PA1 实验报告

211180074 彭安澜

2024 年 3 月 24 日

PA1-1 数据的表示和存取

实验代码

在这一节中我们需要在 `pa_nju/nemu/include/cpu/reg.h` 中修改用于模拟CPU中通用寄存器部分的代码，按照合理的方式设置结构体变量，具体代码如下所示：

```
// define the structure of registers
typedef struct
{
        // general purpose registers
        union
        {
                union
                {
                        union
                        {
                                uint32_t _32;
                                uint16_t _16;
                                uint8_t _8[2];
                        };
                        uint32_t val;
                } gpr[8];
                struct
                { // do not change the order of the registers
                        uint32_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
                };
                // 这个struct被嵌套在了一个union内，而不是在全局命名空间中定义的，因此它的作用域是受限的。并且struct是匿名的，因此这些成员的名称就是它们的标识符，可以在定义它们的union的作用域内直接使用。
        };
        // 同理，这个union也是匿名的，最后效果就是可以直接CPU_STATE.gpr[0].val或者CPU_STATE.eax来使用，
        // 尽管eax是被多层嵌套的

        // EIP
        uint32_t eip;

        // EFLAGS
        union {
                struct
                {
                        uint32_t CF : 1;
                        uint32_t dummy0 : 1;
                        uint32_t PF : 1;
                        uint32_t dummy1 : 1;
                        uint32_t AF : 1;
                        uint32_t dummy2 : 1;
                        uint32_t ZF : 1;
                        uint32_t SF : 1;
                        uint32_t TF : 1;
                        uint32_t IF : 1;
                        uint32_t DF : 1;
                        uint32_t OF : 1;
                        uint32_t OLIP : 2;
                        uint32_t NT : 1;
                        uint32_t dummy3 : 1;
                        uint32_t RF : 1;
                        uint32_t VM : 1;
                        uint32_t dummy4 : 14;
                };
                uint32_t val;
        } eflags;
```

```
#ifdef IA32_SEG
        GDTR gdtr; // GDTR, todo: define type GDTR
        // segment registers, todo: define type SegReg
        union {
                SegReg segReg[6];
                struct
                {
                        SegReg es, cs, ss, ds, fs, gs;
                };
        };
        // control registers, todo: define type CR0
        CR0 cr0;
#else
        uint8_t dummy_seg[142]; // make __ref_ instructions safe to use
#endif
#ifdef IA32_PAGE
        // control registers, todo: define type CR3
        CR3 cr3;
#else
        uint8_t dummy_page[4];
#endif

#ifdef IA32_INTR
        // interrupt
        IDTR idtr; // IDTR, todo: define type IDTR
        uint8_t intr;
#else
        uint8_t dummy_intr[7];
#endif
} CPU_STATE;
```

我们重点关注通用寄存器部分的代码，这是一个联合体（union）类型的变量，包含gpr[8]和一个struct类型的变量总共两个变量，其中这个struct变量是匿名的，可以直接通过其中成员的变量名访问各个成员。gpr[8]是一个联合体数组，对其中每个联合体成员，又是由一个联合体和一个32位无符号类型的变量val组成，容易看出gpr[8]每个成员的长度应该是取最大变量长度，也就是32位。而最外围struct类型的变量中的每个成员也是32位无符号类型的变量，这就意味着struct类型中的八个成员变量，与gpr[8]中的元素是一一对应的。

根据union类型变量的定义，eax, ecx, edx, ebx, esp, ebp, esi, edi等价于gpr[0].val~gpr[8].val；而通过gpr[0]._32又可以访问eax寄存器的全部32位，通过gpr[0]._16可以访问eax寄存器的低16位，对应于ax寄存器，通过gpr[0]._8[0]、gpr[0]._8[1]可以分访问eax寄存器的低8位和次低8位，对应于al和ah寄存器，以此类推。

```
// general purpose registers
union
```

```
{
        union
        {
                union
                {
                        uint32_t _32;
                        uint16_t _16;
                        uint8_t _8[2];
                };
                uint32_t val;
        } gpr[8];
        struct
        { // do not change the order of the registers
                uint32_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
        };
        // 这个struct被嵌套在了一个union内，而不是在全局命名空间中定义的，因此它的作用域是受限的。并且struct是匿名的，因此这些成员的名称就是它们的标识符，可以在定义它们的union的作用域内直接使用。
};
// 同理，这个union也是匿名的，最后效果就是可以直接CPU_STATE.gpr[0].val或者CPU_STATE.eax来使用，
// 尽管eax是被多层嵌套的
```

综上所述，我们设计了一个CPU_STATE类型的结构体，这个结构体模拟了CPU的全部寄存器信息，包括通用寄存器和标志寄存器（eflags）。在调用时，我们可以使用CPU.*（通用寄存器名称）来指代某一通用寄存器，也可以使用CPU.gpr[*].val（通用寄存器对应序号）来进行指代，进一步我们还可以使用CPU.gpr[*]._32 / _16 / _8[0] / _08[1]来分别访问寄存器特定位数的数值；同时，我们可以使用CPU.eflags. *（标志寄存器）来指代某一标志寄存器。

运行结果

1. 修改 `CPU_STATE`结构体中的通用寄存器结构体；
2. `make clean`后使用 `make`编译项目；
3. 在项目根目录通过 `make test_pa-1`命令执行并通过 `reg_test()`测试用例。

```
./nemu/nemu --test-reg
NEMU execute built-in tests
reg_test()      pass
```

实验思考题

1. **C语言中的 `struct`和 `union`关键字都是什么含义，寄存器结构体的参考实现为什么把部分 `struct`改成了 `union`？**

   `struct`关键字用于定义结构体，是一种复合数据类型，它允许将多个不同类型的变量组合到一个单独的类型名称下。结构体中的每个成员可以是不同的数据类型，包括其他的结构体或者数组。结构体的每个成员都有自己的内存空间，成员之间不会相互影响。

   `union`关键字用于定义联合体，也是一种复合数据类型。与结构体不同的是，联合体中的所有成员共享同一块内存空间，这个空间的大小等于联合体中最大成员的大小。任何时候只有一个成员可以有值，对一个成员的更新会覆盖其他成员的值。

