

计算机系统基础
Programming Assignment

PA 1-2 – 整数的表示和运算

2022年2月25日

南京大学《计算机系统基础》课程组

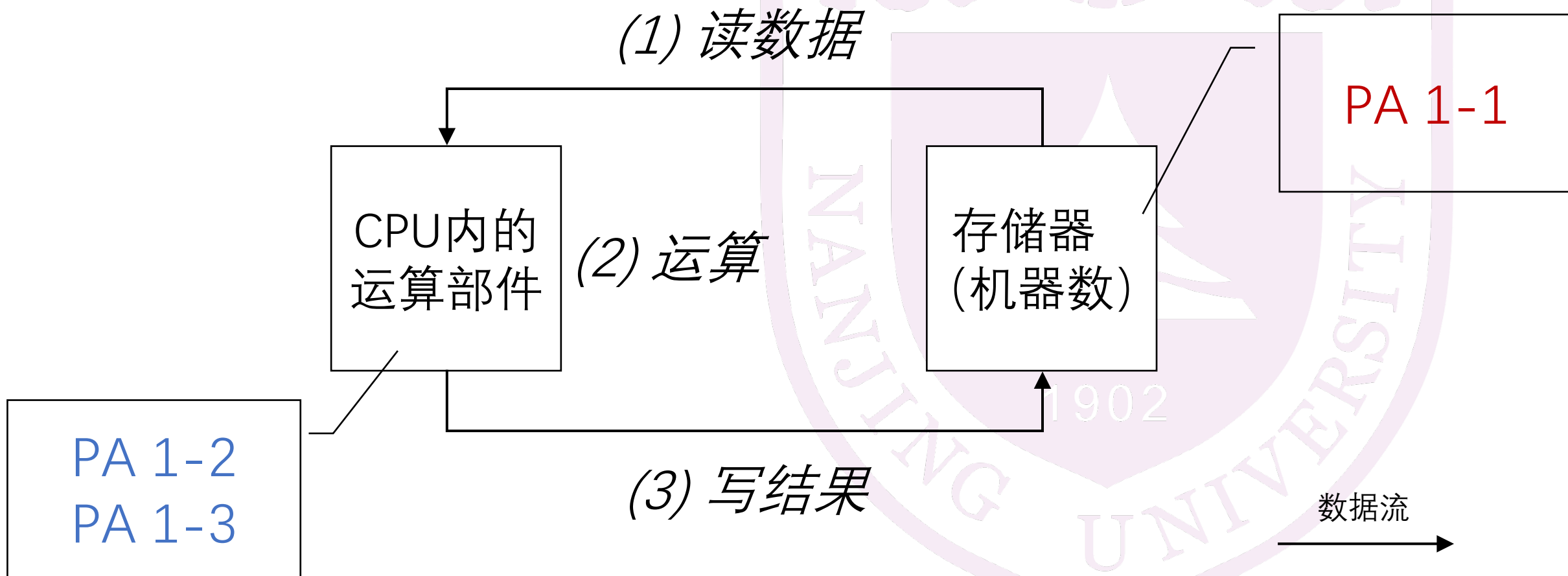
目录

- PA 1-1 数据的类型和存取
- PA 1-2 整数的表示和运算
- PA 1-3 浮点数的表示和运算



机器数的存取

- CPU对数据的处理



通过尝试测试，发现我们需要实现ALU中的相应功能

```
pa_nju$ make clean
pa_nju$ make test_pa-1

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

./nemu/nemu --test-alu add
NEMU execute built-in tests
Please implement me at alu.c
```

需要实现alu的功能

数据的类型及其机器级表示

数据（真值）的类型

编码

机器数

无符号整数

123, 0x8048000

直接编码

带符号整数

-123, 123

原码、补码、反码、移码

1010010010...

