


从零开始实现GameBoy模拟器 #4 逻辑和运算指令

 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

- INC指令
 - 8位INC指令
 - 16位INC指令
- DEC指令
 - 8位DEC指令
 - 16位DEC指令
- ADD指令
 - 8位ADD指令
 - 16位ADD指令
- ADC指令
- SUB指令
- SBC指令
- AND指令
- XOR指令
- OR指令
- CPL指令
- DAA指令



欢迎来到从零开始实现GameBoy模拟器第四章。本章将接着上一章的内容，继续为我们的模拟器实现CPU的逻辑和运算指令。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-04，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

INC指令

INC (increase) 指令用于将指定位置的值增加1。INC指令分为8位和16位两类，8位INC指令在增加值以后会设置CPU标志位，而16位INC只会增加值，不会影响CPU标志位。

8位INC指令

8位INC指令根据运算结果按照如下规则设置标志位的值：

- 如果运算后的值为0（发生了上溢），则设置Z为1，否则设置为0。
- N的值固定设置为0。
- 如果运算过程中发生了第3位向第4位（最低位为0）的进位，则设置H标志位为1，否则为0。

为了避免编写重复代码，我们可以将自增操作作为一个单独的辅助函数实现：

```
inline void inc_8(Emulator* emu, u8& v)
{
    ++v;
    set_zero_flag(emu, v);
    emu->cpu.reset_fn();
    if((v & 0x0F) == 0x00)
    {
        emu->cpu.set_fh();
    }
    else
    {
        emu->cpu.reset_fh();
    }
}
```

然后就可以实现具体的指令。以0x04 INC B为例，该指令将寄存器B的值增加1，花费一个机器

周期：

```
///! INC B : Increases B.
void x04_inc_b(Emulator* emu)
{
    inc_8(emu, emu->cpu.b);
    emu->tick(1);
}
```

下表列出了类似的INC指令，这些指令都可以使用上述的0x04指令通过修改寄存器的方式实现：

操作码	指令	寄存器
0x04	INC B	B
0x0C	INC C	C
0x14	INC D	D
0x1C	INC E	E
0x24	INC H	H
0x2C	INC L	L
0x3C	INC A	A

除此之外，**0x34 INC (HL)**指令会读取HL寄存器指向的总线地址的值，将该值按照上述规则计算后将结果写入总线。该指令花费3个机器周期，代码实现如下：

```
///! INC (HL) : Increases data in memory pointed by HL.
void x34_inc_mhl(Emulator* emu)
{
    u8 data = emu->bus_read(emu->cpu.hl());
    emu->tick(1);
    inc_8(emu, data);
    emu->bus_write(emu->cpu.hl(), data);
    emu->tick(2);
}
```

16位INC指令

16位INC指令只操作寄存器数值，不影响CPU标志位，因此实现起来相对简单。以**0x03 INC BC**为例：

```
///! INC BC : Increases BC.
void x03_inc_bc(Emulator* emu)
{
    emu->cpu.bc(emu->cpu.bc() + 1);
    emu->tick(2);
}
```

下表列出了类似的INC指令，这些指令都可以使用上述的0x03指令通过修改寄存器的方式实现：

操作码	指令	寄存器
0x03	INC BC	BC
0x13	INC DE	DE
0x23	INC HL	HL
0x33	INC SP	SP

其中**0x33 INC SP**指令可以直接使用++emu->cpu.sp实现：

```
///! INC SP : Increases SP.
void x33_inc_sp(Emulator* emu)
{
    ++emu->cpu.sp;
    emu->tick(2);
}
```

DEC指令

DEC（decrease）指令用于将指定位置的值减少1。DEC指令分为8位和16位两类，8位DEC指令在减少值以后会设置CPU标志位，而16位INC只会减少值，不会影响CPU标志位。

8位DEC指令

8位DEC指令根据运算结果按照如下规则设置标志位的值：

- 1. 如果运算后的值为0，则设置Z为1，否则设置为0。
- 2. N的值固定设置为1。
- 3. 如果运算过程中发生了第4位向第3位的退位，则设置H标志位为1，否则为0。

为了避免编写重复代码，我们可以将自减操作作为一个单独的辅助函数实现：

```
inline void dec_8(Emulator* emu, u8& v)
{
    --v;
    set_zero_flag(emu, v);
    emu->cpu.set_fn();
    if((v & 0x0F) == 0x0F)
    {
        emu->cpu.set_fh();
    }
    else
    {
        emu->cpu.reset_fh();
    }
}
```

然后就可以实现具体的指令。以0x05 DEC B为例，该指令将寄存器B的值减少1，花费一个机器周期：

```
///! DEC B : Decreases B.
void x05_dec_b(Emulator* emu)
{
    dec_8(emu, emu->cpu.b);
    emu->tick(1);
}
```