   在寄存器结构体的参考实现中将部分struct改成union是结合CPU中寄存器的真实结构而做出的。具体来说将第一个struct改为union，是因为CPU中的gpr[8]（general purpose register）就对应eax, ecx, edx, ebx, esp, ebp, esi, edi总共8个寄存器，两者指代同一个东西，值应该是始终相同的，所以用union；第二个struct改成union，是因为一个寄存器只会有一个32位的值，因此gpr[8]内的实际数据长度就应该只有32位，除了寄存器的值val，用另一联合体表示寄存器不同位数的值；第三个struct改成union，则是延续第二个struct改成union，val和第三个union同指向一个内存，而第三个union内部又有_32、 _16、 _8[2]三个变量指向同一个内存（即union内存），也就是说通过_32、 _16、 _8[2]可以访问各个寄存器的值val的全部32位、低16位、低8位、次低8位，也就分别模拟了通用寄存器中的eax、ax、al、ah等寄存器。

PA1-2 整数的表示和运算

实验代码

在这一节中我们需要在 `pa_nju/nemu/src/cpu/alu.c` 中编写用于模拟CPU中算术逻辑单元（alu）部分的代码，对于每种运算函数的编写，基本可以分为1. 得到运算结果；2. 设置标志寄存器，两步来完成。

因此在编写一个alu中的新函数时候，我们在手册上只要重点关注两点：1. 运算是如何定义的；2. 标志寄存器行为是如何定义的，然后完成函数编写即可。

运算的定义在i386手册17.2.2.11 Instruction Set Detail一节，针对每种运算，手册都清楚地给出了Operation、Description、Flags Affected，再结合函数预先设定好的形参和返回值，容易编程得到正确的返回结果。

同样在i386手册的Appendix C一节可以找到各种标志寄存器 CF、PF、ZF、SF、AF、OF 在运算中如何变化的定义，这其中有些寄存器只有比较模糊的定义，如 CF、OF等，因为它们的设定不单纯依赖于结果，还取决于具体的运算类型；而有些寄存器则定义清晰，针对它们的设置情况可以简单由运算结果来设定。另外对于各种运算，会影响到哪些标志寄存器，在Appendix C一节的后一页同样明确说明，根据实验手册，我们无视 AF寄存器的设置。

下面就是进行编程：

OF、SF、PF标志寄存器设置

在开始编写函数前，我们可以先基于Appendix C一节，对一些方便置位的标志寄存器的设置方法用函数封装，提升代码复用性，注意到这些寄存器的设置方法是比较简单的，较复杂的寄存器的设置方法需要在后面根据运算类型单独编程。

```
void set_PF(uint32_t result)
{
        // parity of low-order eight bits
        int cnt = 0;
        for (int i = 0; i < 8; i++)
        {
                if ((result >> i) & 0x00000001)
                        cnt++;
        }
        // even number of 1 bits; cleared otherwise
        cpu.eflags.PF = (cnt % 2 == 0);
        return;
}

void set_ZF(uint32_t result, size_t data_size)
{
        // clear data in high part first and then compare the result with 0
        result &= (0xFFFFFFFF >> (32 - data_size));
        // Set if result is zero; cleared otherwise.
        cpu.eflags.ZF = (result == 0);
        return;
}

void set_SF(uint32_t result, size_t data_size)
{
        // clear data in high part first
        result &= (0xFFFFFFFF >> (32 - data_size));
        // Set equal to the high-order bit of the result
        cpu.eflags.SF = (result >> (data_size - 1)) & 0x00000001;
        return;
}
```

这里我们对输入的操作数高位（32位中高于date_size的位）置零，然后按照手册定义（分别是低8位奇偶性、是否为0、正负性）对寄存器完成设置，不好设置的 CF、OF寄存器将在后续结合具体运算再设计。

加减运算函数编写

首先是无进位加法函数：

```
// DEST ← DEST + SRC
// OF/SF/ZF/PF/CF Modified (ignore AF)
uint32_t alu_add(uint32_t src, uint32_t dest, size_t data_size)
{
#ifdef NEMU_REF_ALU
        return __ref_alu_add(src, dest, data_size);
#else
        // When the real size of the date does not correspond to the data_size,
        // the high part should be ignored (Don't leave things undefined!!!)-> this
has influnce to the eflags
        uint32_t result = (dest & (0xFFFFFFFF >> (32 - data_size))) + (src &
(0xFFFFFFFF >> (32 - data_size)));
        // change eflags
        set_CF_add(result, src, data_size);
        set_PF(result);
        set_ZF(result, data_size);
        set_SF(result, data_size);
        set_OF_adx(result, src, dest, data_size);
        // to emulate alu, must cut the result to the right size; high part should
only be used for eflags
        return result & (0xFFFFFFFF >> (32 - data_size));

#endif
}
```

需要注意的点就是对输出的操作数，要按照data_size做高位清零操作，同时对返回的结果也要做高位清零；另外这里用到了专为add操作中CF寄存器和adx操作（即add和adc）中OF寄存器的设置的函数，定义如下：

```
void set_CF_add(uint32_t result, uint32_t src, size_t data_size)
{
        // clear data in high part first and then compare the result with src
        result &= (0xFFFFFFFF >> (32 - data_size));
        src &= (0xFFFFFFFF >> (32 - data_size));
        // 是否一定需要做位扩展后再进行比较？？？
        cpu.eflags.CF = result < src;
        // this won't work for adc, need further modification
        return;
}
```

原理可以结合时钟来理解，在原有位置处再旋转不到一圈，在不溢出时一定比原来值更大，溢出时咋一定比原来值小。

```
void set_OF_adx(uint32_t result, uint32_t src, uint32_t dest, size_t data_size)
{
        // this should also work for adc
        // 多了一个1，只在result为最大正数才会溢出，则src和dest必为正号
        // 此时仍然在判定范围内(注意到0也还是正数)
        // clear data in high part first
        result &= (0xFFFFFFFF >> (32 - data_size));
        src &= (0xFFFFFFFF >> (32 - data_size));
        dest &= (0xFFFFFFFF >> (32 - data_size));
        // get the sign of src, dest and result
        int sign_src = (src >> (data_size - 1)) & 0x00000001;
        int sign_dest = (dest >> (data_size - 1)) & 0x00000001;
        int sign_result = (result >> (data_size - 1)) & 0x00000001;    if (sign_src
== sign_dest)
        {
                if (sign_result != sign_src)
                        cpu.eflags.OF = 1;
                else
                        cpu.eflags.OF = 0;
        }
        else
        {
                cpu.eflags.OF = 0;
        }
        return;
}
```

