以选择，底层都是emulation station + retroarch的样子，后者貌似对树莓派3的板载蓝牙驱动适配...
凤凰院胸针

RA替代计划（下）—PSV全能模拟器太难用？推荐几个替代
leon

为了找出最好用的安卓模拟器，我进行了一次众测
阿虚同学 发表于阿虚同学

▲ 赞同 17 ▼ 2 条评论 分享 喜欢 收藏 申请转载 ···