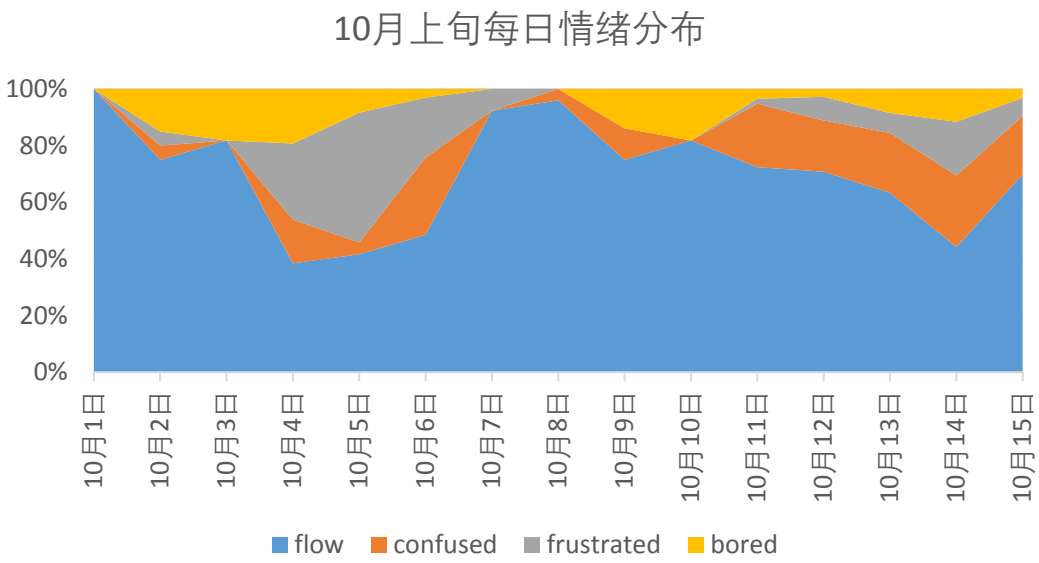
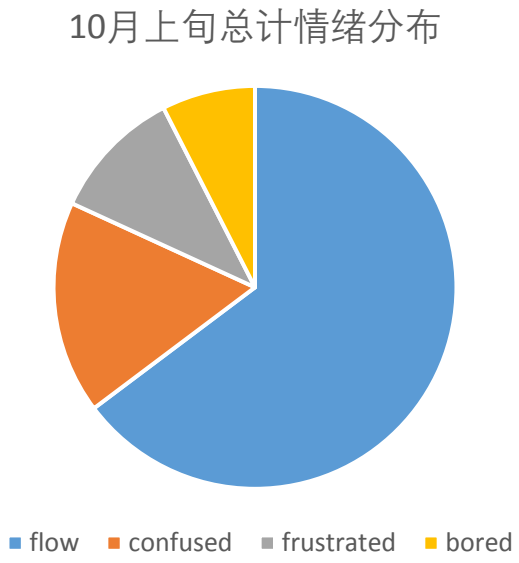
计算机系统基础
Programming Assignment