原理从运算结果的合理性来看，同号相加出现异号就是溢出。

然后是有进位加法函数：

```
// DEST ← DEST + SRC + CF
// OF/SF/ZF/PF/CF Modified (ignore AF)
uint32_t alu_adc(uint32_t src, uint32_t dest, size_t data_size)
{
#ifdef NEMU_REF_ALU
        return __ref_alu_adc(src, dest, data_size);
#else
        // printf("\e[0;31mPlease implement me at alu.c\e[0m\n");
        // fflush(stdout);
        // assert(0);
        // return 0;    // result
        uint32_t result = (dest & (0xFFFFFFFF >> (32 - data_size))) + (src &
(0xFFFFFFFF >> (32 - data_size))) + cpu.eflags.CF;
        // change eflags
        set_CF_adc(result, src, data_size);
        set_PF(result);
```

```
        set_ZF(result, data_size);
        set_SF(result, data_size);
        set_OF_adx(result, src, dest, data_size);
        return result & (0xFFFFFFFF >> (32 - data_size));
#endif
}
```

和无进位加法函数基本一致，但不同的是在进位设置上有所不同——在CF不为0时，原有位置处再加上加数和进位后最大可以转一整圈，这将导致溢出时结果也不比原加数小，需要分情况讨论。

```
void set_CF_adc(uint32_t result, uint32_t src, size_t data_size)
{
        if (cpu.eflags.CF == 0)
        {
                set_CF_add(result, src, data_size);
                return;
        }
        else
        {
                // clear data in high part first and then compare the result with
src
                result &= (0xFFFFFFFF >> (32 - data_size));
                src &= (0xFFFFFFFF >> (32 - data_size));
                // result couldn't equal to src if carry does't happen, as there is
an extra 1
                cpu.eflags.CF = result <= src;
                return;
        }
}
```

接着是无借位减法函数：

```
// IF SRC is a byte and DEST is a word or dword
// THEN DEST ← DEST - Sign-extend(SRC)
// ELSE DEST ← DEST - SRC
// OF/SF/ZF/PF/CF Modified (ignore AF)
uint32_t alu_sub(uint32_t src, uint32_t dest, size_t data_size)
{
#ifdef NEMU_REF_ALU
        return __ref_alu_sub(src, dest, data_size);
#else
        // result
        uint32_t result = (dest & (0xFFFFFFFF >> (32 - data_size))) - (src &
(0xFFFFFFFF >> (32 - data_size)));
        // change eflags
        set_CF_sub(src, dest, data_size);
        set_PF(result);
        set_ZF(result, data_size);
```

```
        set_SF(result, data_size);
        set_OF_sux(result, src, dest, data_size);
        return result & (0xFFFFFFFF >> (32 - data_size));
#endif
}
```

对应的CF设置函数，原理同add：

```
void set_CF_sub(uint32_t src, uint32_t dest, size_t data_size)
{
        // clear data in high part first and then compare the dest with src (dest -
src)
        src &= (0xFFFFFFFF >> (32 - data_size));
        dest &= (0xFFFFFFFF >> (32 - data_size));
        // Set if src > dest; cleared otherwise.
        cpu.eflags.CF = src > dest;
        return;
}
```

对应的OF设置函数，原理同adx：

```
void set_OF_sux(uint32_t result, uint32_t src, uint32_t dest, size_t data_size)
{
        // this should also work for sbb
        // 多减一个1，只在result为最小负数才会溢出，则src和dest必为一正一负
        // 此时仍然在判定范围内(注意到0也还是正数)
        // clear data in high part first
        result &= (0xFFFFFFFF >> (32 - data_size));
        src &= (0xFFFFFFFF >> (32 - data_size));
        dest &= (0xFFFFFFFF >> (32 - data_size));
        // get the sign of src, dest and result
        int sign_src = (src >> (data_size - 1)) & 0x00000001;
        int sign_dest = (dest >> (data_size - 1)) & 0x00000001;
        int sign_result = (result >> (data_size - 1)) & 0x00000001;     if
(sign_src != sign_dest) // different sign
        {
                if (sign_result != sign_dest)
                        cpu.eflags.OF = 1;
                else
                        cpu.eflags.OF = 0;
        }
        else
        {
                cpu.eflags.OF = 0;
        }    return;
}
```

最后是有借位减法，与sub类似，只要对CF设置函数做出调整：

```
// IF SRC is a byte and DEST is a word or dword
// THEN DEST = DEST - (SignExtend(SRC) + CF)
// ELSE DEST ← DEST - (SRC + CF);
// OF/SF/ZF/PF/CF Modified (ignore AF)
uint32_t alu_sbb(uint32_t src, uint32_t dest, size_t data_size)
{
#ifdef NEMU_REF_ALU
        return __ref_alu_sbb(src, dest, data_size);
#else
        // printf("\e[0;31mPlease implement me at alu.c\e[0m\n");
        // fflush(stdout);
        // assert(0);
        // return 0;    // result
        uint32_t result = (dest & (0xFFFFFFFF >> (32 - data_size))) - ((src &
(0xFFFFFFFF >> (32 - data_size))) + cpu.eflags.CF);
        // change eflags
        set_CF_sbb(src, dest, data_size);
        set_PF(result);
        set_ZF(result, data_size);
        set_SF(result, data_size);
        set_OF_sux(result, src, dest, data_size);       return result & (0xFFFFFFFF
>> (32 - data_size));#endif
}
```

CF设置函数：

```
void set_CF_sbb(uint32_t src, uint32_t dest, size_t data_size)
{
        if (cpu.eflags.CF == 0)
        {
                set_CF_sub(src, dest, data_size);
                return;
        }
        else
        {
                // clear data in high part first and then compare the dest with src
(dest - src)
                src &= (0xFFFFFFFF >> (32 - data_size));
                dest &= (0xFFFFFFFF >> (32 - data_size));
                // Set if src >= dest; cleared otherwise.
                cpu.eflags.CF = src >= dest;
                return; }    // clear data in high part first and then compare the
dest with src (dest - src)
        src &= (0xFFFFFFFF >> (32 - data_size));
        dest &= (0xFFFFFFFF >> (32 - data_size));
        // Set if src > dest; cleared otherwise.
        cpu.eflags.CF = src >= dest;
        return;
}
```