二进制位串

整数的表示和运算

- 无符号整数
 - 32位整数: $0x0 \sim 0xFFFFFFFF$ (32个1)
- 二进制编码: 熟练掌握不同进制之间的转换方法

$$\begin{aligned} & 1010\text{B} \\ = & 12\text{O} \\ = & 10\text{D} \\ = & \text{AH} \end{aligned}$$

整数的表示和运算

- 带符号整数
 - 原码表示法：最高位为符号位
 - 补码表示法（普遍采用）：
 - 各位取反末位加一
 - 用加法来实现减法
 - 可以试试 $X + (-X)$ 等于多少，其中 X 为某一32位正整数， $-X$ 为其补码表示，运算结果截取低32位
 - 移码表示法：增加偏置常数，将在浮点数的IEEE 754标准中得到应用

数据的类型及其机器级表示

数据（真值）的类型

编码

机器数

无符号整数

123, 0x8048000

直接编码

带符号整数

-123, 123

原码、补码、反码、移码

1010010010...

二进制位串

小端方式

存储器

寄存器

主存

等

机器数存储到存储器

机器数: 0x 12 34 56 78

高位 31

0001 0010 0011 0100 0101 0110 0111 1000

0 低位

EAX 寄存器

低地址

0x0

0x1

0x2

0x3

0x4

0x5

0x6

0x7

0x8

0x9

高地址

0x78

0x56

0x34

0x12

字节

字节

字节

字节

字节

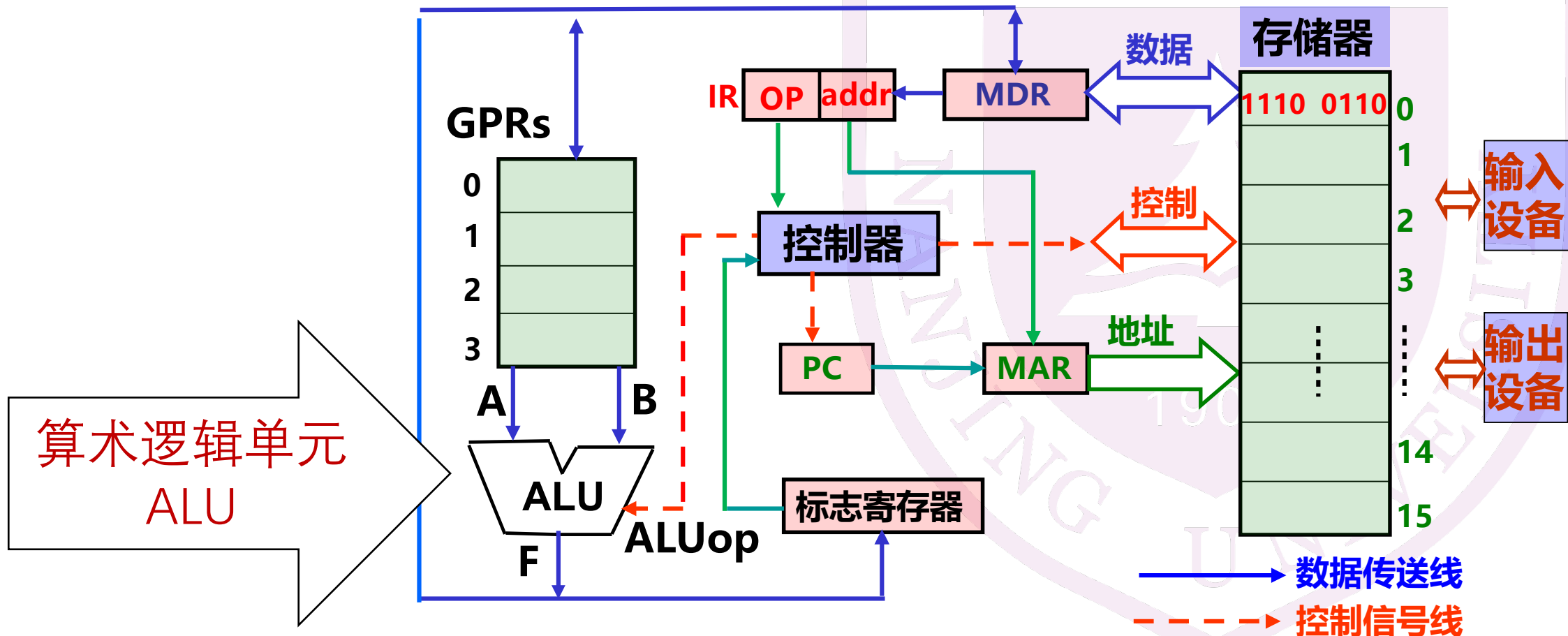
字节

小端: 低有效字节在低地址

主存 (RAM) 按字节编址

整数的运算部件

CPU: 中央处理器; PC: 程序计数器; MAR: 存储器地址寄存器
ALU: 算术逻辑部件; IR: 指令寄存器; MDR: 存储器数据寄存器
GPRs: 通用寄存器组 (由若干通用寄存器组成)



算术逻辑单元

- 我们对ALU的功能进行了抽象和包装
 - 能进行各类算术运算：加减乘除、移位
 - 能进行各种逻辑运算：与或非
- 对应代码：
 - `nemu/include/cpu/alu.h`
 - `nemu/src/cpu/alu.c`



实现对一种运算的模拟

第一步：找到需要实现的函数

执行make test_pa-1遇到错误提示

```
pa_nju$ make clean  
pa_nju$ make test_pa-1
```

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

```
./nemu/nemu --test-alu add  
NEMU execute built-in tests  
Please implement me at alu.c
```

需要实现add

算术逻辑单元的模拟

- 取add为例

函数名称, 对应指令名称 参与运算的两个操作数 操作数长度: 8, 16, 32

```
uint32_t alu_add(uint32_t src, uint32_t dest, size_t data_size)
{
    printf("\e[0;31mPlease implement me at alu.c\e[0m\n");
    assert(0);
    return 0;
}
```

要替换成教程中说明的正确实现:

1. 返回dest + src的结果, data_size不足32时, 高位清0
2. 设置EFLAGS标志位