# PA Macro和Debugging

2019年10月18日

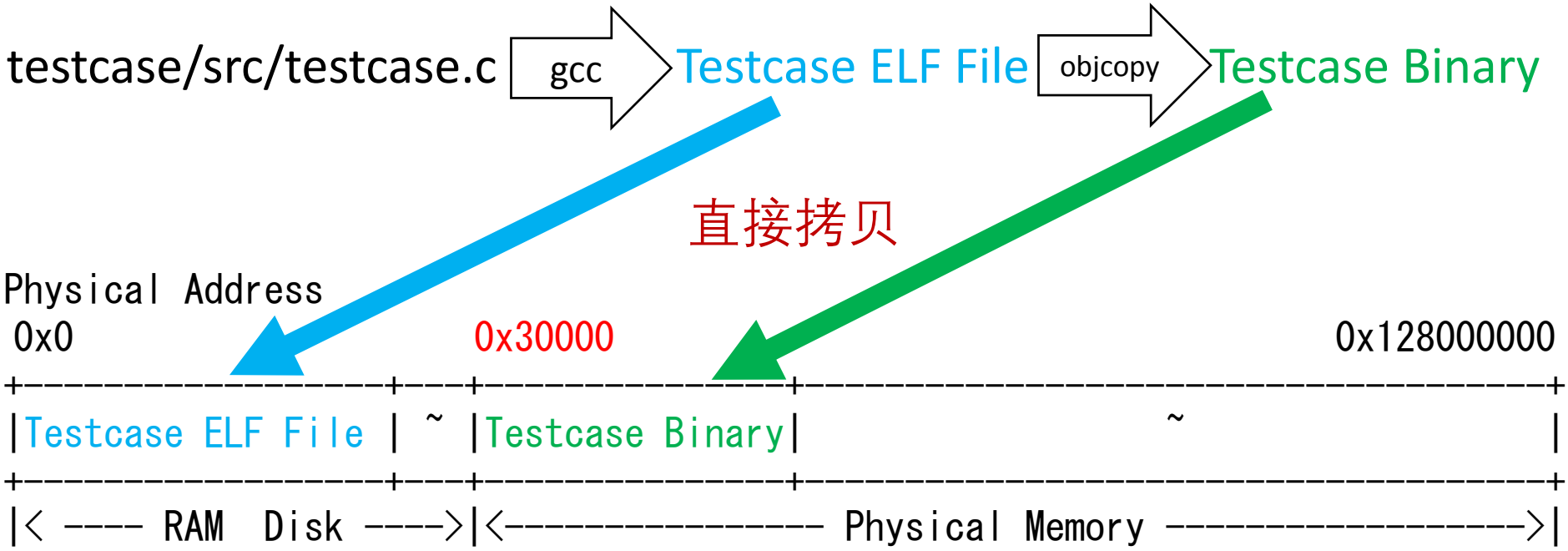

10月上旬每日情绪分布
100%
80%
60%
40%
20%
0%
10月1日
10月2日
10月3日
10月4日
10月5日
10月6日
10月7日
10月8日
10月9日
10月10日
10月11日
10月12日
10月13日
10月14日
10月15日
flow
confused
frustrated
bored


10月上旬总计情绪分布
flow
confused
frustrated
bored

# 前情提要：NEMU的初始化

- 执行make run或者make test之后

第二步，NEMU初始化模拟内存




**带有RAM Disk时的NEMU模拟内存划分方式**

# 前情提要： NEMU模拟指令执行

- 指令循环：一条接一条的执行指令

*nemu/src/cpu/cpu.c*

```
void exec(uint32_t n) {
        ...
        while( n > 0 && nemu_state == NEMU_RUN) {
                ...
                instr_len = exec_inst();
                cpu.eip += instr_len;
                n--;
                ...
        }
        ...
}

int exec_inst() {
        uint8_t opcode = 0;
        // get the opcode，取操作数
        opcode = instr_fetch(cpu.eip, 1);
        // instruction decode and execution，执行这条指令
        int len = opcode_entry[opcode](cpu.eip, opcode);
        return len; // 返回指令长度
}
```

循环执行指令

执行一条指令

```
uint32_t instr_fetch(vaddr_t vaddr, size_t len) {
        assert(len == 1 || len == 2 || len == 4);
        return vaddr_read(vaddr, SREG_CS, len);
}
```

1. 取指令

2. 模拟执行

3. 根据指令长度更新EIP，指向下一条指令

# 内容提要

- 用于精简指令实现的宏
- 以正确的心态和方法去应对bug

# 内容提要

- 用于精简指令实现的宏
- 以正确的心态和方法去应对bug

# 第一遍构建PA时所写的代码

```
make_instr_func(mov_i2rm_b) {
    OPERAND rm, imm;

    imm.data_size = rm.data_size = 8; // 指定操作数长度

    int len = 1;
    len += modrm_rm(eip + 1, &rm); // 操作数寻址
    imm.type = OPR_IMM;
    imm.addr = eip + len;

    operand_read(&imm); // mov操作
    rm.val = imm.val;
    operand_write(&rm);

    return len + 1;
}
```

```
make_instr_func(mov_i2rm_v) {
    OPERAND rm, imm;

    imm.data_size = rm.data_size = data_size; // 指定操作数长度

    int len = 1;
    len += modrm_rm(eip + 1, &rm); // 操作数寻址
    imm.type = OPR_IMM;
    imm.addr = eip + len;

    operand_read(&imm); // mov操作
    rm.val = imm.val;
    operand_write(&rm);

    return len + data_size / 8;
}
```

