


从零开始实现GameBoy模拟器 #3 加载，比较和跳转指令

 銀葉吉祥 

浙江大学 软件工程硕士

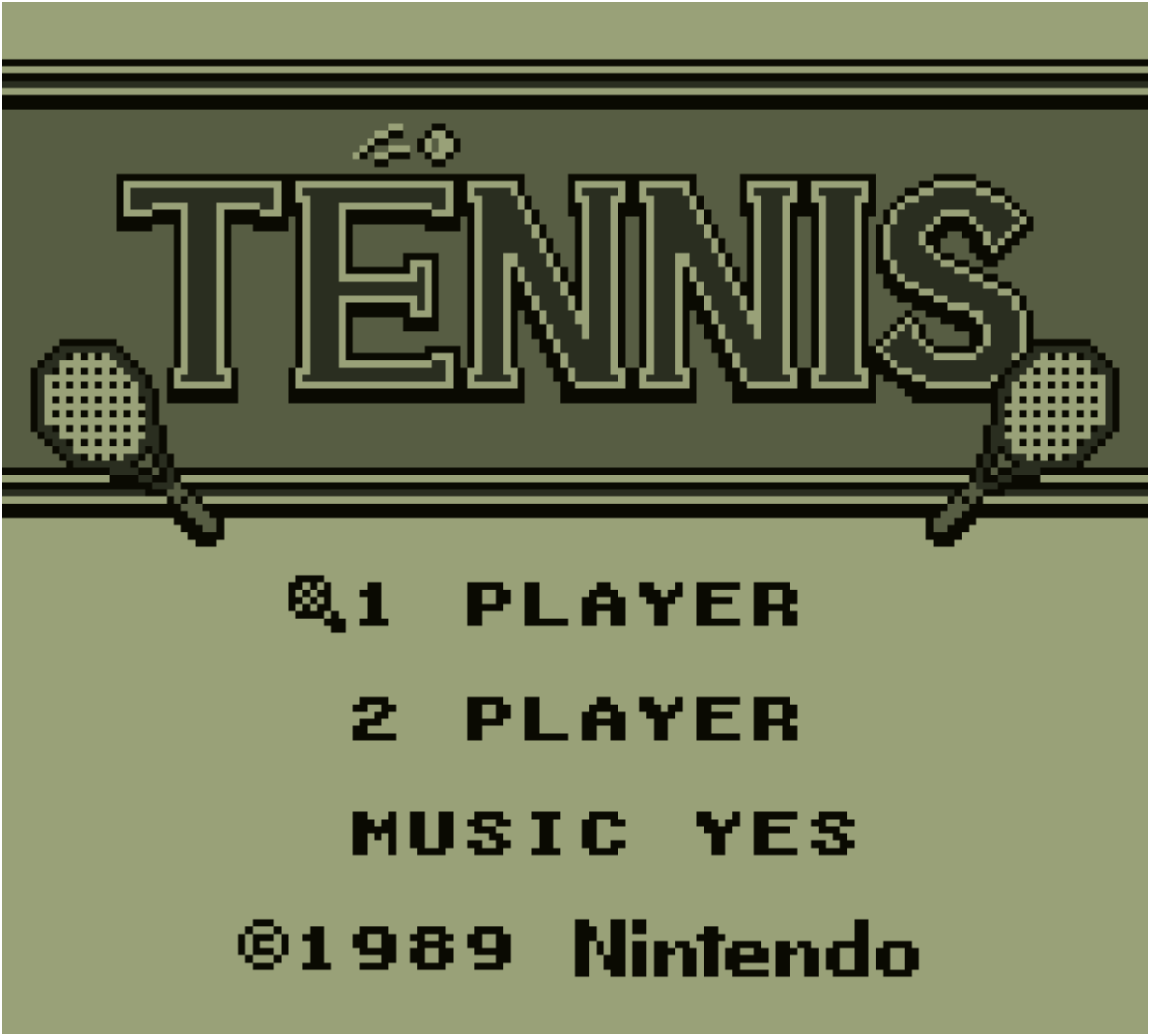
已关注

22 人赞同了该文章

目录

收起

- NOP指令
- LD指令
 - 8位寄存器LD指令
 - 8位总线读写LD指令
 - 8位立即数据LD指令
 - 16位LD指令
- CP指令
- JP指令
 - 0xC3 JP a16
 - 0xC2 JP NZ, a16
 - 0xCA JP Z, a16
 - 0xD2 JP NC, a16
 - 0xDA JP C, a16
 - 0xE9 JP (HL)
- JR指令
 - 无条件JR指令
 - 有条件JR指令



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。

2024.1.28更新：修复了原先代码总线读写时序不正确的问题。

欢迎来到从零开始实现GameBoy模拟器第三章。从本章开始，我们将开始实现CPU的各种指令。由于GameBoy的指令较多，在本章中，我们将首先实现CPU的赋值和流程控制指令，其余指令将在之后的章节中逐步实现。在实现指令时，读者可以参考下表查询GameBoy支持的所有指令：

Gameboy (LR35902) OPCODES

www.pastraiser.com/cpu/gameboy/gameboy_opcodes.ht

该表格中0xE2 LD (C), A和0xF2 LD A, (C)的数据有误，根据任天堂官方手册，其指令长度应为1，而非2。

该表格中0xCB的BIT (HL)系列指令的数据有误，根据任天堂官方手册，其指令的机器周期开销应为3，而非4。

并对照GameBoy编程文档来查阅每一个指令的详细介绍：

Gameboy Development Manual V1.1 - CPU Instruction Set

archive.org/details/GameBoyProgManVer1.1/page/n93/mo

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-03，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

NOP指令

首先让我们熟悉一下“指令”在我们的模拟器中的具体实现。我们以操作码为0x00的NOP指令为例，该指令只是简单地闲置CPU一个机器周期，不执行任何操作，因此可以作为我们实现其余指令的模板。在Instructions.cpp中添加该指令的实现：

```
#include "Instructions.hpp"
#include "Emulator.hpp"
```

[illegible]