乘除运算

乘除运算则按有符号和无符号分为总共四个函数，实现不复杂，涉及的标志寄存器也较少，但规则比较复杂，需要对操作数进行讨论：

mul指令的实现如下所示：

```
// IF byte-size operation
// THEN AX ← AL * r/m8 (AX contains both AL and AH)
// ELSE (* word or doubleword operation *)
// IF OperandSize = 16
// THEN DX:AX ← AX * r/m16
// ELSE (* OperandSize = 32 *)
// EDX:EAX ← EAX * r/m32
// FI;
// 简单总结就是8位乘法要16位存，16位乘法要32位存（DX高16，AX低16），32位乘法要64位存（EDX高32，EAX低32）
// OF/CF Modified ; SF/ZF/PF/AF/CF Undefined
// Undefined means that the value of the flag is not predictable after the operation
uint64_t alu_mul(uint32_t src, uint32_t dest, size_t data_size)
{
#ifdef NEMU_REF_ALU
        return __ref_alu_mul(src, dest, data_size);
#else
        uint64_t result = 0;

        // optimization for 0 and 1
        // operand has 0
        if (src == 0 || dest == 0)
        {
                cpu.eflags.CF = 0;
                cpu.eflags.OF = 0;
                return result;
        }
        // operand has 1
        if (src == 1)
        {
                cpu.eflags.CF = 0;
                cpu.eflags.OF = 0;
                return dest;
        }
        if (dest == 1)
        {
                cpu.eflags.CF = 0;
                cpu.eflags.OF = 0;
                return src;
        }
        // operand has 2 -> CF & OF are complex -> ignore

        switch (data_size)
        {
                case 8:{
                        // result
```

```
                        result = ((src & 0x000000FF) * (dest & 0x000000FF)) &
0x000000000000FFFF;
                        // eflags (OF CF are set 1 when high part is not 0)
                        if (result >>8) // AH is not 0
                                {cpu.eflags.CF = 1;
                                cpu.eflags.OF = 1;}
                        else
                                {cpu.eflags.CF = 0;
                                cpu.eflags.OF = 0;}
                        return result;
                }
                case 16:{
                        // result
                        result = ((src & 0x0000FFFF) * (dest & 0x0000FFFF)) &
0x00000000FFFFFFFF;
                        // eflags (OF CF are set 1 when high part is not 0)
                        if (result >>16) // DX is not 0
                                {cpu.eflags.CF = 1;
                                cpu.eflags.OF = 1;}
                        else
                                {cpu.eflags.CF = 0;
                                cpu.eflags.OF = 0;}
                        return result;
                }
                case 32:{
                        // result
                        result = ((uint64_t)src * (uint64_t)dest);
                        // eflags (OF CF are set 1 when high part is not 0)
                        if (result >>32) // EDX is not 0
                                {cpu.eflags.CF = 1;
                                cpu.eflags.OF = 1;}
                        else
                                {cpu.eflags.CF = 0;
                                cpu.eflags.OF = 0;}
                        return result;
                }
                default:
                        return -1;

        }

#endif
}
```

编写该函数时为一些特殊操作数（操作数有1或0）提供了优化。

---

imul指令的实现如下所示：

```
// result ← multiplicand * multiplier;
// IMUL clears the overflow and carry flags under the following conditions:
// Instruction                  Form Condition for Clearing CF and OF
// r/m8                                 AL = sign-extend of AL to 16 bits
// r/m16                                AX = sign-extend of AX to 32 bits
```

```
// r/m32                                EDX:EAX = sign-extend of EAX to 32 bits
// r16,r/m16                    Result exactly fits within r16
// r/32,r/m32                   Result exactly fits within r32
// r16,r/m16,imm16              Result exactly fits within r16
// r32,r/m32,imm32              Result exactly fits within r32
// 计算结果同MUL一样，x位乘，2x位输出结果
// 不知道指令具体形式，简单按照几位乘，结果不超过几位就clear来设计
// 如果没有发生以上情形，是否也不用手动给OF和CF赋值为1？？？
int64_t alu_imul(int32_t src, int32_t dest, size_t data_size)
{
#ifdef NEMU_REF_ALU
        return __ref_alu_imul(src, dest, data_size);
#else
        src = sign_ext(src, data_size);
        dest = sign_ext(dest, data_size);
        int64_t result = 0;
        // optimization for 0 and 1
        // operand has 0
        if (src == 0 || dest == 0)
        {
                cpu.eflags.CF = 0;
                cpu.eflags.OF = 0;
                return result;
        }
        // operand has 1
        if (src == 1)
        {
                cpu.eflags.CF = 0;
                cpu.eflags.OF = 0;
                return dest;
        }
        if (dest == 1)
        {
                cpu.eflags.CF = 0;
                cpu.eflags.OF = 0;
                return src;
        }
        // src and dest are passed in the form according to the data_size, but all
treated as 32 bits
        // so must be sign-extended to 32 bits first!!! src & dest could only to 32
bits multiplication

        result = (int64_t)src * (int64_t)dest;
        // 8位乘法用16位存，其余高位是置零还是不管？？？
        // 这里多的高位直接置零
        // result = result & (0xFFFFFFFFFFFFFFFF >> (64 - 2 * data_size));
        // 这里多的高位直接不管
        // OF和CF直接按照结果是否超过data_size位来判断
        switch (data_size)
        {
        case 8:
                {if (result == (int64_t)(int8_t)(result & 0xFF)) // 发生截断，高位全
不要，低位机器数不变
                {
                        cpu.eflags.CF = 0;
                        cpu.eflags.OF = 0;
                }
```

```
                else
                {
                        cpu.eflags.CF = 1;
                        cpu.eflags.OF = 1;
                }
                break;}
        case 16:
                {if (result == (int64_t)(int16_t)(result & 0xFFFF))
                {
                        cpu.eflags.CF = 0;
                        cpu.eflags.OF = 0;
                }
                else
                {
                        cpu.eflags.CF = 1;
                        cpu.eflags.OF = 1;
                }
                break;}
        case 32:
                {if (result == (int64_t)(int32_t)(result & 0xFFFFFFFF))
                {
                        cpu.eflags.CF = 0;
                        cpu.eflags.OF = 0;
                }
                else
                {
                        cpu.eflags.CF = 1;
                        cpu.eflags.OF = 1;
                }
                break;}
        default:
                break;
        }
        return result;
#endif
}
```