下表列出了类似的DEC指令，这些指令都可以使用上述的0x05指令通过修改寄存器的方式实现：

操作码	指令	寄存器
0x05	DEC B	B
0x0D	DEC C	C
0x15	DEC D	D
0x1D	DEC E	E
0x25	DEC H	H
0x2D	DEC L	L
0x3D	DEC A	A

除此之外，0x35 DEC (HL)指令会读取HL寄存器指向的总线地址的值，将该值按照上述规则计算后将结果写入总线。该指令花费3个机器周期，代码实现如下：

```
///! DEC (HL) : Decreases data in memory pointed by HL.
void x35_dec_mhl(Emulator* emu)
{
    u8 data = emu->bus_read(emu->cpu.hl());
    emu->tick(1);
    dec_8(emu, data);
    emu->bus_write(emu->cpu.hl(), data);
    emu->tick(2);
}
```

16位DEC指令

16位DEC指令只操作寄存器数值，不影响CPU标志位，因此实现起来相对简单。以0x0B DEC BC为例：

```
///! DEC BC : Decreases BC.
void x0b_dec_bc(Emulator* emu)
{
    emu->cpu.bc(emu->cpu.bc() - 1);
    emu->tick(2);
}
```

下表列出了类似的DEC指令，这些指令都可以使用上述的0x0B指令通过修改寄存器的方式实现：

--	--	--

操作码	指令	寄存器
0x0B	DEC BC	BC
0x1B	DEC DE	DE
0x2B	DEC HL	HL
0x3B	DEC SP	SP

其中**0x3B DEC SP**指令可以直接使用--emu->cpu.sp实现：

```
///! DEC SP : Decreases SP.
void x3b_dec_sp(Emulator* emu)
{
    --emu->cpu.sp;
    emu->tick(2);
}
```

ADD指令

ADD指令执行加法操作，并将结果存储在指定的寄存器中。ADD指令分为8位和16位两类，下面将予以介绍。除了**0xE8 ADD SP, r8**外，所有的加法操作均不考虑符号，即两个操作数均视为无符号整数，且结果也为无符号整数。

8位ADD指令

8位ADD指令将两个8位的值相加，并将结果保存在目标寄存器或者地址中。同时，8位ADD指令会按照如下规则设置CPU标志位：

- 如果运算后的值为0，则设置Z为1，否则设置为0。
- N标志位固定设置为0。
- 如果在加法过程中发生了第3位到第4位（最低位为0）的进位，则设置H标志位为1，否则为0。
- 如果在加法过程中发生了第7位到第8位的进位，则设置C标志位为1，否则为0。

为了避免编写重复代码，我们可以将8位加法操作作为一个单独的辅助函数实现：

```
inline u8 add_8(Emulator* emu, u16 v1, u16 v2)
{
    u8 r = (u8)(v1 + v2);
    set_zero_flag(emu, r);
    emu->cpu.reset_fn();
    if((v1 & 0x0F) + (v2 & 0x0F) > 0x0F) emu->cpu.set_fh();
    else emu->cpu.reset_fh();
    if(v1 + v2 > 0xFF) emu->cpu.set_fc();
    else emu->cpu.reset_fc();
    return r;
}
```

然后就可以实现具体的指令。以**0x80 ADD A, B**为例：

```
///! ADD A, B : Adds B to A.
void x80_add_a_b(Emulator* emu)
{
    emu->cpu.a = add_8(emu, emu->cpu.a, emu->cpu.b);
    emu->tick(1);
}
```

下表列出了类似的ADD指令，这些指令都可以使用上述的0x80指令通过修改寄存器的方式实现：

操作码	指令	寄存器1	寄存器2	结果寄存器
0x80	ADD A, B	A	B	A
0x81	ADD A, C	A	C	A
0x82	ADD A, D	A	D	A
0x83	ADD A, E	A	E	A
0x84	ADD A, H	A	H	A
0x85	ADD A, L	A	L	A
0x87	ADD A, A	A	A	A

除此之外，我们还有一些特殊的8位ADD指令：

0xC6 ADD A, d8

该指令读取8位立即数，并将其与寄存器A的结果相加，存储在寄存器A中，并按照与上述ADD指令相同的规则更新CPU标志位。

```
///! ADD A, d8 : Adds 8-bit immediate data to A.
void xc6_add_a_d8(Emulator* emu)
{
    u8 data = read_d8(emu);
    emu->tick(1);
    emu->cpu.a = add_8(emu, emu->cpu.a, data);
    emu->tick(1);
}
```

0x86 ADD A, (HL)

该指令读取HL寄存器所存储的地址的值，并将其与寄存器A的结果相加，存储在寄存器A中，并按照与上述ADD指令相同的规则更新CPU标志位。

```
///! ADD A, (HL) : Adds data pointed by HL to A.
void x86_add_a_mhl(Emulator* emu)
{
    u8 data = emu->bus_read(emu->cpu.hl());
    emu->tick(1);
    emu->cpu.a = add_8(emu, emu->cpu.a, data);
    emu->tick(1);
}
```