我们添加了一个新的函数x00_nop表示我们的NOP指令，函数名称中的x00是指令以十六进制表示的操作码，后面的nop为指令名称。所有的指令都以Emulator*作为参数，因此其可以操作模拟器的所有数据。在函数实现中，由于NOP指令不执行任何操作，因此我们只是简单添加了一行注释，表示该指令不执行操作，而在别的指令中，我们将会把该注释替换成实际执行指令的代码。

然后，我们需要根据该指令的时间开销来步进模拟器的其余组件，以便这些组件能够与CPU的时序同步。通过查表可知，NOP指令的时间开销是1个机器周期（或4个时钟周期），因此我们调用emu->tick(1)来步进其余组件一个机器周期。

最后，我们将instructions_map中的空指针替换成实际的指令函数，就可以让模拟器调用我们的指令。此时编译并运行模拟器，可以看到模拟器提示的未实现指令从0x00变成了0xC3（对于绝大多数卡带），说明第一个指令0x00已经被成功加载并识别。通过查表可知，0xC3对应的指令是JP a16，即跳转到指定的16位地址，我们将在下文中实现该指令，以及其余的跳转指令。

当Instructions.cpp中的指令越来越多后，其文件会变得非常长，因此下文中的我们不会再给出该文件的完整代码，读者可以自行查阅项目仓库中的代码文件来查看完整代码。在向Instructions.cpp中添加指令时，笔者推荐按照操作码顺序整理这些指令代码，以方便后期维护和检查。

将指令函数添加到instructions_map的操作对每一个指令都是一样的，因此下文不再赘述。

LD指令

LD是load的缩写，表示“加载”。在实际的使用中，LD指令类似于编程语言中的赋值运算符，可以将一个8比特或者16比特的数据从一个位置拷贝到另一个位置。此处的“位置”可以是寄存器，也可以是一个指定的总线地址，因此LD指令能够实现总线的读写操作。

GameBoy总共提供了85种不同的LD指令，供程序在不同的寄存器，以及寄存器和总线地址间拷贝数据，可查表来了解每一个指令的操作码和详细描述。以下我们将分类讨论和实现。

8位寄存器LD指令

这类指令将一个8比特寄存器的数据拷贝至另一个8比特寄存器，例如0x41 LD B, C指令就是将寄存器C的值拷贝给寄存器B，代码如下：

```

//!! LD B, C : Loads C to B.
void x41_ld_b_c(Emulator* emu)
{
    emu->cpu.b = emu->cpu.c;
    emu->tick(1);
}

```

下标列出了该类型的所有指令，所有指令的机器周期均为1。所有的指令都可以按照0x41的实现代码，通过替换寄存器变量来实现，因此这里不一一举例。

操作码	指令	目标寄存器	源寄存器
0x40	LD B, B	B	B
0x41	LD B, C	B	C

0x42	LD B, D	B	D
0x43	LD B, E	B	E
0x44	LD B, H	B	H
0x45	LD B, L	B	L
0x47	LD B, A	B	A
0x48	LD C, B	C	B
0x49	LD C, C	C	C
0x4A	LD C, D	C	D
0x4B	LD C, E	C	E
0x4C	LD C, H	C	H
0x4D	LD C, L	C	L
0x4F	LD C, A	C	A
0x50	LD D, B	D	B
0x51	LD D, C	D	C
0x52	LD D, D	D	D
0x53	LD D, E	D	E
0x54	LD D, H	D	H
0x55	LD D, L	D	L
0x57	LD D, A	D	A
0x58	LD E, B	E	B
0x59	LD E, C	E	C
0x5A	LD E, D	E	D
0x5B	LD E, E	E	E
0x5C	LD E, H	E	H
0x5D	LD E, L	E	L
0x5F	LD E, A	E	A
0x60	LD H, B	H	B
0x61	LD H, C	H	C
0x62	LD H, D	H	D
0x63	LD H, E	H	E
0x64	LD H, H	H	H
0x65	LD H, L	H	L
0x67	LD H, A	H	A
0x68	LD L, B	L	B
0x69	LD L, C	L	C
0x6A	LD L, D	L	D
0x6B	LD L, E	L	E
0x6C	LD L, H	L	H
0x6D	LD L, L	L	L
0x6F	LD L, A	L	A
0x78	LD A, B	A	B
0x79	LD A, C	A	C
0x7A	LD A, D	A	D
0x7B	LD A, E	A	E
0x7C	LD A, H	A	H
0x7D	LD A, L	A	L
0x7F	LD A, A	A	A

从表中我们可以发现有一些指令实际上是冗余指令（例如LD B, B）， 这些指令只是为了硬件电路的正交而存在， 在实际的游戏中基本上不会用到。

8位总线读写LD指令

这类指令通过总线读取一个8比特的数据，并存储在寄存器中，或是将寄存器中的一个8比特的数据通过总线写入指定的地址。除了0xE2和0xF2以外，总线地址均使用一个16比特的寄存器表示。所有的总线读写操作都需要花费一个额外的机器周期，因此这一系列的LD指令全部花费2个机器周期完成。

作为案例，0x46 LD B, (HL)指令会将总线数据读入寄存器B，总线地址由HL寄存器提供，实现代码如下：

```
/// LD B, (HL) : Loads 8-bit data pointed by HL to B.
void x46_ld_b_mhl(Emulator* emu)
{
    emu->cpu.b = emu->bus_read(emu->cpu.hl());
    emu->tick(2);
}
```

代码中的mhl代表从hl所指向的内存中取值（m前缀表示memory），以便和hl（寄存器本身的值）做区分。

下表列出了类似的8比特总线读取操作，这些操作都可以使用上述的0x46指令通过修改目标寄存器和总线地址寄存器的方式实现：

操作码	指令	目标寄存器	总线地址
0x0A	LD A, (BC)	A	BC
0x1A	LD A, (DE)	A	DE
0x46	LD B, (HL)	B	HL
0x4E	LD C, (HL)	C	HL
0x56	LD D, (HL)	D	HL
0x5E	LD E, (HL)	E	HL
0x66	LD H, (HL)	H	HL
0x6E	LD L, (HL)	L	HL
0x7E	LD A, (HL)	A	HL