虽然乘法指令是两个32位乘得到一个64位的结果（存放到不同寄存器中去），但这里依然设计为去除高位上的结果再返回。

在编写除法函数前，需要先完成mod和imod函数：

```
uint32_t alu_mod(uint64_t src, uint64_t dest)
{
#ifdef NEMU_REF_ALU
        return __ref_alu_mod(src, dest);
#else
        uint32_t result = 0;
        if (src == 0)
        {
                printf("Divided by 0\n");
                fflush(stdout);
```

```
                assert(0);
                return 0;
        }
        if (dest == 0)
        {
                return 0;
        }
        if (src == 1)
        {
                return 0;
        }
        result = dest % src;
        return result;
#endif
}

int32_t alu_imod(int64_t src, int64_t dest)
{
#ifdef NEMU_REF_ALU
        return __ref_alu_imod(src, dest);
#else
        if (src == 0)
        {
                printf("Divided by 0\n");
                fflush(stdout);
                assert(0);
                return 0;
        }
        if (dest == 0)
        {
                return 0;
        }
        if (src == 1)
        {
                return 0;
        }
        int32_t result = dest % src;
        return result;
#endif
}
```

完成无符号除法函数，同样对一些特殊情况做了优化，无需设置任何标志位，但注意除0产生异常：

```
// result ← DEST / SRC; remainder ← DEST % SRC;
// no eflags modified
// need to implement alu_mod before testing
uint32_t alu_div(uint64_t src, uint64_t dest, size_t data_size)
{
#ifdef NEMU_REF_ALU
        return __ref_alu_div(src, dest, data_size);
#else   uint32_t result = 0;
        // optimization for 0 and 1
```

```
        // operand has 0
        if (src ==0)
        {
                printf("Divided by 0\n");
                fflush(stdout);
                assert(0);
                return 0;
        }
        if (dest == 0)
        {
                return 0;
        }
        // operand has 1
        if (src == 1)
        {
                return dest;
        }
        // result
        result = (uint32_t)(dest & (0xFFFFFFFFFFFFFFFF >> (64 - data_size))) / (src
& (0xFFFFFFFFFFFFFFFF >> (64 - data_size)));
        return result;
#endif
}
```

最后是有符号除法：

```
// need to implement alu_imod before testing
int32_t alu_idiv(int64_t src, int64_t dest, size_t data_size)
{
#ifdef NEMU_REF_ALU
        return __ref_alu_idiv(src, dest, data_size);
#else
        // 下面的转换仅适合32位及一下；实际上考虑到手册中说仅关心两个64位整数相除得到一个
32位整数的问题，因此这里不对传入的src和dest进行任何处理
        // src = (int64_t)sign_ext(src, data_size);
        // dest = (int64_t)sign_ext(dest, data_size);
        // optimization for 0 and 1
        // operand has 0
        if (src ==0)
        {
                printf("Divided by 0\n");
                fflush(stdout);
                assert(0);
                return 0;
        }
        if (dest == 0)
        {
                return 0;
        }
        // operand has 1
        if (src == 1)
        {
                return dest;
```

```
        }
        // result
        int32_t result = dest / src;
        return result;
#endif
}
```

逻辑运算

逻辑运算只需要实现and、xor、or三种，同样没太多技巧，按照手册完成即可：

```
// DEST ← DEST AND SRC;
// CF ← 0;
// OF ← 0;
// SF/ZF/PF Modified; CF/OF are set zero
uint32_t alu_and(uint32_t src, uint32_t dest, size_t data_size)
{
#ifdef NEMU_REF_ALU
        return __ref_alu_and(src, dest, data_size);
#else
        // result
        uint32_t result = (dest & (0xFFFFFFFF >> (32 - data_size))) & (src &
(0xFFFFFFFF >> (32 - data_size)));
        // change eflags
        cpu.eflags.CF = 0;
        cpu.eflags.OF = 0;
        set_PF(result);
        set_ZF(result, data_size);
        set_SF(result, data_size);
        return result;
#endif
}

// DEST ← LeftSRC XOR RightSRC
// CF ← 0
// OF ← 0
// SF/ZF/PF Modified; CF/OF are set zero; AF is undefined
uint32_t alu_xor(uint32_t src, uint32_t dest, size_t data_size)
{
#ifdef NEMU_REF_ALU
        return __ref_alu_xor(src, dest, data_size);
#else
        // result
        uint32_t result = (dest & (0xFFFFFFFF >> (32 - data_size))) ^ (src &
(0xFFFFFFFF >> (32 - data_size)));
        // change eflags
        cpu.eflags.CF = 0;
        cpu.eflags.OF = 0;
        set_PF(result);
        set_ZF(result, data_size);
        set_SF(result, data_size);
        return result;
#endif
```

```
}

// DEST ← DEST OR SRC;
// CF ← 0;
// OF ← 0
// SF/ZF/PF Modified; CF/OF are set zero
uint32_t alu_or(uint32_t src, uint32_t dest, size_t data_size)
{
#ifdef NEMU_REF_ALU
        return __ref_alu_or(src, dest, data_size);
#else
        // result
        uint32_t result = (dest & (0xFFFFFFFF >> (32 - data_size))) | (src &
(0xFFFFFFFF >> (32 - data_size)));
        // change eflags
        cpu.eflags.CF = 0;
        cpu.eflags.OF = 0;
        set_PF(result);
        set_ZF(result, data_size);
        set_SF(result, data_size);
        return result;
#endif
}
```