实现对一种运算的模拟

第二步：掏出i386手册

1. 找到i386手册Sec. 17.2.2.11
2. 有关ADD指令的描述 p.g. 261 of 421
3. 看Flags Affected: OF, SF, ZF, AF, CF, and PF as described in Appendix C
4. 找到Appendix C并仔细体会 (AF不模拟)

实现对一种运算的模拟

第三步：按照手册规定的操作进行实现

[nemu/src/cpu/alu.c](#)
add的参考实现方案

```
uint32_t alu_add(uint32_t src, uint32_t dest, size_t data_size) {  
    uint32_t res = 0;  
    res = dest + src;                                // 获取计算结果  
  
    set_CF_add(res, src, data_size);                 // 设置标志位  
    set_PF(res);  
    // set_AF();                                     // 我们不模拟AF  
    set_ZF(res, data_size);  
    set_SF(res, data_size);  
    set_OF_add(res, src, dest, data_size);  
  
    return res & (0xFFFFFFFF >> (32 - data_size)); // 高位清零并返回  
}
```

ADD运算CF的判断逻辑

- 参考表盘这一模运算系统




```
// CF contains information relevant to unsigned integers
void set_CF_add(uint32_t result, uint32_t src, size_t data_size) {
    result = sign_ext(result & (0xFFFFFFFF >> (32 - data_size)), data_size);
    src = sign_ext(src & (0xFFFFFFFF >> (32 - data_size)), data_size);
    cpu.eflags.CF = result < src; // 对cpu中eflags寄存器的访问
}                                     i386手册 sec 2.3.4.1

void set_ZF(uint32_t result, size_t data_size) {
    result = result & (0xFFFFFFFF >> (32 - data_size));
    cpu.eflags.ZF = (result == 0);
}

// SF and OF contain information relevant to signed integers
void set_SF(uint32_t result, size_t data_size) {
    result = sign_ext(result & (0xFFFFFFFF >> (32 - data_size)), data_size);
    cpu.eflags.SF = sign(result);
}

void set_PF(uint32_t result) { ... } // 简单暴力穷举也行
```