另一个例子是0x70 LD (HL), B指令，该指令则是反过来将寄存器B的值写入总线，总线地址由HL寄存器提供，代码如下：

```
/// LD (HL), B : Stores B to the memory pointed by HL.
void x70_ld_mhl_b(Emulator* emu)
{
    emu->bus_write(emu->cpu.hl(), emu->cpu.b);
    emu->tick(2);
}
```

下表列出了类似的8比特总线写入操作，这些操作都可以使用上述的0x70指令修改源寄存器和总线地址寄存器的方式实现：

操作码	指令	总线地址	源寄存器
0x02	LD (BC), A	BC	A
0x12	LD (DE), A	DE	A
0x70	LD (HL), B	HL	B
0x71	LD (HL), C	HL	C
0x72	LD (HL), D	HL	D
0x73	LD (HL), E	HL	E
0x74	LD (HL), H	HL	H
0x75	LD (HL), L	HL	L
0x77	LD (HL), A	HL	A

除了以上这些常规的8比特总线读取和写入指令以外，我们还有一些具有特殊行为的8比特总线读取和写入指令。下面将讲解这些特殊的指令。

0x22 LD (HL+), A

该指令类似0x77 LD (HL), A指令，区别在于其会在将8比特值写入总线以后自动将HL寄存器的值加1，并保持指令开销不变：

```
/// LD (HL+), A : Stores A to the memory pointed by HL, then increases HL.
void x22_ldi_mhl_a(Emulator* emu)
{
    emu->bus_write(emu->cpu.hl(), emu->cpu.a);
    emu->cpu.hl(emu->cpu.hl() + 1);
    emu->tick(2);
}
```

0x32 LD (HL-), A

该指令类似0x77 LD (HL), A指令，区别在于其会在将8比特值写入总线以后自动将HL寄存器的值减1，并保持指令开销不变：

```
/// LD (HL-) A : Stores A to the memory pointed by HL, then decreases HL.
void x32_ldd_mhl_a(Emulator* emu)
{
    emu->bus_write(emu->cpu.hl(), emu->cpu.a);
    emu->cpu.hl(emu->cpu.hl() - 1);
    emu->tick(2);
}
```

与之相对的还有**0x2A LD A, (HL+)**指令和**0x3A LD A, (HL-)** 指令，分别是在读取总线值至A寄存器以后将HL寄存器的值加1或减1，并保持指令开销不变：

```
/// LD A (HL+) : Loads value from memory pointed by HL to A, then increases HL.
void x2a_ldi_a_mhl(Emulator* emu)
{
    emu->cpu.a = emu->bus_read(emu->cpu.hl());
    emu->cpu.hl(emu->cpu.hl() + 1);
    emu->tick(2);
}
/// LD A (HL-) : Loads value from memory pointed by HL to A, then decreases HL.
void x3a_ldd_a_mhl(Emulator* emu)
{
    emu->cpu.a = emu->bus_read(emu->cpu.hl());
    emu->cpu.hl(emu->cpu.hl() - 1);
    emu->tick(2);
}
```

0xE2 LD (C), A

该指令将寄存器A的值写入总线地址0xFF00+C，开销为2个机器周期：

```
/// LD (C), A : Stores A to memory at 0xFF00 + C.
void xe2_ld_mc_a(Emulator* emu)
{
    emu->bus_write(0xFF00 + (u16)emu->cpu.c, emu->cpu.a);
    emu->tick(2);
}
```

0xF2 LD A, (C)

该指令从总线地址0xFF00+C读取值，并存储在寄存器A中，开销为2个机器周期：

```
/// LD A, (C) : Loads data at memory 0xFF00 + C to A.
void xf2_ld_a_mc(Emulator* emu)
{
    emu->cpu.a = emu->bus_read(0xFF00 + (u16)emu->cpu.c);
    emu->tick(2);
}
```

8位立即数据LD指令

除了从寄存器、内存读取8比特数据外，我们也可以在指令中直接指定需要存储在寄存器或者地址里的数据，就像我们在编程时直接声明一个常量数值一样。直接指定的数据通常称为立即数据（immediate data），其会存储在操作码之后的一个字节（8比特立即数据）或者两个字节（16比特立即数据）中，因此这类指令至少具有2个字节的长度，我们需要在解析这类指令时多读取1~2位，以读取指令附带的立即数据。

为了方便之后的使用，我们定义两个专门用于读取8比特立即数据和16比特立即数据的函数：

```
/// Combines one 16-bit value from two 8-bit values.
inline constexpr u16 make_u16(u8 low, u8 high)
{
    return ((u16)low) | (((u16)high) << 8);
}
/// Reads 16-bit immediate data.
inline u16 read_d16(Emulator* emu)
{
    u16 r = make_u16(emu->bus_read(emu->cpu.pc), emu->bus_read(emu->cpu.pc + 1));
    emu->cpu.pc += 2;
    return r;
}
/// Reads 8-bit immediate data.
inline u8 read_d8(Emulator* emu)
{

```



```
    u8 r = emu->bus_read(emu->cpu.pc);  
    ++emu->cpu.pc;  
    return r;  
}
```

read_d8会读取PC指向的地址的数据，然后将PC指针向前移动1字节，并返回该数据；read_d16则会读取PC指向的下两个字节的数​​据，然后将PC指针向前移动2字节，并将数据组合成一个16位数据返回。由于GameBoy的CPU是一个小端（Little Endian）CPU，因此在读取16字节数据时，低8字节的数据存储在低地址位中，高8字节的数据存储在高地址位中，因此我们先读取低地址位数据，再读取高地址位数据，最后用一个make_u16辅助函数合并成16位数据返回。

有了以上读取立即数据的辅助函数，我们便可以实现操作立即数据的LD指令了。以**0x06 LD B, d8**指令为例，该指令读取8比特的立即数，并存储在寄存器B中。由于读取操作码的下一个字节需要额外的一个机器周期，因此该指令的机器周期为2，同时指令长度也为2：

```
/// LD B, d8 : Loads 8-bit immediate data to B.  
void x06_ld_b_d8(Emulator* emu)  
{  
    emu->cpu.b = read_d8(emu);  
    emu->tick(2);  
}
```