移位运算

首先是逻辑左移：

```
// 返回将 dest 左移 src 位后的结果， data_size 用于指明操作数长度（比特数）
// SF/ZF/PF/CF/OF modified (OF could be ignored)
// 特别说明：针对上面四个移位操作，约定只影响 dest操作数的低 data_size位，而不影响其高 32
- data_size位。标志位的设置根据结果的低 data_size位来设置。
// dest逻辑左移（补零）src位
// CF ← 最后一次移出dest的位（也就是高位）
uint32_t alu_shl(uint32_t src, uint32_t dest, size_t data_size)
{
#ifdef NEMU_REF_ALU
        return __ref_alu_shl(src, dest, data_size);
#else
        // specially for OF
        // For left shifts, the OF flag is set to the exclusive OR of the CF bit
(after the shift) and the most-significant bit (top two bits) of the result.
        if (src == 1)
        {
                // get the CF
                int CF = (dest >> (data_size - 1)) & 0x00000001;
                // get the top
                int top = (dest >> (data_size - 2)) & 0x00000001;
                cpu.eflags.OF = (CF != top);
        }
        // result
        // to save the high 32 - datasize bits
        // attention: C语言中,当移位操作数超过了其数据类型的位数时,行为是未定义的。
```

```
        // 根据实际检验情况，右移32位时候，结果竟然变成了全1！！！！  // test code
        // dest = sign_ext(dest, data_size);
        // printf("dest: 0x%x\n", dest);
        // printf("data_size: 0x%x\n", data_size);
        // printf("src: 0x%x\n", src);  uint32_t high_32_datasize_dest = dest &
((uint32_t)0xFFFFFFFF << data_size);
        if (data_size == 32)
                high_32_datasize_dest = 0;
        // only save the low datasize bits, to be calculated
        uint32_t low_detasize_result = 0;
        // optimization for src > data_size
        if (src > data_size)
        {
                // low_detasize_result = 0;
                cpu.eflags.CF = 0;
                cpu.eflags.PF = 1;
                cpu.eflags.ZF = 1;
                cpu.eflags.SF = 0;
                // return (high_32_datasize_dest + low_detasize_result);
                return (high_32_datasize_dest + low_detasize_result) & (0xFFFFFFFF
>> (32 - data_size));
                // 实际只返回截断后低date_size位？？？
        }
        else
        {
                low_detasize_result = ((dest & (0xFFFFFFFF >> (32 - data_size))) <<
src) & (0xFFFFFFFF >> (32 - data_size));
                // change eflags
                cpu.eflags.CF = (dest >> (data_size - src + 1 - 1)) & 0x00000001;
                set_PF(low_detasize_result);
                set_ZF(low_detasize_result, data_size);
                set_SF(low_detasize_result, data_size);
                // return (high_32_datasize_dest + low_detasize_result);
                // test code
                // printf("high_32_datasize_dest: 0x%x\n", high_32_datasize_dest);
                // printf("low_detasize_result: 0x%x\n", low_detasize_result);
return (high_32_datasize_dest + low_detasize_result) & (0xFFFFFFFF >> (32 -
data_size));
                // 实际只返回截断后低date_size位？？？
        }#endif
}
```

然后是算术左移：

```
// dest算术左移（补1或补0）src位
// CF ← 最后一次移出dest的位（也就是高位）
uint32_t alu_sal(uint32_t src, uint32_t dest, size_t data_size)
{
#ifdef NEMU_REF_ALU
        return __ref_alu_sal(src, dest, data_size);
#else
        // specially for OF
        // same to shl
```

```
        if (src == 1)
        {
                // get the CF
                int CF = (dest >> (data_size - 1)) & 0x00000001;
                // get the top
                int top = (dest >> (data_size - 2)) & 0x00000001;
                cpu.eflags.OF = (CF != top);
        }
        // result
        // to save the high 32 - datasize bits
        uint32_t high_32_datasize_dest = dest & (0xFFFFFFFF << data_size);
        if (data_size == 32)
                high_32_datasize_dest = 0;
        // only save the low datasize bits, to be calculated
        uint32_t low_detasize_result = 0;
        // optimization for src > data_size
        if (src > data_size)
        {
                // set low detasize bits according to sign
                if ((dest >> (data_size - 1)) & 0x00000001)
                {
                        low_detasize_result = 0xFFFFFFFF >> (32 - data_size);
                        cpu.eflags.CF = 1;
                        cpu.eflags.PF = 1;
                        cpu.eflags.ZF = 0;
                        cpu.eflags.SF = 1;
                }
                else
                {
                        low_detasize_result = 0;
                        cpu.eflags.CF = 0;
                        cpu.eflags.PF = 1;
                        cpu.eflags.ZF = 1;
                        cpu.eflags.SF = 0;
                }
                // return (high_32_datasize_dest + low_detasize_result);
                return (high_32_datasize_dest + low_detasize_result) & (0xFFFFFFFF
>> (32 - data_size));
                // 实际只返回截断后低date_size位? ? ?
        }
        else
        {
                // set low detasize bits according to sign
                // if ((dest >> (data_size - 1)) & 0x00000001)
                // {
                //      // 左移低位都补0! 无论算术 or 逻辑! ! ! !
                //      // 补位为1 (this is wrong! )
                //      low_detasize_result = ( ((dest & (0xFFFFFFFF >> (32 -
data_size))) << src) | (0xFFFFFFFF >> (32 - src)) ) & (0xFFFFFFFF >> (32 -
data_size));
                //      set_PF(low_detasize_result);
                //      set_ZF(low_detasize_result, data_size);
                //      set_SF(low_detasize_result, data_size);
                //      cpu.eflags.CF = dest >> (data_size - src) & 0x00000001;
                // }
                // else
                // {
```

```
                        // 补位为0
                        low_detasize_result = ((dest & (0xFFFFFFFF >> (32 -
data_size))) << src) & (0xFFFFFFFF >> (32 - data_size));
                        set_PF(low_detasize_result);
                        set_ZF(low_detasize_result, data_size);
                        set_SF(low_detasize_result, data_size);
                        cpu.eflags.CF = dest >> (data_size - src) & 0x00000001;
                // }
                // return (high_32_datasize_dest + low_detasize_result);
                return (high_32_datasize_dest + low_detasize_result) & (0xFFFFFFFF
>> (32 - data_size));
                // 实际只返回截断后低date_size位? ? ?
        }
#endif
}
```

逻辑右移：

```
// dest逻辑右移src位
// CF ← 最后一次移出dest的位（也就是低位）
uint32_t alu_shr(uint32_t src, uint32_t dest, size_t data_size)
{
#ifdef NEMU_REF_ALU
        return __ref_alu_shr(src, dest, data_size);
#else
        // specially for OF
        // For right shifts, the OF flag is set to the high order of the original
operand.
        if (src == 1)
        {
                // get the OF
                cpu.eflags.OF = (dest >> (data_size - 1)) & 0x00000001;
        }

        // to save the high 32 - datasize bits
        uint32_t high_32_datasize_dest = dest & (0xFFFFFFFF << data_size);
        if (data_size == 32)
                high_32_datasize_dest = 0;
        // only save the low datasize bits, to be calculated
        uint32_t low_detasize_result = 0;

        // optimization for src > data_size
        if (src > data_size)
        {
                // low_detasize_result = 0;
                cpu.eflags.CF = 0;
                cpu.eflags.PF = 1;
                cpu.eflags.ZF = 1;
                cpu.eflags.SF = 0;
                // return (high_32_datasize_dest + low_detasize_result);
                return (high_32_datasize_dest + low_detasize_result) & (0xFFFFFFFF
>> (32 - data_size));
                // 实际只返回截断后低date_size位? ? ?
```

```
        }
        else
        {
                low_detasize_result = ((dest & (0xFFFFFFFF >> (32 - data_size))) >>
src) & (0xFFFFFFFF >> (32 - data_size));
                // change eflags
                cpu.eflags.CF = (dest >> (src - 1)) & 0x00000001;
                set_PF(low_detasize_result);
                set_ZF(low_detasize_result, data_size);
                set_SF(low_detasize_result, data_size);
                // return (high_32_datasize_dest + low_detasize_result);
                return (high_32_datasize_dest + low_detasize_result) & (0xFFFFFFFF
>> (32 - data_size));
                // 实际只返回截断后低date_size位? ? ?
        }
#endif
}
```