# 第一遍构建PA时所写的代码

```
make_instr_func(mov_rm2r_b) {
    OPERAND r, rm;

    r.data_size = rm.data_size = 8; // 指定操作数长度

    int len = 1;
    len += modrm_r_rm(eip + 1, &r, &rm); // 操作数寻址

    operand_read(&rm);// mov操作
    r.val = rm.val;
    operand_write(&r);

    return len;
}
```

```
make_instr_func(mov_r2rm_v) {
    OPERAND r, rm;

    rm.data_size = r.data_size = data_size; // 指定操作数长度

    int len = 1;
    len += modrm_r_rm(eip + 1, &r, &rm); // 操作数寻址

    operand_read(&r); // mov操作
    rm.val = r.val;
    operand_write(&rm);

    return len;
}
```

# 每次都是对代码进行局部修改

```
make_instr_func(mov_i2rm_b) {
    OPERAND rm, imm;

    imm.data_size = rm.data_size = 8; // 指定操作数长度

    int len = 1;
    len += modrm_rm(eip + 1, &rm); // 操作数寻址
    imm.type = OPR_IMM;
    imm.addr = eip + len;

    operand_read(&imm); // mov操作
    rm.val = imm.val;
    operand_write(&rm);

    return len + 1;
}
```

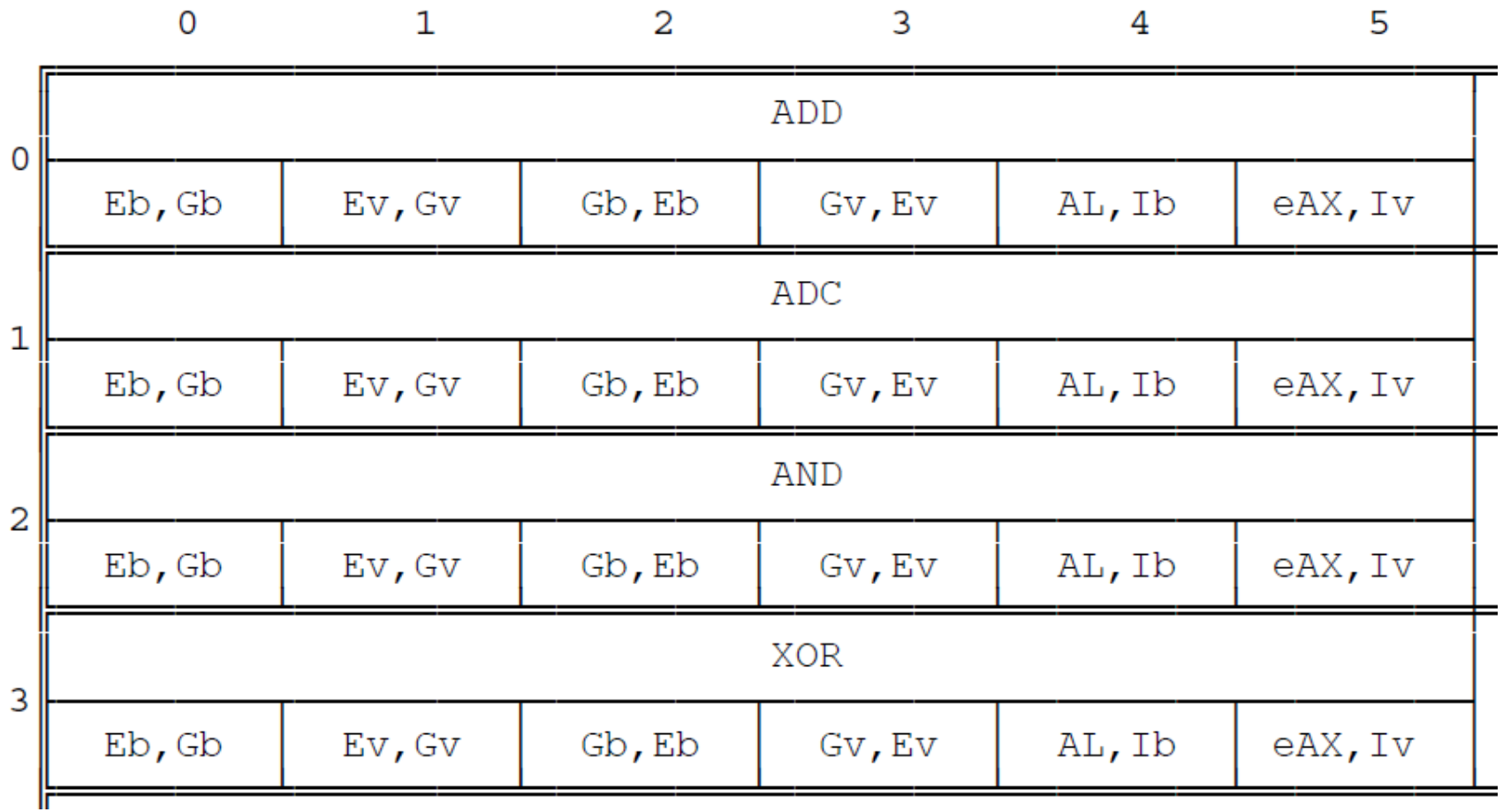
```
make_instr_func(mov_i2rm_v) {
    OPERAND rm, imm;

    imm.data_size = rm.data_size = data_size; // 指定操作数长度

    int len = 1;
    len += modrm_rm(eip + 1, &rm); // 操作数寻址
    imm.type = OPR_IMM;
    imm.addr = eip + len;

    operand_read(&imm); // mov操作
    rm.val = imm.val;
    operand_write(&rm);

    return len + data_size / 8;
}
```