下表列出了类似的8比特立即数据读取操作，这些操作都可以使用上述的0x06指令通过修改目标寄存器的方式实现：

操作码	指令	目标寄存器
0x06	LD B, d8	B
0x0E	LD C, d8	C
0x16	LD D, d8	D
0x1E	LD E, d8	E
0x26	LD H, d8	H
0x2E	LD L, d8	L
0x3E	LD A, d8	A

除此之外，还有一些将立即数据和地址总线读取结合的指令：

0x36 LD (HL), d8

该指令将立即数据写入寄存器HL所指向的地址中。由于该指令需要读一次立即数并写入一次数据，因此需要总共3个机器周期：

```
/// LD (HL), d8 : Stores 8-bit immediate data to memory pointed by HL.  
void x36_ld_mhl_d8(Emulator* emu)  
{  
    u8 data = read_d8(emu);  
    emu->tick(1);  
    emu->bus_write(emu->cpu.hl(), data);  
    emu->tick(2);  
}
```

该指令与之前我们遇到过的指令有一个很大的不同：我们并非只在指令结束时调用一次emu->tick(3)，而是在指令中间调用一次emu->tick(1)，然后在指令结束时调用一次emu->tick(2)。我们这么做的目的是实现正确的总线读写时序，在前一章节中，我们曾经讲到过：

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。

回到我们的代码。x36_ld_mhl_d8函数的第一行调用read_d8(emu)读取了一个立即数，由于读取立即数的操作调用bus_read函数执行了一次总线读取操作，因此同一个机器周期中，我们无法再调用bus_write向总线写入数据，因此我们需要先调用emu->tick(1)步进到下一个机器周期，然后才能调用bus_write向总线写入数据，这样才能正确地实现总线的读写时序。

正确实现总线读写时序对于某些游戏来说至关重要。例如，在之后的章节中，我们将会实现GameBoy的计时器组件。GameBoy的计时器在每个时钟周期都会更新一次，当我们向计时器对应的总线地址读写数据时，一个机器周期的时差就会导致我们读取的数据产生差异，或者使得计时器过早或者过晚更新，从而导致计时器的计时产生误差。该误差如果累计起来，就可能对游戏的逻辑产生影响。在之后的指令中，当我们需要在一个指令内多次读写总线时，我们都需要考虑这些操作的总线时序，才能保证指令行为的正确性。

0xE0 LDH (a8), A

该指令将寄存器A的值写入0xFF00+a8中。该指令同样需要3个机器周期，并且在总线读写操作之间需要步进一个机器周期：

```
///LDH (a8) A : Stores A to high memory address (0xFF00 + a8).
void xe0_ldh_m8_a(Emulator* emu)
{
    u8 addr = read_d8(emu);
    emu->tick(1);
    emu->bus_write(0xFF00 + (u16)addr, emu->cpu.a);
    emu->tick(2);
}
```

0xF0 LDH A, (a8)

该指令读取地址0xFF00+a8的值，并存储在寄存器A中，花费3个机器周期。

```
///LDH A, (a8) : Loads data in high memory address (0xFF00 + a8) to A.
void xf0_ldh_a_m8(Emulator* emu)
{
    u8 addr = read_d8(emu);
    emu->tick(1);
    emu->cpu.a = emu->bus_read(0xFF00 + (u16)addr);
    emu->tick(2);
}
```

16位LD指令

除了8比特LD指令之外，GameBoy还提供了几个专门的16比特指令，用于操作16位寄存器。

0x01 LD BC, d16

该指令读取16位立即数据，并保存在BC寄存器中。由于读取2个字节的立即数据需要2个机器周期，因此该指令长度为3，且需要3个机器周期：

```
///LD BC, d16 : Loads 16-bit immediate data to BC.
void x01_ld_bc_d16(Emulator* emu)
{
    emu->cpu.bc(read_d16(emu));
    emu->tick(3);
}
```

下表列出了类似的16位立即数读取指令，这些指令都可以使用上述的0x01指令修改目标寄存器的方式实现：

操作码	指令	目标寄存器
0x01	LD BC, d16	BC
0x11	LD DE, d16	DE
0x21	LD HL, d16	HL
0x31	LD SP, d16	SP

0x08 LD (a16), SP

该指令将SP寄存器的数据写入指定的地址。由于写入的是16位数据，因此该地址的字节以及该地址的下一字节会被写入。该指令需要读取两个字节的总线数据，同时写入两个字节的总线数据，因此花费5个机器周期：

```
///LD (a16), SP : Stores SP to the specified address.
void x08_ld_a16_sp(Emulator* emu)
{
    u16 addr = read_d16(emu);
    emu->tick(2);
    emu->bus_write(addr, (u8)(emu->cpu.sp & 0xFF));
    emu->tick(1);
    emu->bus_write(addr + 1, (u8)(emu->cpu.sp >> 8));
    emu->tick(2);
}
```

该指令也指令集中唯一的16位总线操作指令。

0xEA LD (a16), A

该指令将A寄存器的值写入指定的地址。该指令需要读取两个字节的总线数据，同时写入一个字

节的总线数据，因此花费4个机器周期：

```
/// LD (a16), A : Stores A to the memory address.
void xea_ld_a16_a(Emulator* emu)
{
    u16 addr = read_d16(emu);
    emu->tick(2);
    emu->bus_write(addr, emu->cpu.a);
    emu->tick(2);
}
```

0xFA LD A, (a16)

该指令从指定的地址读取值，并存储在寄存器A中。该指令需要读取四个字节的总线数据，因此花费4个机器周期：

```
/// LD A, (a16) : Loads data at memory address to A.
void xfa_ld_a_a16(Emulator* emu)
{
    u16 addr = read_d16(emu);
    emu->tick(2);
    emu->cpu.a = emu->bus_read(addr);
    emu->tick(2);
}
```