16位ADD指令

16位ADD指令将两个16位的值相加，或将一个16位的值与一个8位的值相加，并将结果保存在16位寄存器中。同时，16位ADD指令会按照如下规则设置CPU标志位：

- 1. N标志位固定设置为0。
- 2. 如果在加法过程中发生了第11位到第12位（最低位为0）的进位，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15位到第16位的进位，则设置C标志位为1，否则为0。
- 4. Z标志位不做修改。

为了避免编写重复代码，我们可以将16位加法操作作为一个单独的辅助函数实现：

```
inline u16 add_16(Emulator* emu, u32 v1, u32 v2)
{
    emu->cpu.reset_fn();
    if((v1 & 0xFFF) + (v2 & 0xFFF) > 0xFFF) emu->cpu.set_fh();
    else emu->cpu.reset_fh();
    if(v1 + v2 > 0xFFFF) emu->cpu.set_fc();
    else emu->cpu.reset_fc();
    return (u16)(v1 + v2);
}
```

然后就可以实现具体的指令。以0x09 ADD HL, BC为例：

```
///! ADD HL, BC : Adds BC to HL.
void x09_add_hl_bc(Emulator* emu)
{
    emu->cpu.hl(add_16(emu, emu->cpu.hl(), emu->cpu.bc()));
    emu->tick(2);
}
```

下表列出了类似的ADD指令，这些指令都可以使用上述的0x09指令通过修改寄存器的方式实现：

操作码	指令	寄存器1	寄存器2	结果寄存器
0x09	ADD HL, BC	HL	BC	HL
0x19	ADD HL, DE	HL	DE	HL
0x29	ADD HL, HL	HL	HL	HL
0x39	ADD HL, SP	HL	SP	HL

除此之外，我们还有一些特殊的16位ADD指令：

0xE8 ADD SP, r8

该指令读取8位立即数，将其作为**有符号整数**（范围为-128至127）与SP寄存器的值相加，并将结果存储在SP寄存器中。同时，该指令按照如下规则操作标志位：

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。

此处的标志位实现与上一章中实现**0xF8 LD HL, SP+r8**指令完全一致，只是将目标寄存器从HL换成了SP，同时开销从3个周期变为了4个周期。可以参考上一章中对0xF8指令的解释来理解。代码实现如下：

```

///! ADD SP, r8 : Adds 8-bit immediate signed integer to SP.
void xe8_add_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.sp = r;
    emu->tick(3);
}
```

ADC指令

ADC指令（add with carry）与ADD指令一样用于将两个值相加，但是如果在调用ADC指令时CPU的C标志位为1，则ADC指令会将结果额外加1，以考虑上一次运算中的进位操作。ADC指令只有8位的版本，没有16位的版本，一般用于配合ADD指令实现多字节的加法操作，也可以将多个ADC指令串起来以对以个任意字节数据进行连续加法。

ADC对标志位的更新逻辑和8位的ADD指令完全一致，因此不再赘述。为了避免编写重复代码，我们可以将ADC操作作为一个单独的辅助函数实现：

```

inline u8 adc_8(Emulator* emu, u16 v1, u16 v2)
{
    u16 c = emu->cpu.fc() ? 1 : 0;
    u8 r = (u8)(v1 + v2 + c);
    set_zero_flag(emu, r);
    emu->cpu.reset_fn();
    if((v1 & 0x0F) + (v2 & 0x0F) + c > 0x0F) emu->cpu.set_fh();
    else emu->cpu.reset_fh();
    if((v1 + v2 + c) > 0xFF) emu->cpu.set_fc();
    else emu->cpu.reset_fc();
    return r;
}
```

然后就可以实现具体的指令。以**0x88 ADC A, B**为例：

```

///! ADC A, B : Adds B to A with carry.
void x88_adc_a_b(Emulator* emu)
{
    emu->cpu.a = adc_8(emu, emu->cpu.a, emu->cpu.b);
    emu->tick(1);
}
```

下表列出了类似的ADC指令，这些指令都可以使用上述的0x88指令通过修改寄存器的方式实现：

操作码	指令	寄存器1	寄存器2	结果寄存器
0x88	ADC A, B	A	B	A
0x89	ADC A, C	A	C	A
0x8A	ADC A, D	A	D	A
0x8B	ADC A, E	A	E	A
0x8C	ADC A, H	A	H	A
0x8D	ADC A, L	A	L	A