# 复制-黏贴写代码法

- 通过复制-黏贴来写代码
  - 采用前面所示的方法自然能够写出所有指令的实现
  - 但会涉及到大量重复的代码
  - 于是不断使用CP（复制-粘贴）大法来进行编码
- 但是代码克隆是很糟糕的！


猜我是怎么写错的？

# 大量指令操作相同，只是操作数的类型和长度不同



| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | ADD | | | | | |
| | Eb,Gb | Ev,Gv | Gb,Eb | Gv,Ev | AL,Ib | eAX,Iv |
| 1 | ADC | | | | | |
| | Eb,Gb | Ev,Gv | Gb,Eb | Gv,Ev | AL,Ib | eAX,Iv |
| 2 | AND | | | | | |
| | Eb,Gb | Ev,Gv | Gb,Eb | Gv,Ev | AL,Ib | eAX,Iv |
| 3 | XOR | | | | | |
| | Eb,Gb | Ev,Gv | Gb,Eb | Gv,Ev | AL,Ib | eAX,Iv |

# 用于精简指令的宏

nemu/src/cpu/instr/mov.c

```
static void instr_execute_2op()
{
        operand_read(&opr_src);
        opr_dest.val = opr_src.val;
        operand_write(&opr_dest);
}

make_instr_impl_2op(mov, r, rm, b)
make_instr_impl_2op(mov, r, rm, v)
make_instr_impl_2op(mov, rm, r, b)
make_instr_impl_2op(mov, rm, r, v)
make_instr_impl_2op(mov, i, rm, b)
make_instr_impl_2op(mov, i, rm, v)
make_instr_impl_2op(mov, i, r, b)
make_instr_impl_2op(mov, i, r, v)
make_instr_impl_2op(mov, a, o, b)
make_instr_impl_2op(mov, a, o, v)
make_instr_impl_2op(mov, o, a, b)
make_instr_impl_2op(mov, o, a, v)
```

# 怎么从左边变到右边的？

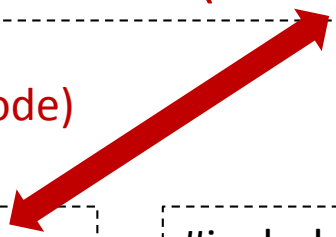
```
make_instr_func(mov_r2rm_v) {
    OPERAND r, rm;
    // 指定操作数长度
    rm.data_size = r.data_size = data_size;
    int len = 1;
    // 操作数寻址
    len += modrm_r_rm(eip + 1, &r, &rm);
    // 执行mov操作
    operand_read(&r);
    rm.val = r.val;
    operand_write(&rm);
    // 返回操作数长度
    return len;
}
```

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

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

make_instr_impl_2op(mov, r, rm, v)
```

*nemu/src/cpu/instr/mov.c*

## *nemu/include/cpu/instr_helper.h*

```
// macro for making an instruction entry
#define make_instr_func(name) int name(uint32_t eip, uint8_t opcode)
```



```
int mov_r2rm_v (uint32_t eip, uint8_t opcode)
```

```
make_instr_func(mov_r2rm_v) {
    OPERAND r, rm;
    // 指定操作数长度
    rm.data_size = r.data_size = data_size;
    int len = 1;
    // 操作数寻址
    len += modrm_r_rm(eip + 1, &r, &rm);
    // 执行mov操作
    operand_read(&r);
    rm.val = r.val;
    operand_write(&rm);
    // 返回操作数长度
    return len;
}
```

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

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

make_instr_impl_2op(mov, r, rm, v)
```