nemu/src/cpu/alu.c
add的参考实现方案

```
// CF contains information relevant to unsigned integers
void set_CF_add(uint32_t result, uint32_t src, size_t data_size) {
    result = sign_ext(result & (0xFFFFFFFF >> (32 - data_size)), data_size);
    src = sign_ext(src & (0xFFFFFFFF >> (32 - data_size)), data_size);
    cpu.eflags.CF =
}

void set_ZF(uint32_t result, size_t data_size) {
    result = result & (0xFFFFFFFF >> (32 - data_size));
    cpu.eflags.ZF =
}

// SF and OF contain information relevant to signed integers
void set_SF(uint32_t result, size_t data_size) {
    result = sign_ext(result & (0xFFFFFFFF >> (32 - data_size)), data_size);
    cpu.eflags.SF =
}

void set_PF(uint32_t result, size_t data_size) {
    result = result & (0xFFFFFFFF >> (32 - data_size));
    cpu.eflags.PF =
}
```

nemu/include/cpu/alu.h

```
// sign extend
#define sign(x) ((uint32_t)(x) >> 31)
// #define sign_ext(x) ((int32_t)((int8_t)(x)))

inline uint32_t sign_ext(uint32_t x, size_t data_size)
{
    assert(data_size == 16 || data_size == 8 || data_size == 32);
    switch (data_size)
    {
        case 8:
            return (int32_t)((int8_t)(x & 0xff));
        case 16:
            return (int32_t)((int16_t)(x & 0xffff));
        default:
            return (int32_t)x;
    }
}
```

u.c
方案

```
// CF contains information about carry
void set_CF_add(uint32_t result, uint32_t src) {
    result = sign_ext(result, src);
    cpu.eflags.CF = 1;
}

void set_ZF(uint32_t result) {
    result = sign_ext(result, 0);
    cpu.eflags.ZF = 1;
}

// SF and OF contain information about sign and overflow
void set_SF(uint32_t result) {
    result = sign_ext(result, 0);
    cpu.eflags.SF = 1;
}

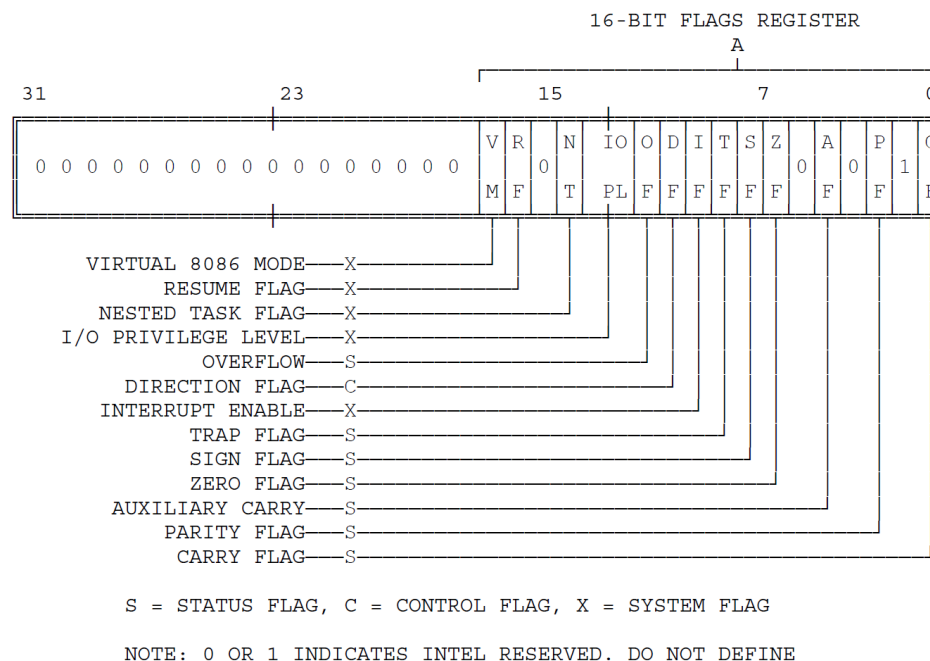
void set_PF(uint32_t result) {
    result = sign_ext(result, 0);
    cpu.eflags.PF = 1;
}
```

```
// define the structure of registers
typedef struct {
    // general purpose registers
    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;
} CPU_STATE;
```