0x8F	ADC A, A	A	A	A
------	----------	---	---	---

除此之外，我们还有一些特殊的8位ADC指令：

0xCE ADC A, d8

该指令读取8位立即数，并将其与寄存器A的结果相加，存储在寄存器A中，并按照与上述ADC指令相同的规则更新CPU标志位。

```
/// ADC A, d8 : Adds 8-bit immediate data to A with carry.
void xce_adc_a_d8(Emulator* emu)
{
    u8 data = read_d8(emu);
    emu->tick(1);
    emu->cpu.a = adc_8(emu, emu->cpu.a, data);
    emu->tick(1);
}
```

0x8E ADC A, (HL)

该指令读取HL寄存器所存储的地址的值，并将其与寄存器A的结果相加，存储在寄存器A中，并按照与上述ADC指令相同的规则更新CPU标志位。

```
/// ADC A, (HL) : Adds data pointed by HL to A with carry.
void x8e_adc_a_mhl(Emulator* emu)
{
    u8 data = emu->bus_read(emu->cpu.hl());
    emu->tick(1);
    emu->cpu.a = adc_8(emu, emu->cpu.a, data);
    emu->tick(1);
}
```

SUB指令

SUB指令执行减法操作，并将结果存储在指定的寄存器中。所有的SUB指令均操作8位无符号整数，且被减数均为**A寄存器**。SUB指令会根据下列规则设置CPU寄存器：

- 如果运算后的值为0，则设置Z为1，否则设置为0。
- N标志位固定设置为1。
- 如果在减法过程中发生了第4位到第3位（最低位为0）的退位，则设置H标志位为1，否则为0。
- 如果在减法过程中发生了第8位到第7位的退位（即发生下溢），则设置C标志位为1，否则为0。

为了避免编写重复代码，我们可以将减法操作作为一个单独的辅助函数实现：

```
inline u8 sub_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();
    return r;
}
```

然后就可以实现具体的指令。以**0x90 SUB B**为例：

```
/// SUB B : Subtracts B from A.
void x90_sub_b(Emulator* emu)
{
    emu->cpu.a = sub_8(emu, emu->cpu.a, emu->cpu.b);
    emu->tick(1);
}
```

下表列出了类似的SUB指令，这些指令都可以使用上述的0x90指令通过修改寄存器的方式实现：

操作码	指令	寄存器1	寄存器2	结果寄存器
0x90	SUB B	A	B	A
0x91	SUB C	A	C	A

0x92	SUB D	A	D	A
0x93	SUB E	A	E	A
0x94	SUB H	A	H	A
0x95	SUB L	A	L	A
0x97	SUB A	A	A	A

除此之外，我们还有一些特殊的SUB指令：

0xD6 SUB d8

该指令读取8位立即数，并将其与寄存器A的结果相减，存储在寄存器A中，并按照与上述SUB指令相同的规则更新CPU标志位。

```
///! SUB d8 : Subtracts 8-bit immediate data from A.
void xd6_sub_d8(Emulator* emu)
{
    u8 data = read_d8(emu);
    emu->tick(1);
    emu->cpu.a = sub_8(emu, emu->cpu.a, data);
    emu->tick(1);
}
```

0x96 SUB (HL)

该指令读取HL寄存器所存储的地址的值，并将其与寄存器A的结果相减，存储在寄存器A中，并按照与上述SUB指令相同的规则更新CPU标志位。

```
///! SUB (HL) : Subtracts data pointed by HL from A.
void x96_sub_mhl(Emulator* emu)
{
    u8 data = emu->bus_read(emu->cpu.hl());
    emu->tick(1);
    emu->cpu.a = sub_8(emu, emu->cpu.a, data);
    emu->tick(1);
}
```

SBC指令

SBC指令（subtract with carry）与SUB指令一样用于将两个值相减，但是如果在调用SBC指令时CPU的C标志位为1，则SBC指令会将结果额外减1，以考虑上一次运算中的退位操作。SBC指令一般用于配合SUB指令实现多字节的减法操作，也可以将多个SBC指令串起来以对以个任意字节数据进行连续减法。

SBC对标志位的更新逻辑和SUB指令完全一致，因此不再赘述。为了避免编写重复代码，我们可以将SBC操作作为一个单独的辅助函数实现：

```
inline u8 sbc_8(Emulator* emu, u16 v1, u16 v2)
{
    u8 c = emu->cpu.fc() ? 1 : 0;
    u8 r = (u8)v1 - (u8)v2 - c;
    set_zero_flag(emu, r);
    emu->cpu.set_fn();
    if((v1 & 0x0F) < ((v2 & 0x0F) + c)) emu->cpu.set_fh();
    else emu->cpu.reset_fh();
    if(v1 < (v2 + c)) emu->cpu.set_fc();
    else emu->cpu.reset_fc();
    return r;
}
```

然后就可以实现具体的指令。以0x98 SBC A, B为例：

```
///! SBC A, B : Subtracts B from A with carry.
void x98_sbc_a_b(Emulator* emu)
{
    emu->cpu.a = sbc_8(emu, emu->cpu.a, emu->cpu.b);
    emu->tick(1);
}
```