## *nemu/src/cpu/instr/mov.c*

```
// macro for generating the implementation of an instruction with two operands
#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) \
    // 等于 make_instr_impl_2op(mov, r, rm, v)
    make_instr_func(concat7(inst_name, _, src_type, 2, dest_type, _, suffix)) {\
    // 宏展开等于 make_instr_func(mov_ r2rm_v) {
            int len = 1; \
            concat(decode_data_size_, suffix) \
            concat3(decode_operand, _, concat3(src_type, 2, dest_type)) \
            print_asm_2(...); \
            instr_execute_2op(); \
            return len; \
    }
```

*nemu/include/cpu/instr_helper.h*

```
    // 操作数寻址
    len += modrm_r_rm(eip + 1, &r, &rm);
    // 执行mov操作
    operand_read(&r);
    rm.val = r.val;
    operand_write(&rm);
    // 返回操作数长度
    return len;
}
```

```
    operand_write(&opr_dest);
}

make_instr_impl_2op(mov, r, rm, v)
```

*nemu/src/cpu/instr/mov.c*

```
// macro for generating the implementation of an instruction with two operands
#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) \
        // 等于 make_instr_impl_2op(mov, r, rm, v)
        make_instr_func(concat7(inst_name, _, src_type, 2, dest_type, _, suffix)) {\
         // 宏展开等于 make_instr_func(mov_ r2rm_v) {
                int len = 1; \ // 不变
                concat(decode_data_size_, suffix) \
                concat3(decode_operand, _, concat3(src_type, 2, dest_type)) \
                print_asm_2(...); \
                instr_execute_2op(); \
                return len; \
        }
```

*nemu/include/cpu/instr_helper.h*

```
    // 操作数寻址
    len += modrm_r_rm(eip + 1, &r, &rm);
    // 执行mov操作
    operand_read(&r);
    rm.val = r.val;
    operand_write(&rm);
    // 返回操作数长度
    return len;
}
```

```
operand_write(&opr_dest);
}

make_instr_impl_2op(mov, r, rm, v)
```

*nemu/src/cpu/instr/mov.c*

```
// macro for generating the implementation of an instruction with two operands
#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) \
        // 等于 make_instr_impl_2op(mov, r, rm, v)
        make_instr_func(concat7(inst_name, _, src_type, 2, dest_type, _, suffix)) {\
         // 宏展开等于 make_instr_func(mov_ r2rm_v) {
                int len = 1; \ // 不变
                concat(decode_data_size_, suffix) \
// 宏展开等于 decode_data_size_v
// 下方宏定义 #define decode_data_size_v opr_src.data_size = opr_dest.data_size =
data_size;
                concat3(decode_operand, _, concat3(src_type, 2, dest_type)) \
                print_asm_2(...); \
                instr_execute_2op(); \
                return len; \
```

*nemu/include/cpu/instr_helper.h*

```
    // 执行mov操作
    operand_read(&r);
    rm.val = r.val;
    operand_write(&rm);
    // 返回操作数长度
    return len;
}
```

```
make_instr_impl_2op(mov, r, rm, v)
```

*nemu/src/cpu/instr/mov.c*

```
// macro for generating the implementation of an instruction with two operands
#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) \
        // 等于 make_instr_impl_2op(mov, r, rm, v)
        make_instr_func(concat7(inst_name, _, src_type, 2, dest_type, _, suffix)) {\
         // 宏展开等于 make_instr_func(mov_ r2rm_v) {
                int len = 1; \ // 不变
                concat(decode_data_size_, suffix) \
// 宏展开等于 decode_data_size_v
// 下方宏定义 #define decode_data_size_v opr_src.data_size = opr_dest.data_size =
data_size;
                concat3(decode_operand, _, concat3(src_type, 2, dest_type)) \
// 宏展开等于 decode_operand_r2rm
// 下方宏定义 #define decode_operand_r2rm \
//                 len += modrm_r_rm(eip + 1, &opr_src, &opr_dest);
                print_asm_2(...); \
                instr_execute_2op(); \
                return len; \
```