0xF9 LD SP, HL

该指令将HL寄存器的值拷贝至SP寄存器，花费2个机器周期。该指令也是GameBoy中唯一的16位寄存器间拷贝数据的LD指令。

```
/// LD SP, HL : Loads HL to SP.
void xf9_ld_sp_hl(Emulator* emu)
{
    emu->cpu.sp = emu->cpu.hl();
    emu->tick(2);
}
```

0xF8 LD HL, SP+r8

该指令是所有LD指令中最复杂的指令。其首先读取SP寄存器的值，然后读取8位立即数，将其相加以后存储在HL寄存器中。该指令长度为2，需要3个机器周期完成。

该指令将8位立即数解析成8位有符号整数，因此其可以将SP寄存器的值在[-128, 127]之间偏移。同时，由于该指令涉及加法操作，因此其需要按照加法操作的规则（在下一章节中讲解）正确设置CPU的标志位，具体规则为：

- 1. 设置Z和N标志位为0。
- 2. 如果在加法过程中发生了第3位与第4位（最低位为0）的借位，则设置H标志位为1，否则为0。
- 3. 如果在加法过程中发生了第7位与第8位的借位，则设置C标志位为1，否则为0。

此处的借位可以是向上进位，也可以是向下退位。例如，如果原本SP的数据是15（0x0F），在加1之后变成了16（0x10），便触发第3位向第4位进位，并设置H标志位为1；反之，如果原本SP的数据是16（0x10），在减1之后变成了15（0x0F），便触发第4位向第3位退位，同样会设置H标志位为1。

那么如何实现这一检测呢？先给代码：

```
/// LD HL, SP + r8 : Loads SP + r8 to HL.
void xf8_ld_hl_sp_r8(Emulator* emu)
{
    emu->cpu.reset_fz();
    emu->cpu.reset_fn();
    u16 v1 = emu->cpu.sp;
    i16 v2 = (i16)((i8)read_d8(emu));
    emu->tick(1);
    u16 r = v1 + v2;
    u16 check = v1 ^ v2 ^ r;
    if(check & 0x10) emu->cpu.set_fh();
    else emu->cpu.reset_fh();
    if(check & 0x100) emu->cpu.set_fc();
    else emu->cpu.reset_fc();
    emu->cpu.hl(r);
    emu->tick(2);
}
```


此处v1是加法的第一个操作数（即SP的值），v2是第二个操作数（从指令中读取的8位有符号整数，转换到了16位有符号整数，以便和SP长度对齐），r是运算结果。在检查借位时，我们编写了以下代码：

```
u16 check = v1 ^ v2 ^ r;
if(check & 0x10) emu->cpu.set_fh();
else emu->cpu.reset_fh();
if(check & 0x100) emu->cpu.set_fc();
else emu->cpu.reset_fc();
```

该代码将v1、v2与r的值进行了两次异或操作，然后检查结果的第4位、第8位是否为1来判断是否发生了3-4借位和7-8借位。为了理解该操作，我们可以将该过程分成两步考虑：首先是v1与v2异或，然后将运算结果与r异或。

对于v1和v2的任意一位，在不考虑与低位的借位的情况下，加法操作无非就三种情况：

- 1. 如果运算前该位的值均为0，则相加后值为0。
- 2. 如果运算前该位的值一个为0一个为1，则相加后值为1。
- 3. 如果运算前该位的值均为1，则相加会发生进位，进位后该位的值为0。

这个结果与我们将v1和v2该位的值做异或操作完全符合！因此v1^v2实际上代表了两个数的每一位在不考虑进位时，在加法运算后的值。而我们将该值与实际的运算结果r再一次异或，相当于比较了考虑进位和不考虑进位情况下该位的值是否相等，如果第二次异或操作返回1，则说明两种情况下的结果不相等，换言之就是发生了借位。

CP指令

CP（compare）指令比较两个寄存器的值，并按照比较结果设置CPU的标志位。其中，比较的一方永远是寄存器A，而另一方则根据指令的不同可以是寄存器或者从地址读取的值。CP指令的主要作为是与JP或者JR指令配合，在特定条件满足时跳转至新的程序地址，以实现流程控制。

以0xB8 CP B为例。该指令比较寄存器A与寄存器B的值，并根据比较结果按照如下规则设置四个标志位的值：

- 1. 如果A与B值相等，则设置Z为1，否则设置为0。
- 2. N标志位永远设置为1。
- 3. 如果A的低四位小于B的低四位，则设置H标志位为1，否则设置为0。
- 4. 如果A小于B，则设置C标志位为1，否则设置为0。

有了这些设置以后，我们只需要检查Z和C的值，就可以区分两个值大于，等于，小于三种情况。为了避免编写重复代码，我们可以将比较操作作为一个单独的辅助函数实现：

```
///! Sets the zero flag if `v` is 0, otherwise resets the zero flag.
inline void set_zero_flag(Emulator* emu, u8 v)
{
    if(v)
    {
        emu->cpu.reset_fz();
    }
    else
    {
        emu->cpu.set_fz();
    }
}

///! Compares v1 with v2, and sets flags based on result.
inline void cp_8(Emulator* emu, u16 v1, u16 v2)
{
    u8 r = (u8)v1 - (u8)v2;
    set_zero_flag(emu, r);
    emu->cpu.set_fn();
    if((v1 & 0x0F) < (v2 & 0x0F)) emu->cpu.set_fh();
    else emu->cpu.reset_fh();
    if(v1 < v2) emu->cpu.set_fc();
    else emu->cpu.reset_fc();
}
```

这样一来，0xB8 CP B就可以如下实现：

```
///! CP B : Compares B with A.
void xb8_cp_b(Emulator* emu)
{
    cp_8(emu, emu->cpu.a, emu->cpu.b);
}
```

```
        emu->tick(1);
    }
```

下表列出了类似的CP指令，这些指令都可以使用上述的0xB8指令通过修改第二个寄存器的方式实现：

操作码	指令	比较数1	比较数2
0xB8	CP B	A	B
0xB9	CP C	A	C
0xBA	CP D	A	D
0xBB	CP E	A	E
0xBC	CP H	A	H
0xBD	CP L	A	L
0xBF	CP A	A	A