下表列出了类似的SBC指令，这些指令都可以使用上述的0x98指令通过修改寄存器的方式实现：

操作码	指令	寄存器1	寄存器2	结果寄存器

0x98	SBC A, B	A	B	A
0x99	SBC A, C	A	C	A
0x9A	SBC A, D	A	D	A
0x9B	SBC A, E	A	E	A
0x9C	SBC A, H	A	H	A
0x9D	SBC A, L	A	L	A
0x9F	SBC A, A	A	A	A

除此之外，我们还有一些特殊的SBC指令：

0xDE SBC A, d8

该指令读取8位立即数，并将其与寄存器A的结果相减，存储在寄存器A中，并按照与上述SBC指令相同的规则更新CPU标志位。

```
///! SBC A, d8 : Subtracts 8-bit immediate data from A.
void xde_sbc_a_d8(Emulator* emu)
{
    u8 data = read_d8(emu);
    emu->tick(1);
    emu->cpu.a = sbc_8(emu, emu->cpu.a, data);
    emu->tick(1);
}
```

0x9E SBC A, (HL)

该指令读取HL寄存器所存储的地址的值，并将其与寄存器A的结果相减，存储在寄存器A中，并按照与上述SBC指令相同的规则更新CPU标志位。

```
///! SBC A, (HL) : Subtracts data pointed by HL from A with carry.
void x9e_sbc_a_mhl(Emulator* emu)
{
    u8 data = emu->bus_read(emu->cpu.hl());
    emu->tick(1);
    emu->cpu.a = sbc_8(emu, emu->cpu.a, data);
    emu->tick(1);
}
```

AND指令

AND指令执行按位与操作，并将结果存储在指定的寄存器中。所有的AND指令均操作8位无符号整数，且其中一个操作数均为A寄存器。AND指令会根据下列规则设置CPU寄存器：

1. 如果运算后的值为0，则设置Z为1，否则设置为0。
2. N标志位固定设置为0。
3. H标志位固定设置为1。
4. C标志位固定设置为0。

为了避免编写重复代码，我们可以将AND操作作为一个单独的辅助函数实现：

```
inline u8 and_8(Emulator* emu, u8 v1, u8 v2)
{
    u8 r = v1 & v2;
    set_zero_flag(emu, r);
    emu->cpu.reset_fn();
    emu->cpu.set_fh();
    emu->cpu.reset_fc();
    return r;
}
```

然后就可以实现具体的指令。以0xA0 AND B为例：

```
///! AND B : Performs bitwise AND on A and B.
void xa0_and_b(Emulator* emu)
{
    emu->cpu.a = and_8(emu, emu->cpu.a, emu->cpu.b);
    emu->tick(1);
}
```

下表列出了类似的AND指令，这些指令都可以使用上述的0xA0指令通过修改寄存器的方式实现：

操作码	指令	寄存器1	寄存器2	结果寄存器
0xA0	AND B	A	B	A
0xA1	AND C	A	C	A
0xA2	AND D	A	D	A
0xA3	AND E	A	E	A
0xA4	AND H	A	H	A
0xA5	AND L	A	L	A
0xA7	AND A	A	A	A

除此之外，我们还有一些特殊的AND指令：

0xE6 AND, d8

该指令读取8位立即数，并将其与寄存器A的结果做按位与操作，存储在寄存器A中，并按照与上述AND指令相同的规则更新CPU标志位。

```
///! AND d8 : Performs bitwise AND between A and 8-bit immediate data.
void xe6_and_d8(Emulator* emu)
{
    u8 data = read_d8(emu);
    emu->tick(1);
    emu->cpu.a = and_8(emu, emu->cpu.a, data);
    emu->tick(1);
}
```

0xA6 AND (HL)

该指令读取HL寄存器所存储的地址的值，并将其与寄存器A的结果做按位与操作，存储在寄存器A中，并按照与上述AND指令相同的规则更新CPU标志位。

```
///! AND (HL) : Performs bitwise AND on A and data pointed by HL.
void xa6_and_mhl(Emulator* emu)
{
    u8 data = emu->bus_read(emu->cpu.hl());
    emu->tick(1);
    emu->cpu.a = and_8(emu, emu->cpu.a, data);
    emu->tick(1);
}
```

XOR指令

XOR指令执行按位异或操作，并将结果存储在指定的寄存器中。所有的XOR指令均操作8位无符号整数，且其中一个操作数均为A寄存器。XOR指令会根据下列规则设置CPU寄存器：