*nemu/include/cpu/instr_helper.h*

```
        }
        rm.val = r.val;
        operand_write(&rm);
        // 返回操作数长度
        return len;
    }
```

```
// macro for generating the implementation of an instruction with two operands
#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) \
        // 等于 make_instr_impl_2op(mov, r, rm, v)
        make_instr_func(concat7(inst_name, _, src_type, 2, dest_type, _, suffix)) {\
         // 宏展开等于 make_instr_func(mov_ r2rm_v) {
                int len = 1; \ // 不变
                concat(decode_data_size_, suffix) \
// 宏展开等于 decode_data_size_v
// 下方宏定义 #define decode_data_size_v opr_src.data_size = opr_dest.data_size =
data_size;
                concat3(decode_operand, _, concat3(src_type, 2, dest_type)) \
// 宏展开等于 decode_operand_r2rm
// 下方宏定义 #define decode_operand_r2rm \
//                  len += modrm_r_rm(eip + 1, &opr_src, &opr_dest);
                print_asm_2(...); \ // 单步执行打印调试信息，不变
                instr_execute_2op(); \
                return len; \
        }
```

*nemu/include/cpu/instr_helper.h*

```
    rm.val = r.val;
    operand_write(&rm);
    // 返回操作数长度
    return len;
}
```

```
// macro for generating the implementation of an instruction with two operands
#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) \
        // 等于 make_instr_impl_2op(mov, r, rm, v)
        make_instr_func(concat7(inst_name, _, src_type, 2, dest_type, _, suffix)) {\
         // 宏展开等于 make_instr_func(mov_ r2rm_v) {
                int len = 1; \ // 不变
                concat(decode_data_size_, suffix) \
// 宏展开等于 decode_data_size_v
// 下方宏定义 #define decode_data_size_v opr_src.data_size = opr_dest.data_size =
data_size;
                concat3(decode_operand, _, concat3(src_type, 2, dest_type)) \
// 宏展开等于 decode_operand_r2rm
// 下方宏定义 #define decode_operand_r2rm \
//                  len += modrm_r_rm(eip + 1, &opr_src, &opr_dest);
                print_asm_2(...); \ // 单步执行打印调试信息，不变
                instr_execute_2op(); \ //调用执行函数
                return len; \
        }
```

*nemu/include/cpu/instr_helper.h*

```
    rm.val = r.val;
    operand_write(&rm);
    // 返回操作数长度
    return len;
}
```

```
// macro for generating the implementation of an instruction with two operands
#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) \
        // 等于 make_instr_impl_2op(mov, r, rm, v)
                make_instr_func(concat7(inst_name, _, src_type, 2, dest_type, _, suffix)) {\
                 // 宏展开等于 make_instr_func(mov_ r2rm_v) {
                                int len = 1; \ // 不变
                                concat(decode_data_size_, suffix) \
// 宏展开等于 decode_data_size_v
// 下方宏定义 #define decode_data_size_v opr_src.data_size = opr_dest.data_size = data_size;
                                concat3(decode_operand, _, concat3(src_type, 2,
dest_type)) \
// 宏展开等于 decode_operand_r2rm
// 下方宏定义 #define decode_operand_r2rm \
//                 len += modrm_r_rm(eip + 1, &opr_src, &opr_dest);
                                print_asm_2(...); \ // 单步执行打印调试信息，不变
                                instr_execute_2op(); \ //调用执行函数
                                return len; \
                }
```

*nemu/include/cpu/instr_helper.h*

```
    int len = 1;
    // 操作数寻址
    len += modrm_r_rm(eip + 1, &r, &rm);
    // 执行mov操作
    operand_read(&r);
    rm.val = r.val;
    operand_write(&rm);
    // 返回操作数长度
    return len;
}
```

Static关键字很关键！

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

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

make_instr_impl_2op(mov, r, rm, v)
```

*nemu/src/cpu/instr/mov.c*

```
// macro for generating the implementation of an instruction with two operands
#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) \
        // 等于 make_instr_impl_2op(mov, r, rm, v)
        make_instr_func(concat7(inst_name, _, src_type, 2, dest_type, _, suffix)) {\
         // 宏展开等于 make_instr_func(mov_ r2rm_v) {
                int len = 1; \ // 不变
                concat(decode_data_size_, suffix) \
// 宏展开等于 decode_data_size_v
// 下方宏定义 #define decode_data_size_v opr_src.data_size = opr_dest.data_size =
data_size;
                concat3(decode_operand, _, concat3(src_type, 2, dest_type)) \
// 宏展开等于 decode_operand_r2rm
// 下方宏定义 #define decode_operand_r2rm \
//                  len += modrm_r_rm(eip + 1, &opr_src, &opr_dest);
                print_asm_2(...); \ // 单步执行打印调试信息，不变
                instr_execute_2op(); \ //调用执行函数
                return len; \  // 返回指令长度
```