除此之外，**0xBE CP (HL)**指令将寄存器A的值与寄存器HL指向的总线数据进行比较。由于该指令需要额外读取一次总线，因此占用2个时间周期，其余行为与上述CP指令保持一致：

```
/// CP (HL) : Compares data pointed by HL with A.
void xbe_cp_mhl(Emulator* emu)
{
    cp_8(emu, emu->cpu.a, emu->bus_read(emu->cpu.hl()));
    emu->tick(2);
}
```

0xFE CP d8指令将寄存器A的值与给定的立即数进行比较。由于该指令需要额外读取一次总线，因此占用2个时间周期，其余行为与上述CP指令保持一致：

```
/// CP d8 : Compares A with 8-bit immediate data.
void xfe_cp_d8(Emulator* emu)
{
    cp_8(emu, emu->cpu.a, read_d8(emu));
    emu->tick(2);
}
```

JP指令

JP (jump) 指令将程序流程跳转到指定的地址。由于CPU永远从PC寄存器指向的地址读取下一条指令，因此只要修改PC寄存器的值就可以实现跳转。由于跳转会导致CPU流水线停顿，因此JP指令在真正执行跳转时需要一个机器周期的额外开销，而在条件不符合，没有执行跳转时则没有这一个机器周期的开销。唯一的例外是0xE9 JP (HL)指令，根据文档的描述，该跳转指令只需要一个机器周期就可以完成，原因作者也不知道。

0xC3 JP a16

该指令读取16位立即数据表示的地址，并跳转到对应的地址继续执行程序。该指令需要读取2个字节的立即数据，同时具有1个机器周期的停顿，因此花费4个机器周期。

```
/// JP a16 : Jumps to the 16-bit address.
void xc3_jp_a16(Emulator* emu)
{
    u16 addr = read_d16(emu);
    emu->cpu.pc = addr;
    emu->tick(4);
}
```

0xC2 JP NZ, a16

该指令与0xC3类似，但是其只有在Z标志位为0时才进行跳转，否则不执行任何操作，继续执行原先位置的下一条指令。该指令在跳转时花费4个机器周期，在不跳转时花费3个机器周期。

```
/// JP NZ, a16 : Jumps to the 16-bit address if Z is 0.
void xc2_jp_nz_a16(Emulator* emu)
{
    u16 addr = read_d16(emu);
    if(!emu->cpu.fz())
    {
        emu->cpu.pc = addr;
        emu->tick(4);
    }
}
```

```
    }
    else
    {
        emu->tick(3);
    }
}
```

0xCA JP Z, a16

该指令与0xC3类似，但是其只有在Z标志位为1时才进行跳转，否则不执行任何操作，继续执行原先位置的下一条指令。该指令在跳转时花费4个机器周期，在不跳转时花费3个机器周期。

```
///! JP Z, a16 : Jumps to the 16-bit address if Z is 1.
void xca_jp_z_a16(Emulator* emu)
{
    u16 addr = read_d16(emu);
    if(emu->cpu.fz())
    {
        emu->cpu.pc = addr;
        emu->tick(4);
    }
    else
    {
        emu->tick(3);
    }
}
```

0xD2 JP NC, a16

该指令与0xC3类似，但是其只有在C标志位为0时才进行跳转，否则不执行任何操作，继续执行原先位置的下一条指令。该指令在跳转时花费4个机器周期，在不跳转时花费3个机器周期。

```
///! JP NC, a16 : Jumps to the 16-bit address if C is 0.
void xd2_jp_nc_a16(Emulator* emu)
{
    u16 addr = read_d16(emu);
    if(!emu->cpu.fc())
    {
        emu->cpu.pc = addr;
        emu->tick(4);
    }
    else
    {
        emu->tick(3);
    }
}
```

0xDA JP C, a16

该指令与0xC3类似，但是其只有在C标志位为1时才进行跳转，否则不执行任何操作，继续执行原先位置的下一条指令。该指令在跳转时花费4个机器周期，在不跳转时花费3个机器周期。

```
///! JP C, a16 : Jumps to the 16-bit address if C is 1.
void xda_jp_c_a16(Emulator* emu)
{
    u16 addr = read_d16(emu);
    if(emu->cpu.fc())
    {
        emu->cpu.pc = addr;
        emu->tick(4);
    }
    else
    {
        emu->tick(3);
    }
}
```

0xE9 JP (HL)

该指令跳转到HL寄存器对应的地址继续执行程序，实际上就是将寄存器HL的值赋值给寄存器PC，如下所示：

```
///! JP HL : Jumps to HL.
```

```
void xe9_jp_hl(Emulator* emu)
{
    emu->cpu.pc = emu->cpu.hl();
    emu->tick(1);
}
```

该指令消耗1个机器周期。

JR指令

JR (jump relative) 指令与JP指令类似，但是其并不是直接覆盖PC寄存器的值，而是在PC寄存器的值的基础上加一个偏移值。该偏移值表示为8位有符号整数，因此只能表示[-128, 127]的偏移范围。在实际使用时，JP指令通常用于函数调用等长距离跳转，而JR指令则用于函数内部的跳转，例如条件分支，循环等。

无条件JR指令

0x18 JR r8指令读取8位立即数据，将其作为一个8位有符号整数与PC寄存器相加，并将结果存储至PC寄存器中。该指令需要读取1个字节的立即数据，同时具有1个机器周期的停顿，因此消耗3个机器周期。

```
///! JR r8 : Jumps to PC + r8.
void x18_jr_r8(Emulator* emu)
{
    i8 offset = (i8)read_d8(emu);
    emu->cpu.pc += (i16)offset;
    emu->tick(3);
}
```

有条件JR指令

与JP指令类似，JR指令也可以根据标志位C和Z的值来判断是否进行跳转。该指令在跳转时花费3个机器周期，在不跳转时花费2个机器周期。

以**0x28 JR Z, r8**为例，有条件JR指令的实现如下：

```
///! JR Z, r8 : Jumps to PC + r8 if Z is 1.
void x28_jr_z_r8(Emulator* emu)
{
    i8 offset = (i8)read_d8(emu);
    if(emu->cpu.fz())
    {
        emu->cpu.pc += (i16)offset;
        emu->tick(3);
    }
    else
    {
        emu->tick(2);
    }
}
```