- 1. 如果运算后的值为0，则设置Z为1，否则设置为0。
- 2. N标志位固定设置为0。
- 3. H标志位固定设置为0。
- 4. C标志位固定设置为0。

为了避免编写重复代码，我们可以将XOR操作作为一个单独的辅助函数实现：

```
inline u8 xor_8(Emulator* emu, u8 v1, u8 v2)
{
    u8 r = v1 ^ v2;
    set_zero_flag(emu, r);
    emu->cpu.reset_fn();
    emu->cpu.reset_fh();
    emu->cpu.reset_fc();
    return r;
}
```

然后就可以实现具体的指令。以0xA8 XOR B为例：

```
///! XOR B : Performs bitwise XOR on A and B.
void xa8_xor_b(Emulator* emu)
{
    emu->cpu.a = xor_8(emu, emu->cpu.a, emu->cpu.b);
    emu->tick(1);
}
```

下表列出了类似的XOR指令，这些指令都可以使用上述的0xA8指令通过修改寄存器的方式实

现：

操作码	指令	寄存器1	寄存器2	结果寄存器
0xA8	XOR B	A	B	A
0xA9	XOR C	A	C	A
0xAA	XOR D	A	D	A
0xAB	XOR E	A	E	A
0xAC	XOR H	A	H	A
0xAD	XOR L	A	L	A
0xAF	XOR A	A	A	A

除此之外，我们还有一些特殊的XOR指令：

0xEE XOR, d8

该指令读取8位立即数，并将其与寄存器A的结果做按位异或操作，存储在寄存器A中，并按照与上述XOR指令相同的规则更新CPU标志位。

```
/// XOR d8 : Performs bitwise XOR between A and 8-bit immediate data.
void xee_xor_d8(Emulator* emu)
{
    u8 data = read_d8(emu);
    emu->tick(1);
    emu->cpu.a = xor_8(emu, emu->cpu.a, data);
    emu->tick(1);
}
```

0xAE XOR (HL)

该指令读取HL寄存器所存储的地址的值，并将其与寄存器A的结果做按位异或操作，存储在寄存器A中，并按照与上述XOR指令相同的规则更新CPU标志位。

```
/// XOR (HL) : Performs bitwise XOR on A and data pointed by HL.
void xae_xor_mhl(Emulator* emu)
{
    u8 data = emu->bus_read(emu->cpu.hl());
    emu->tick(1);
    emu->cpu.a = xor_8(emu, emu->cpu.a, data);
    emu->tick(1);
}
```

OR指令

OR指令执行按位或操作，并将结果存储在指定的寄存器中。所有的OR指令均操作8位无符号整数，且其中一个操作数均为A寄存器。OR指令会根据下列规则设置CPU寄存器：

- 1. 如果运算后的值为0，则设置Z为1，否则设置为0。
- 2. N标志位固定设置为0。
- 3. H标志位固定设置为0。
- 4. C标志位固定设置为0。

为了避免编写重复代码，我们可以将OR操作作为一个单独的辅助函数实现：

```
inline u8 or_8(Emulator* emu, u8 v1, u8 v2)
{
    u8 r = v1 | v2;
    set_zero_flag(emu, r);
    emu->cpu.reset_fn();
    emu->cpu.reset_fh();
    emu->cpu.reset_fc();
    return r;
}
```

然后就可以实现具体的指令。以0xB0 OR B为例：

```
/// OR B : Performs bitwise OR on A and B.
void xb0_or_b(Emulator* emu)
{
    emu->cpu.a = or_8(emu, emu->cpu.a, emu->cpu.b);
    emu->tick(1);
}
```

下表列出了类似的OR指令，这些指令都可以使用上述的0xB0指令通过修改寄存器的方式实现：

操作码	指令	寄存器1	寄存器2	结果寄存器
0xB0	OR B	A	B	A
0xB1	OR C	A	C	A
0xB2	OR D	A	D	A
0xB3	OR E	A	E	A
0xB4	OR H	A	H	A
0xB5	OR L	A	L	A
0xB7	OR A	A	A	A

除此之外，我们还有一些特殊的OR指令：

0xF6 OR, d8

该指令读取8位立即数，并将其与寄存器A的结果做按位或操作，存储在寄存器A中，并按照与上述OR指令相同的规则更新CPU标志位。

```

///! OR d8 : Performs bitwise OR between A and 8-bit immediate data.
void xf6_or_d8(Emulator* emu)
{
    u8 data = read_d8(emu);
    emu->tick(1);
    emu->cpu.a = or_8(emu, emu->cpu.a, data);
    emu->tick(1);
}
```

0xB6 OR (HL)

该指令读取HL寄存器所存储的地址的值，并将其与寄存器A的结果做按位或操作，存储在寄存器A中，并按照与上述OR指令相同的规则更新CPU标志位。

```

///! OR (HL) : Performs bitwise OR on A and data pointed by HL.
void xb6_or_mhl(Emulator* emu)
{
    u8 data = emu->bus_read(emu->cpu.hl());
    emu->tick(1);
    emu->cpu.a = or_8(emu, emu->cpu.a, data);
    emu->tick(1);
}
```