```
    }
    rm.val = r.val;
    operand_write(&rm);
    // 返回操作数长度
    return len;
}
```

*nemu/include/cpu/instr_helper.h*

# 怎么从左边变到右边的？

```
make_instr_func(mov_r2rm_v) {
    OPERAND r, rm;
    // 指定操作数长度
    rm.data_size = r.data_size = data_size;
    int len = 1;
    // 操作数寻址
    len += modrm_r_rm(eip + 1, &r, &rm);
    // 执行mov操作
    operand_read(&r);
    rm.val = r.val;
    operand_write(&rm);
    // 返回操作数长度
    return len;
}
```

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

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

make_instr_impl_2op(mov, r, rm, v)
// 将其进行宏展开后，变为。。。
```

*nemu/src/cpu/instr/mov.c*

# 怎么从左边变到右边的？

*nemu/src/cpu/instr/mov.c*

```
make_instr_func(mov_r2rm_v) {
    OPERAND r, rm;
    // 指定操作数长度
    rm.data_size = r.data_size = data_size;
    int len = 1;
    // 操作数寻址
    len += modrm_r_rm(eip + 1, &r, &rm);
    // 执行mov操作
    operand_read(&r);
    rm.val = r.val;
    operand_write(&rm);
    // 返回操作数长度
    return len;
}
```

等价

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

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

make_instr_impl_2op(mov, r, rm, v)
// 将其进行宏展开后，变为。。。
make_instr_func(mov_ r2rm_v) {
        int len = 1;
        opr_src.data_size = opr_dest.data_size = data_size;
        len += modrm_r_rm(eip + 1, &opr_src, &opr_dest);
        print_asm_2(…);
        instr_execute_2op();
        return len;
}
```

# 怎么从左边变到右边的？

*nemu/src/cpu/instr/mov.c*

```
make_instr_func(mov_r2rm_v) {
    OPERAND r, rm;
    // 指定操作数长度
    rm.data_size = r.data_size = data_size;
    int len = 1;
    // 操作数寻址
    len += modrm_r_rm(eip + 1, &r, &rm);
    // 执行mov操作
    operand_read(&r);
    rm.val = r.val;
    operand_write(&rm);
    // 返回操作数长度
    return len;
}
```

等价

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

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

make_instr_impl_2op(mov, r, rm, v)
// 将其进行宏展开后，变为。。。
make_instr_func(mov_ r2rm_v) {
        int len = 1;
        opr_src.data_size = opr_dest.data_size = data_size;
        len += modrm_r_rm(eip + 1, &opr_src, &opr_dest);
        print_asm_2(…);
        instr_execute_2op();
        return len;
}
```

opr_src和opr_dest是定义在operand.c中的两个全局变量

# 怎么从左边变到右边的？

*nemu/src/cpu/instr/mov.c*

```
make_instr_func(mov_r2rm_v) {
    OPERAND r, rm;
    // 指定操作数长度
    rm.data_size = r.data_size = data_size;
    int len = 1;
    // 操作数寻址
    len += modrm_r_rm(eip + 1, &r, &rm);
    // 执行mov操作
    operand_read(&r);
    rm.val = r.val;
    operand_write(&rm);
    // 返回操作数长度
    return len;
}
```