下表列出了类似的有条件JR指令，这些指令都可以使用上述的0x28指令通过修改跳转条件方式实现：

操作码	指令	跳转条件
0x20	JR NZ, r8	Z为0
0x28	JR Z, r8	Z为1
0x30	JR NC, r8	C为0
0x38	JR C, r8	C为1

PUSH指令

PUSH指令将给定的16位寄存器中的数据压入栈顶。栈是一个由SP（stack pointer）寄存器表示的任意地址，向栈中压入数据的操作会先将SP寄存器中的地址减2，然后将寄存器中的值写入总线（GameBoy的栈是从高地址位往低地址位入栈的）。

为了减少重复代码，我们先编写一个辅助函数用于将16位数据入栈：

```
///! Pushes 16-bit data into stack.
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));
}
```

然后就可以实现push指令。以0xC5 PUSH BC为例：

```
///! PUSH BC : Pushes BC to the stack.
void xc5_push_bc(Emulator* emu)
{
    push_16(emu, emu->cpu.bc());
    emu->tick(4);
}
```

所有的PUSH指令均消耗4个机器周期。下表列出了类似的PUSH指令，这些指令都可以使用上述的0xC5指令通过修改寄存器的方式实现：

操作码	指令	源寄存器
0xC5	PUSH BC	BC
0xD5	PUSH DE	DE
0xE5	PUSH HL	HL
0xF5	PUSH AF	AF

POP指令

POP指令从栈顶取出16位数据，然后存储到给定寄存器中。为了减少重复代码，我们先编写一个辅助函数用于将16位数据出栈：

```
///! Pops 16-bit data from stack.
inline u16 pop_16(Emulator* emu)
{
    u8 lo = emu->bus_read(emu->cpu.sp);
    u8 hi = emu->bus_read(emu->cpu.sp + 1);
    emu->cpu.sp += 2;
    return make_u16(lo, hi);
}
```

然后就可以实现pop指令。以0xC1 POP BC为例：

```
///! POP BC : Pops 16-bit data from stack to BC.
void xc1_pop_bc(Emulator* emu)
{
    emu->cpu.bc(pop_16(emu));
    emu->tick(3);
}
```

所有的POP指令均消耗4个机器周期。下表列出了类似的POP指令，这些指令都可以使用上述的0xC1指令通过修改寄存器的方式实现：

操作码	指令	目标寄存器
0xC1	POP BC	BC
0xD1	POP DE	DE
0xE1	POP HL	HL
0xF1	POP AF	AF

CALL指令

CALL指令用于调用一个函数。其首先会将当前的PC寄存器通过类似PUSH指令的操作入栈，然后使用类似JP指令的操作将PC寄存器设置为新的程序地址。

无条件CALL指令

0xCD CALL a16指令读取16位立即数，然后将其作为地址调用。该指令花费6个机器周期。

```
///! CALL a16 : Calls the function.
void xcd_call_a16(Emulator* emu)
{
    u16 addr = read_d16(emu);
    emu->tick(2);
}
```



```
    push_16(emu, emu->cpu.pc);
    emu->cpu.pc = addr;
    emu->tick(4);
}
```

有条件CALL指令

与JP和JR类似，CALL指令可以根据C和Z标志位的值来判断是否执行跳转。该指令在跳转时花费6个机器周期，在不跳转时花费3个机器周期。

以0xC4 CALL NZ, a16为例，有条件CALL指令的实现如下：

```
///CALL NZ, a16 : Calls the function if Z is 0.
void xc4_call_nz_a16(Emulator* emu)
{
    u16 addr = read_d16(emu);
    emu->tick(2);
    if(!emu->cpu.fz())
    {
        push_16(emu, emu->cpu.pc);
        emu->cpu.pc = addr;
        emu->tick(4);
    }
    else
    {
        emu->tick(1);
    }
}
```

下表列出了类似的有条件CALL指令，这些指令都可以使用上述的0xC4指令通过修改跳转条件方式实现：

操作码	指令	跳转条件
0xC4	CALL NZ, a16	Z为0
0xCC	CALL Z, a16	Z为1
0xD4	CALL NC, a16	C为0
0xDC	CALL C, a16	C为1

RET指令

RET指令与CALL指令相反，用于从一个函数返回其调用函数。RET指令首先从栈顶取出16位数据（假定该数据是通过CALL压入栈中的），然后将其存储至PC寄存器，这样CPU就会回到调用函数的程序中接着执行下一条指令。

我们在通过CALL指令将PC寄存器值压入栈之前，PC寄存器的值就已经指向CALL指令的下一条指令了，因此在我们调用RET返回以后，CPU可以直接执行PC寄存器指向的指令，而无需对其再加1。

无条件RET指令

0xC9 RET指令无条件返回至调用函数，花费4个机器周期。

```
///RET : Returns the function.
void xc9_ret(Emulator* emu)
{
    emu->cpu.pc = pop_16(emu);
    emu->tick(4);
}
```

有条件RET指令

与JP、JR和CALL类似，RET指令可以根据C和Z标志位的值来判断是否执行跳转。该指令在跳转时花费5个机器周期，在不跳转时花费2个机器周期。

以0xC0 RET NZ为例，有条件RET指令的实现如下：

```
///RET NZ : Returns the function if Z is 0.
void xc0_ret_nz(Emulator* emu)
{
    if(!emu->cpu.fz())
    {
```

```
        emu->cpu.pc = pop_16(emu);
        emu->tick(5);
    }
    else
    {
        emu->tick(2);
    }
}
```