CPL指令

CPL（complement）指令反转A寄存器的所有位的值，并将标志位N和H设置为1。

```

///! CPL : Takes complements of A (inverts every bit of A).
void x2f_cpl(Emulator* emu)
{
    emu->cpu.a = emu->cpu.a ^ 0xFF;
    emu->cpu.set_fn();
    emu->cpu.set_fh();
    emu->tick(1);
}
```

DAA指令

DAA（decimal adjust A）指令在ADD、ADC、SUB和SBC指令之后立即调用，用于将上一步的运算结果（从寄存器A中读取）转换为BCD（binary coded decimal）码表示，并保存在寄存器A中。BCD是一种对数字的编码方式，在GameBoy中，每一个数字具有一个4位的BCD编码，如下表所示：

数字	编码
0	0000b
1	0001b
2	0010b
3	0011b
4	0100b
5	0101b

6	0110b
7	0111b
8	1000b
9	1001b

显然，一个字节可以用于存储2个BCD编码的数字，且其在十六进制下的写法正好就是其编码的数字（例如，0x69对应的数字正好就是6和9）。如果我们希望对两个使用BCD编码的整数进行加法和减法操作，则我们可以先使用ADD、ADC、SUB、SBC指令按照常规步骤进行运算，然后调用DAA指令，将结果翻译为BCD码表示的数据。

举个例子，假设A=0x45，B=0x38；

此时我们调用ADD A, B，则A=0x7D，N=0；

然后我们调用DAA，其在检查A和N的值后，会执行ADD A, 0x06，此时A=0x83，正好是45+38在BCD编码下的结果。

同样，假设A=0x83，B=0x38；

此时我们调用SUB B，则A=0x4B，N=1

然后我们调用DAA，其在检查A和N的值后，会执行ADD A, 0xFA，此时A=0x45，正好是83-38在BCD编码下的结果。

DAA指令会通过标志位C来指示运算是否发生溢出：

- 在加法运算中，如果运算后的结果大于99（0x99），则结果的低两位会保存在寄存器A中，同时将标志位C设置为1，否则设置为0。例如：90+80=170，因此A寄存器的值最终为0x70，同时标志位C置1。
- 在减法运算中，如果运算后的结果小于0，则结果会加上100后保存在寄存器A中，同时将标志位C设置为1，否则设置为0。例如：5-21=-16，加上100后为84，因此A寄存器的值最终为0x84，同时标志位C置1。

同时DAA指令会固定将标志位H设置为0，并根据运算后寄存器A是否为0来设置标志位Z。

事实上，对于任意两个有效BCD编码值的加法和减法得到的结果，我们都可以通过对其加上一个固定的数值来将其转换为正确的BCD编码结果，且添加的该数值只和运算结果（A寄存器和H、C标志位获取），以及运算类型（N标志位获得）有关。因此DAA指令实际上是一个映射表，当N为0（加法操作）的情况下，需要增加的值得C标志位的操作如下：

C标志位	H标志位	A高四位	A低四位	A的值增加	设置C的值为
0	0	0-9	0-9	0x00	0
0	0	0-8	A-F	0x06	0
0	1	0-9	0-3	0x06	0
0	0	A-F	0-9	0x60	1
0	0	9-F	A-F	0x66	1
0	1	A-F	0-3	0x66	1
1	0	0-2	0-9	0x60	1
1	0	0-2	A-F	0x66	1
1	1	0-3	0-3	0x66	1

当N为1（减法操作）的情况下，需要增加的值得C标志位的操作如下：

C标志位	H标志位	A高四位	A低四位	A的值增加	设置C的值为
0	0	0-9	0-9	0x00	0
0	1	0-8	6-F	0xFA	0
1	0	7-F	0-9	0xA0	1
1	1	6-F	6-F	0x9A	1

在实现该指令时，我们只需要判断N、C、H标志位以及A的值，并根据条件增加寄存器A的值以及设置C的值就行：

```

    ///! DAA : Decimal adjust for A.
    void x27_daa(Emulator* emu)
    {
        if(emu->cpu.fn())
        {
            if(emu->cpu.fc())
            {
                if(emu->cpu.fh()) emu->cpu.a += 0x9A;
            }
        }
    }

```

```
        else emu->cpu.a += 0xA0;
    }
    else
    {
        if(emu->cpu.fh()) emu->cpu.a += 0xFA;
    }
}
else
{
    if(emu->cpu.fc() || (emu->cpu.a > 0x99))
    {
        if(emu->cpu.fh() || ((emu->cpu.a & 0x0F) > 0x09)) emu->cpu.a += 0x01;
        else emu->cpu.a += 0x60;
        emu->cpu.set_fc();
    }
    else
    {
        if(emu->cpu.fh() || ((emu->cpu.a & 0x0F) > 0x09)) emu->cpu.a += 0x01;
    }
}
set_zero_flag(emu, emu->cpu.a);
emu->cpu.reset_fh();
emu->tick(1);
}
```