算术右移：

```
// dest算术右移src位
// CF ← 最后一次移出dest的位（也就是低位）
uint32_t alu_sar(uint32_t src, uint32_t dest, size_t data_size)
{
#ifdef NEMU_REF_ALU
        return __ref_alu_sar(src, dest, data_size);
#else
        // specially for OF
        // For sar, the OF flag is cleared.
        if (src == 1)
        {
                // get the OF
                cpu.eflags.OF = 0;
        }

        // to save the high 32 - datasize bits
        uint32_t high_32_datasize_dest = dest & (0xFFFFFFFF << data_size);
        if (data_size == 32)
                high_32_datasize_dest = 0;
        // only save the low datasize bits, to be calculated
        uint32_t low_detasize_result = 0;

        // optimization for src > data_size
        if (src > data_size)
        {
                // set low detasize bits according to sign
                if ((dest >> (data_size - 1)) & 0x00000001)
                {
                        low_detasize_result = 0xFFFFFFFF >> (32 - data_size);
                        cpu.eflags.CF = 1;
                        cpu.eflags.PF = 1;
                        cpu.eflags.ZF = 0;
                        cpu.eflags.SF = 1;
```

```
                }
                else
                {
                        low_detasize_result = 0;
                        cpu.eflags.CF = 0;
                        cpu.eflags.PF = 1;
                        cpu.eflags.ZF = 1;
                        cpu.eflags.SF = 0;
                }

                // return (high_32_datasize_dest + low_detasize_result);
                return (high_32_datasize_dest + low_detasize_result) & (0xFFFFFFFF
>> (32 - data_size));
                // 实际只返回截断后低date_size位? ? ?
        }
        else
        {
                // set low detasize bits according to sign
                if ((dest >> (data_size - 1)) & 0x00000001)
                {
                        // 补位为1，先截断，再右移，再把data_size - src + 1 到
data_size位补1，最后再截断
                        low_detasize_result = (((dest & (0xFFFFFFFF >> (32 -
data_size))) >> src) | ((0xFFFFFFFF >> (32 - src)) << (data_size - src))) &
(0xFFFFFFFF >> (32 - data_size));
                        set_PF(low_detasize_result);
                        set_ZF(low_detasize_result, data_size);
                        set_SF(low_detasize_result, data_size);
                        cpu.eflags.CF = dest >> (src - 1) & 0x00000001;
                }
                else
                {
                        // 补位为0
                        low_detasize_result = ((dest & (0xFFFFFFFF >> (32 -
data_size))) >> src) & (0xFFFFFFFF >> (32 - data_size));
                        set_PF(low_detasize_result);
                        set_ZF(low_detasize_result, data_size);
                        set_SF(low_detasize_result, data_size);
                        cpu.eflags.CF = dest >> (src - 1) & 0x00000001;
                }
                // return (high_32_datasize_dest + low_detasize_result);
                return (high_32_datasize_dest + low_detasize_result) & (0xFFFFFFFF
>> (32 - data_size));
                // 实际只返回截断后低date_size位? ? ?
        }
#endif
}
```

在实际操作中遇到了一些异常：

首先是按照实验手册中内容：

> 特别说明：针对上面四个移位操作，约定只影响 `dest`操作数的低 `data_size`位，而不影响其高 `32 - data_size`位。标志位的设置根据结果的低 `data_size`位来设置。

在移位操作时应该保证不对高位进行操作，但在实际操作中，必须对高位进行清零操作后再能通过测试案例，而不是保证位移过程中不影响高 `32 - data_size`位。

然后是移位操作的异常现象：

将一个32位数左移32位后的结果并不是全0，而是全1！根据查阅资料，对一个数位移超过其位数的行为的结果是未定义的，因此在代码中增加了一些对这种情况的特别处理。

运行结果

1. 实现 `nemu/src/cpu/alu.c`中的各个整数运算函数；
2. 使用

> make clean
>
> make

命令编译项目；

3. 使用

> make test_pa-1

命令执行NEMU并通过各个整数运算测试用例。

结果如下：

```
./nemu/nemu --test-alu add
NEMU execute built-in tests
alu_test_add()  pass
./nemu/nemu --test-alu adc
NEMU execute built-in tests
alu_test_adc()  pass
./nemu/nemu --test-alu sub
NEMU execute built-in tests
alu_test_sub()  pass
./nemu/nemu --test-alu sbb
NEMU execute built-in tests
alu_test_sbb()  pass
./nemu/nemu --test-alu and
NEMU execute built-in tests
```

```
alu_test_and()  pass
./nemu/nemu --test-alu or
NEMU execute built-in tests
alu_test_or()   pass
./nemu/nemu --test-alu xor
NEMU execute built-in tests
alu_test_xor()  pass
./nemu/nemu --test-alu shl
NEMU execute built-in tests
alu_test_shl()  pass
./nemu/nemu --test-alu shr
NEMU execute built-in tests
alu_test_shr()  pass
./nemu/nemu --test-alu sal
NEMU execute built-in tests
alu_test_sal()  pass
./nemu/nemu --test-alu sar
NEMU execute built-in tests
alu_test_sar()  pass
./nemu/nemu --test-alu mul
NEMU execute built-in tests
alu_test_mul()  pass
./nemu/nemu --test-alu div
NEMU execute built-in tests
alu_test_div()  pass
./nemu/nemu --test-alu imul
NEMU execute built-in tests
alu_test_imul() pass
./nemu/nemu --test-alu idiv
NEMU execute built-in tests
alu_test_idiv() pass
```

PA1-3 浮点数的表示和运算

实验代码

整个代码框架已经设计得较为完整，因此主要展示需要完成的部分：

浮点数的规范化运算

在这一部分中需要补充完成的主要是一些右移、返回无穷、返回0、左移、去掉GRS位的操作，其中需要注意的是：

1. 右移时需要对最后一位的sticky bit做特殊处理；但代码在讨论“左移遇到exp=0，需要额外右移”的操作时，也有注释提到需要注意sticky bit的特殊处理，认为这个其实应该是不必要的，因为sticky bit位必然是0；