等价

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

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

make_instr_impl_2op(mov, r, rm, v)
// 将其进行宏展开后，变为。。。
make_instr_func(mov_ r2rm_v) {
        int len = 1;
        opr_src.data_size = opr_dest.data_size = data_size;
        len += modrm_r_rm(eip + 1, &opr_src, &opr_dest);
        print_asm_2(…);
        instr_execute_2op();
        return len;
}
```

modrm系列函数看Guide的描述

# 内容提要

- 用于精简指令实现的宏
- 以正确的心态和方法去应对bug

# Bug总是会有的

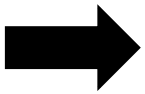
- 在实现的过程中出现了许多稀奇古怪的bug
- 基本的心理发展过程
  - 第一阶段：**不可能是我的错！**一定是框架代码、编译器、操作系统、虚拟机、CPU……里有bug！
  - 第二阶段：嗯……似乎这里有一点小问题，但是**不至于**吧~
  - 第三阶段：**当初这代码怎么能跑起来的！！！！？？？？**
- 调试公理
  - 机器永远是对的
  - 未测试代码永远是错的

# 从Fault到Failure

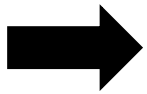
- 一个Bug是怎样最终导致程序出错的？

*从程序中的Fault到Failure的传播过程*

有一个Fault
也就是Bug



运行时导致
了一个Error



最后导致了
程序的Failure

ALU中sub的标志位设置逻辑错误

cmp指令的结果错误，jcc指令跳转到了错误的分支

Bad trap
Segmentation Fault

# 防止出错的方法

- 启用严格的静态检查

```
CFLAGS := -ggdb3 -MMD -MP -Wall -Werror -O2 -I./include -I../include
```

现在能理解为何在./nemu/Makefile的编译选项中，加入-Wall和-Werror选项?

- 在尽早让Error变成Failure
  - 设置一系列的检查点，assert，if判断，……

```
uint32_t hw_mem_read(paddr_t paddr, size_t len)
{
        uint32_t ret = 0;
        memcpy(&ret, hw_mem + paddr, len);
        return ret;
}

void hw_mem_write(paddr_t paddr, size_t len, uint32_t data)
{
        memcpy(hw_mem + paddr, &data, len);
}
```

**应当加入地址越界检查**

nemu/src/memory/memory.c

# 防止出错的方法

- 降低程序对预设条件的依赖（永远假设人只要有机会，就一定会犯错）

```
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;
}
```

- 保持随手测试的习惯
- 不写理解困难的，可能会引起歧义的代码
  - 如后页的代码

# 内容2.4：无符号和带符号整数的转换

- 课堂习题2.4.2

**不要写这种代码！！！！**

```
us = -32768;
i = us;
printf("i = %d, 0x%08x\n"

s = -32768;
ui = s;
printf("ui = %d, 0x%08x\n", ui, ui);

us = -32768;
ui = us;
printf("ui = %d, 0x%08x\n", ui, ui);
```

**Relative order of conversion between data size and signed/unsigned:**
**First change the size and then convert between signed/unsigned**

ui = -32768, 0xffff8000

经验：同时变长度和无符号/带符号的类型转换的赋值操作别做！宁肯分两步走！

# Debugging是需要锻炼的

- 写出不易出错的高质量代码的同时
- Debugging是码农们一生都要面对的问题
  - 基本过程
    - 重现错误（成功一半）：再跑一次、构造新的有针对性的测试用例……
    - 分离和定位root-cause：单步执行、断点……
    - 查看和分析：assert、printf……
    - 运用工具：gdb
    - 总结：不容易犯错的编码方式、构造对测试友好的代码……
  - 踩遍所有的坑，成就伟大程序员

# 框架代码也是有bug的

- 欢迎大家贡献建议和patch，包括但不限于
  - 逻辑bug
  - 代码风格
  - 交互方式
  - 新的测试用例
  - 新的功能模块
  - 讲课、答疑的内容和方式
- 如果建议最终被采纳
  - 你将作为Contributor被记录在PA最后的彩蛋中

# 作为学生，你还有老师和助教

- 提问的艺术
  - https://github.com/ryanhanwu/How-To-Ask-Questions-The-Smart-Way/blob/master/README-zh_CN.md
- 我们不愿意或无法回答的问题
  - 没有经过思考或尝试就来问问题
  - 把我们当作human debugger或者试探参考实现
  - 表述不清的问题
    - 如，没有上下文直接问：“我为什么hit bad trap？”
  - 资料里已经讲清楚但没有看资料
- 我们愿意回答的问题
  - 经过思考和努力后，在能够清晰表述问题，并提供一定分析和尝试结果的基础上所提出来的问题
  - 任何其他有趣的问题

# 作为学生，你还有老师和助教

- 但你也无需太担心
  - 如果问的问题不够友好，我们会回复：“这个问题我无法回答”，并给出简短的理由，并且不存在小本本来记录
  - 如果实在讲不清楚，我们也不会抛弃任何人
- 最关键的原则

You have tried HARD before asking

PA 2-1 Deadline:

2019年10月31日24时，即，2019年11月1日0时

Happy Hacking  O_o