SCF指令

SCF (Set C Flag) 指令将标志位C设置为1，同时将N和H标志位设置为0。

```
///! SCF : Sets C to 1.
void x37_scf(Emulator* emu)
{
    emu->cpu.reset_fn();
    emu->cpu.reset_fh();
    emu->cpu.set_fc();
    emu->tick(1);
}
```

CCF指令

CCF (Clear C Flag) 指令反转标志位C的值，同时将N和H标志位设置为0。可以通过SCF+CCF的指令组合将标志位C设置为0。

```
///! CCF : Flips C.
void x3f_ccf(Emulator* emu)
{
    emu->cpu.reset_fn();
    emu->cpu.reset_fh();
    if(emu->cpu.fc())
    {
        emu->cpu.reset_fc();
    }
    else
    {
        emu->cpu.set_fc();
    }
    emu->tick(1);
}
```

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,  x03_inc_bc, x04_inc_b,   x05_de
    nullptr,     x11_ld_de_d16, x12_ld_mde_a,  x13_inc_de, x14_inc_d,   x15_de
    x20_jr_nz_r8, x21_ld_hl_d16, x22_ldi_mhl_a, x23_inc_hl, x24_inc_h,   x25_de
    x30_jr_nc_r8, x31_ld_sp_d16, x32_ldd_mhl_a, x33_inc_sp, x34_inc_mhl, x35_de
    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_
    x80_add_a_b,  x81_add_a_c,  x82_add_a_d,  x83_add_a_e,  x84_add_a_h,  x85_add_a
    x90_sub_b,    x91_sub_c,    x92_sub_d,    x93_sub_e,    x94_sub_h,    x95_sub_l, x96_sub_r
```



```
    xa0_and_b, xa1_and_c, xa2_and_d, xa3_and_e, xa4_and_h, xa5_and_l, xa6_and_r
    xb0_or_b,  xb1_or_c,  xb2_or_d,  xb3_or_e,  xb4_or_h,  xb5_or_l,  xb6_or_ml
    xc0_ret_nz,   xc1_pop_bc, xc2_jp_nz_a16, xc3_jp_a16, xc4_call_nz_a16, xc5_l
    xd0_ret_nc,   xd1_pop_de, xd2_jp_nc_a16, nullptr,   xd4_call_nc_a16, xd5_l
    xe0_ldh_m8_a, xe1_pop_hl, xe2_ld_mc_a,   nullptr,   nullptr,           xe5_l
    xf0_ldh_a_m8, xf1_pop_af, xf2_ld_a_mc,   nullptr,   nullptr,           xf5_l
};
```

以上就是本章节的全部内容了。至此我们已经实现了GameBoy CPU的大部分指令，在下一章中，我们将实现剩下的一些特殊指令，以及实现CPU的中断机制。那么，下一章见！

銀葉吉祥：从零开始实现GameBoy模拟器 #5 CB指令、特殊指令、中断

17 赞同 · 2 评论 文章

编辑于 2024-02-02 21:32 · IP 属地上海

游戏机模拟器

中央处理器 (CPU)



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



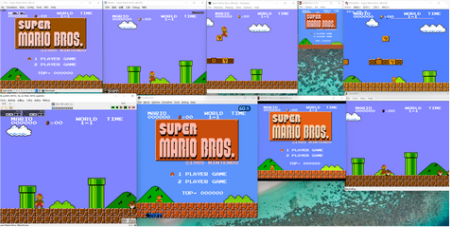
吉祥的游戏制作笔记
游戏制作中的技术、艺术和设计灵感记录

推荐阅读

玩模拟器的掌机神器-RG

想要原汁原味玩各种老游戏当然是烧原装主机，上模拟屏。原装主机如果运气好，也许可以买到箱说全三码合一的二十多年的全新珍藏版，不过除了价格贵以外还很占地方，如果书房没有多余地方放第...

韦易笑



FC/NES模拟器选择指南

2呵呵2



介绍几个安卓模拟器

软件小编

发表于软件探索

永恒经典的索尼掌机！PSP模拟器深度教程PC篇：模拟器系

PSP是一代经典掌机，如今在手机、PC上已经能比较完美的模拟器了，相对于手机模拟的便捷性，因为PC主机的性能更强大，PC模拟可以带来更好的画质和运行帧数，一些原先在PSP模糊锯齿严重的游
乌托邦游戏 发表于乌托邦游戏

▲ 赞同 22



● 添加评论

🔗 分享

❤️ 喜欢

★ 收藏

📄 申请转载

...