nemu/include/cpu/reg.h

Figure 2-8. EFLAGS Register



nemu/src/cpu/cpu.c

```
CPU_STATE cpu;
FPU fpu;
int nemu_state;
uint8_t data_size = 32;
bool verbose = false;
bool is_nemu_hlt = false;
bool has_prefix = false;

#define sign(x) ((uint32_t)(x) >> 31)

void do_devices();
void init_cpu(const uint32_t init_eip)
{
    cpu.eflags.val = 0x0;
    fpu.status.val = 0x0;
    int i = 0;
```

u.c
方案

```
void set_OF_add(uint32_t result, uint32_t src, uint32_t dest, size_t data_size) {
    switch(data_size) {
        case 8:
            result = sign_ext(result & 0xFF, 8);
            src = sign_ext(src & 0xFF, 8);
            dest = sign_ext(dest & 0xFF, 8);
            break;
        case 16:
            result = sign_ext(result & 0xFFFF, 16);
            src = sign_ext(src & 0xFFFF, 16);
            dest = sign_ext(dest & 0xFFFF, 16);
            break;
        default: break; // do nothing
    }
    if(sign(src) == sign(dest)) {
        if(sign(src) != sign(result))
            cpu.eflags.OF = 1;
        else
            cpu.eflags.OF = 0;
    } else {
        cpu.eflags.OF = 0;
    }
}
```

1902
nemu/src/cpu/alu.c
add的参考实现方案

对于ADC和SUB的特殊说明

- ADC需要结合CF的取值判断
 - 考虑 $CF == 1$ 时, $src == res$ 的情况是否产生进位 ?
- SUB为什么不能对减数取补码后, 简单复用add的CF判断标准 ?

实现对一种运算的模拟

重复第一步：找到需要实现的函数
执行make test_pa-1遇到错误提示

```
./nemu/nemu --test-alu add  
NEMU execute built-in tests  
alu_test_add() pass  
./nemu/nemu --test-alu adc  
NEMU execute built-in tests  
Please implement me at alu.c
```

需要实现adc

实现对ALU的模拟

完成对ALU的模拟

```
alu_test_add()    pass
alu_test_adc()    pass
alu_test_sub()    pass
alu_test_sbb()    pass
alu_test_and()    pass
alu_test_or()     pass
alu_test_xor()    pass
alu_test_shl()    pass
alu_test_shr()    pass
alu_test_sal()    pass
alu_test_sar()    pass
alu_test_mul()    pass
alu_test_div()    pass
alu_test_imul()   pass
alu_test_idiv()   pass
```

测试用例代码:

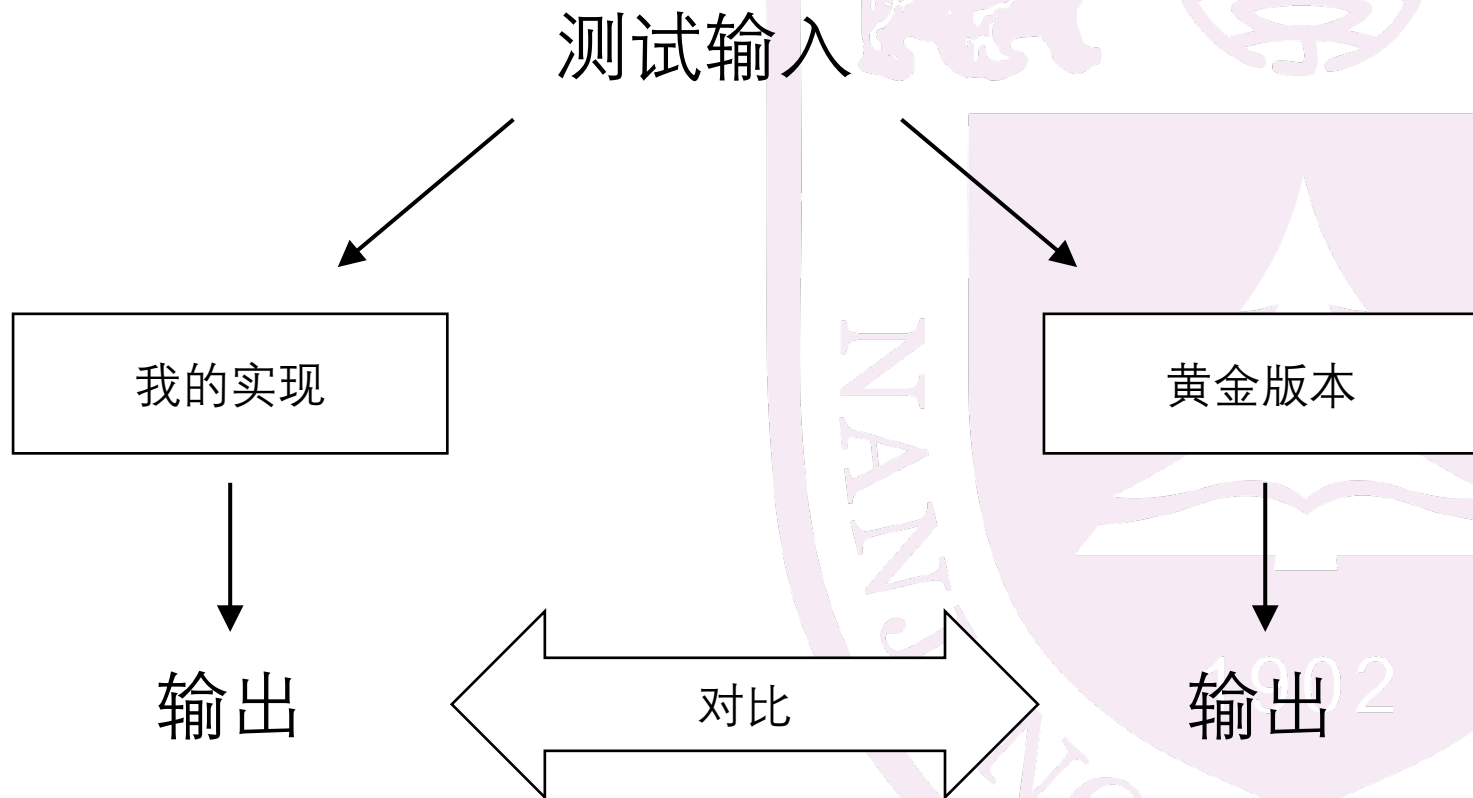
[nemu/src/cpu/test/alu_test.c](#)