2. 返回无穷、返回0时要注意根据符号位来进行返回，返回值可以直接用宏定义，如N_INF_F、P_INF_F之类。

浮点数的加法

这一部分需要补充完成的是对阶时的右移位数的确定，而这实际上计算的是真实阶数的差，因此对于非规格化数部分，需要对exp做+1补偿：

```
/* TODO: shift = ? */
shift = (fb.exponent==0 ? fb.exponent+1 : fb.exponent) - (fa.exponent==0 ?
fa.exponent+1 : fa.exponent);
```

浮点数的减法

这一部分没有要编写的部分，而且可以基于浮点数的加分部分完成，只要需要额外注意一些corner case。

浮点数的乘法

这一部分需要完成的是对计算结果阶数的确定，对这段代码做了详细地论证：

```
// exp=0 and exp=1 means the same thing: 2^-126
// the only diferece is that the hidden bit is 0 or 1
// as we have already set the hidden bit according to the exp, we can just set the
exp to 1
// this has a benefit that we can avoid the special case of exp=0, every exp
substracting 127 is the real exp
if (fa.exponent == 0)
        fa.exponent++;
if (fb.exponent == 0)
        fb.exponent++;
sig_res = sig_a * sig_b; // 24b * 24b
uint32_t exp_res = 0;
/* TODO: exp_res = ? leave space for GRS bits. */

// realexp = (exp_a-127) + (exp_b-127) is the real exp
// and considering .23 * .23 = .46,
// if we still consider that the point is between the 23nd and 24th bit, the exp
should minus 23
// so realexp = (exp_a-127) + (exp_b-127) - 23
// as for exp_res, accoring to the definition of "internal_normalize",
// the real exp is exp_res-127 when exp_res != 0, and the real exp is -126 when
exp_res = 0
// so exp_res = realexp + 127 = (exp_a-127) + (exp_b-127) - 23 + 127 = exp_a +
exp_b - 127 - 23
// should be careful for the corner case
// when exp_a + exp_b - 127 - 23 = 0, the real exp is -127, but the exp_res should
```

```
be 0, indicates real exp is -126
// when exp_a + exp_b - 127 - 23 ≠ 0, the real exp is (exp_a-127) + (exp_b-127) -
23, the exp_res should be exp_a + exp_b - 127 - 23, indicates real exp is exp_a +
exp_b - 127 - 23 -127, the result is the same
// finally need to add 3 for the GRS bits
exp_res = fa.exponent + fb.exponent - 127 - 23 + 3;
// uint32_t sticky = 0;
//  if (exp_res == 0) // indicates real exp is -126, while the real exp is actually
-127
//  {
//      // sigres shift right once
//      sticky = sig_res & 0x1;
//      sig_res = sig_res >> 1;
//      // sticky bits
//      sig_res |= sticky;
// }
```

这一部分的代码实验者有些不理解，考虑到规格化数和非规格化数在exp和阶数真实值的映射法则上有所不同，在最后结果exp_res = 0时，代表的真实的阶应该是-127，但是按照映射关系将exp_res送入internal_normalize()做处理时，是按照真实阶是-126（也就是非规格化）来处理的，因此应该要额外进行一次右移操作。

但从最后结果来看，加上对0的讨论不能通过全部案例，只要注释掉这部分才有正确结果。这一部分让实验者感到很困惑。

运行结果

1. 实现 `nemu/src/cpu/fpu.c`中的各个浮点数运算函数；
2. 使用

> make clean
>
> make

命令编译项目；

3. 使用

> make test_pa-1

4. 使用 `make test_pa-1`命令执行NEMU并通过各个浮点数运算测试用例。

```
./nemu/nemu --test-fpu add
NEMU execute built-in tests
fpu_test_add()  pass
./nemu/nemu --test-fpu sub
```

```
NEMU execute built-in tests
fpu_test_sub()  pass
./nemu/nemu --test-fpu mul
NEMU execute built-in tests
fpu_test_mul()  pass
./nemu/nemu --test-fpu div
NEMU execute built-in tests
fpu_test_div()  pass
```

实验思考题

1. **为浮点数加法和乘法各找两个例子：1）对应输入是规格化或非规格化数，而输出产生了阶码上溢结果为正（负）无穷的情况；2）对应输入是规格化或非规格化数，而输出产生了阶码下溢结果为正（负）零的情况。是否都能找到？若找不到，说出理由。**

   对于浮点数加法：

   1）对应输入是规格化数时，两数为最大规格化正数即 `0b0|11111110|11111111111111111111111`，对应数值约为2*2^127，在实际运算中尾数相加产生进位，需要右规，则此时阶码变为 `0xFF`，同时将尾数清0，发生上溢，得到正无穷；同理取最小规格化数 `0b1|11111110|11111111111111111111111`时，相加发生上溢，得到负无穷；对应输入是非规格化数时，则不会产生上溢，因为数值已经很小。

   2）对应输入是规格化数时，取绝对值最小的规格化数即 `0b0|00000001|00000000000000000000000`，对应数值为2^-126，即使与 `0b1|00000001|00000000000000000000001`发生加法运算，得到的结果为 `0b1|00000000|00000000000000000000001`也不至于下溢；对应输入是非规格化数时，得到的结果一定也能用规格或非规格化数表示。

   总结来说，对浮点数加法，存在上溢，不存在下溢，因为对阶码部分，可以超过254后继续增大发生上溢，却没法在变成0后继续减小发生下溢。

   对于浮点数乘法：

   1）对应输入是规格化数时，同上，取 `0b0|11111110|11111111111111111111111`就能上溢，即使不右规，阶码也严重上溢了；对应于非规格化数时，则不仅不会上溢，甚至一定会下溢——2^-252已经不在float能表示的范围中了。

2）对应输入是规格化数时，考虑到float最小也只能表示到2^(-126-23)，因此规格化数也能发生下溢，如两个数的阶码为-75 + 127 = 52时，相乘之后数量级约在2^(-150)，就会发生下溢，具体机器级表现就是计算exp_res结果为负数，对尾数进行右规，一直规到0也未能使exp_res为0或整数；对应于非规格化数，则如前所述，一定会发生下溢。

总结来说，对浮点数乘法，既有可能发生上溢，也有可能发生下溢。