下表列出了类似的有条件RET指令，这些指令都可以使用上述的0xC0指令通过修改跳转条件方式实现：

操作码	指令	跳转条件
0xC0	RET NZ	Z为0
0xC8	RET Z	Z为1
0xD0	RET NC	C为0
0xD8	RET C	C为1

RETI指令

除了上述指令以外，RET系列还有一个特殊的指令——**0xD9 RETI**，该指令表现与**0xC9 RET**一致，只是其会在返回的同时激活中断（如果当前中断处于未激活状态）。RETI指令主要用于从中断处理函数返回，以便在返回函数的同时打开中断。我们将在后续的章节中讲解中断及其实现，现在我们先在CPU类中添加一个空函数用于在后续章节中执行激活中断的操作：

```
void enable_interrupt_master() { /*TODO*/ }
```

然后就可以实现RETI指令：

```
/// RETI : Returns and enables interruption.
void xd9_reti(Emulator* emu)
{
    emu->cpu.enable_interrupt_master();
    xc9_ret(emu);
}
```

RST指令

RST (reset) 指令与CALL指令类似，其将当前PC寄存器的值压入栈顶，然后将PC寄存器设置为固定值。每个RST指令都只能将PC寄存器设置为一个特定值，其机器周期均为4，且均为无条件跳转指令。

以**0xC7 RST 00H**为例，其将PC寄存器的值压入栈顶后，将PC寄存器设置为0x0000，实现如下：

```
/// RST 00H : Pushes PC to stack and resets PC to 0x00.
void xc7_rst_00h(Emulator* emu)
{
    push_16(emu, emu->cpu.pc);
    emu->cpu.pc = 0x0000;
    emu->tick(4);
}
```

下表列出了类似的RST指令，这些指令都可以使用上述的0xC7指令通过修改跳转地址方式实现：

操作码	指令	跳转地址
0xC7	RST 00H	0x0000
0xCF	RST 08H	0x0008
0xD7	RST 10H	0x0010
0xDF	RST 18H	0x0018
0xE7	RST 20H	0x0020
0xEF	RST 28H	0x0028
0xF7	RST 30H	0x0030
0xFF	RST 38H	0x0038

在实现了上述所有指令以后，instructions_map应类似如下所示：

```
instruction_func_t* instructions_map[256] =
```

```
{
    x00_nop,      x01_ld_bc_d16, x02_ld_mbc_a,  nullptr, nullptr, nullptr, x06_
    nullptr,      x11_ld_de_d16, x12_ld_mde_a,  nullptr, nullptr, nullptr, x16_
    x20_jr_nz_r8, x21_ld_hl_d16, x22_ldi_mhl_a,  nullptr, nullptr, nullptr, x26_
    x30_jr_nc_r8, x31_ld_sp_d16, x32_ldd_mhl_a,  nullptr, nullptr, nullptr, x36_
    x40_ld_b_b,   x41_ld_b_c,   x42_ld_b_d,   x43_ld_b_e,   x44_ld_b_h,   x45_
    x50_ld_d_b,   x51_ld_d_c,   x52_ld_d_d,   x53_ld_d_e,   x54_ld_d_h,   x55_
    x60_ld_h_b,   x61_ld_h_c,   x62_ld_h_d,   x63_ld_h_e,   x64_ld_h_h,   x65_
    x70_ld_mhl_b, x71_ld_mhl_c, x72_ld_mhl_d, x73_ld_mhl_e, x74_ld_mhl_h, x75_
    nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nu
    nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nu
    nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nu
    nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, xbf
    xc0_ret_nz,   xc1_pop_bc, xc2_jp_nz_a16, xc3_jp_a16, xc4_call_nz_a16, xc5_
    xd0_ret_nc,   xd1_pop_de, xd2_jp_nc_a16, nullptr,   xd4_call_nc_a16, xd5_
    xe0_ldh_m8_a, xe1_pop_hl, xe2_ld_mc_a,   nullptr,   nullptr,           xe5_
    xf0_ldh_a_m8, xf1_pop_af, xf2_ld_a_mc,   nullptr,   nullptr,           xf5_
};
```

以上就是本章节的全部内容了。下一章我们将继续实现CPU的逻辑和算术指令。这几章的内容会有些枯燥，但是如果能够坚持下来，将对以后实现别的模拟器或者虚拟机系统会有很大的帮助。那么，下一章见！

銀葉吉祥：从零开始实现GameBoy模拟器 #4 逻辑和运算指令

22 赞同 · 0 评论 文章

编辑于 2024-03-11 21:05 · IP 属地上海

Game Boy (GB) 模拟器 游戏机模拟器

362 赞同

欢迎参与讨论

2 条评论

默认 最新

firast

...

0xf9 "该指令将SP寄存器的值拷贝至HL寄存器" 是否写错了

01-30 · IP 属地安徽

回复 喜欢

銀葉吉祥 作者

...

确实，已更正，感谢反馈！

01-30 · IP 属地上海

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

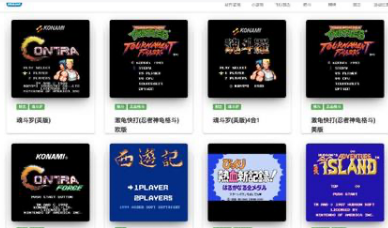
iOS Delta 模拟器使用说明及游戏资源大全

GBA4iOS 的正统续作 Delta 在 App Store 上架了，完全免费，不用再折腾越狱或自签名了。Delta emulator支持NES、SNES、Game Boy、Game Boy Color、Game Boy Advance、Nintendo DS、

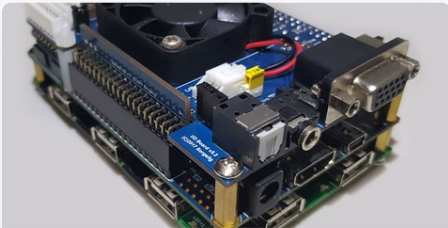
梁川 发表于互联网支付...



绝地求生：刺激战场专用安卓模拟器是哪款？什么模拟器玩七七的金色城堡



在这款神还原的小霸王模拟器上，我终于玩到了20年前的老扩展迷Extfans



开源FPGA硬件模拟游戏机，原汁原味的复古游戏体验带你回量子位 发表于量子位

赞同 22 2 条评论 分享 喜欢 收藏 申请转载 ...