注意: 移位操作不测试OF位
所有操作都不测试AF位
imul操作所有标志位都不测试

注意: 禁止采用测试用例里面使用内联汇编进行实现的方法, 但是可以学习这种交叉验证的方法

说明: 其中mod和imod运算是我们单独抽象出来的, 需参照div和idiv的说明进行实现并测试

框架代码对于ALU运算的测试方法



框架代码对于ALU运算的测试方法

nemu/src/cpu/test/alu_test.c

```
#define internal_alu_test_CPSZO(alu_func, dataSize, asm_instr) \  
uint32_t res, res_asm, res_eflags;\br/>TEST_EFLAGS test_eflags;\br/>res = alu_func(b, a, dataSize);\br/>asm (    asm_instr \  
assert_res_CPSZO(dataSize)
```

```
#define assert_res_CPSZO(dataSize) \  
    "pushf;" \  
    "popl %%edx;" \  
    : "=a" (res_asm), "=d" (res_eflags) \  
    : "a" (a), "c" (b)); \  
test_eflags.val = res_eflags; \  
res_asm = res_asm & (0xFFFFFFFF >> (32 - dataSize)); \  
fflush(stdout); \  
assert(res == res_asm); \  
assert(cpu.eflags.CF == test_eflags.CF); \  
assert(cpu.eflags.PF == test_eflags.PF); \  
assert(cpu.eflags.SF == test_eflags.SF); \  
assert(cpu.eflags.ZF == test_eflags.ZF); \  
assert(cpu.eflags.OF == test_eflags.OF); \  

```

```
void alu_test_add() {  
    uint32_t a, b;  
    int input[] = {0x10000000, -3, -2, -1, 0, 1, 2};  
    int n = sizeof(input) / sizeof(int);  
    int i, j;  
    for(i = 0 ; i < n ; i++) {  
        for(j = 0 ; j < n ; j++) {  
            a = input[i];  
            b = input[j];  
            {internal_alu_test_CPSZO(alu_add, 32, "addl %%ecx, %%eax;")}  
            {internal_alu_test_CPSZO(alu_add, 16, "addw %%cx, %%ax;")}  
            {internal_alu_test_CPSZO(alu_add, 8 , "addb %%cl, %%al;")}  
        }  
    }  
  
    srand(time(0));  
    for(i = 0 ; i < 1000000 ; i++) {  
        a = rand();  
        b = rand();  
        {internal_alu_test_CPSZO(alu_add, 32, "addl %%ecx, %%eax;")}  
        {internal_alu_test_CPSZO(alu_add, 16, "addw %%cx, %%ax;")}  
        {internal_alu_test_CPSZO(alu_add, 8 , "addb %%cl, %%al;")}  
    }  
    printf("alu_test_add()  \e[0;32mpass\e[0m\n");  
    if( get_ref() ) printf("\e[0;31mYou have used reference implementatio  
}
```

后续阶段对ALU模拟的依赖

- 在PA 2-1阶段的算术运算指令中，调用本阶段所实现的alu函数

```
#include "cpu/instr.h"

static void instr_execute_2op() {
    operand_read(&opr_src);
    operand_read(&opr_dest);
    opr_dest.val = alu_add(sign_ext(opr_src.val, opr_src.data_size),
                          sign_ext(opr_dest.val, opr_dest.data_size),
                          data_size);
    operand_write(&opr_dest);
}

make_instr_impl_2op(add, r, rm, b)
```

重要说明

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。

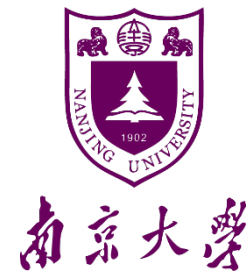
——感谢16级何峰彬、张明超同学的建议

该特别说明不再生效，移位操作也需要将高位清零（此时就不存在是否影响高 $32 - data_size$ 位的问题了）

提醒注意SLR和SAR的区别

实现对ALU的模拟

- 实现ALU的目的
 - 复习课本第二章内容
 - 在alu.c中实现的这些函数，到了PA 2实现对应指令的时候，就可以直接调用了
- PA 1不设置小的阶段截止，PA 1-3完成后统一提交
 - 建议PA 1-1和PA 1-2在一周内完成



PA 1-2 结束