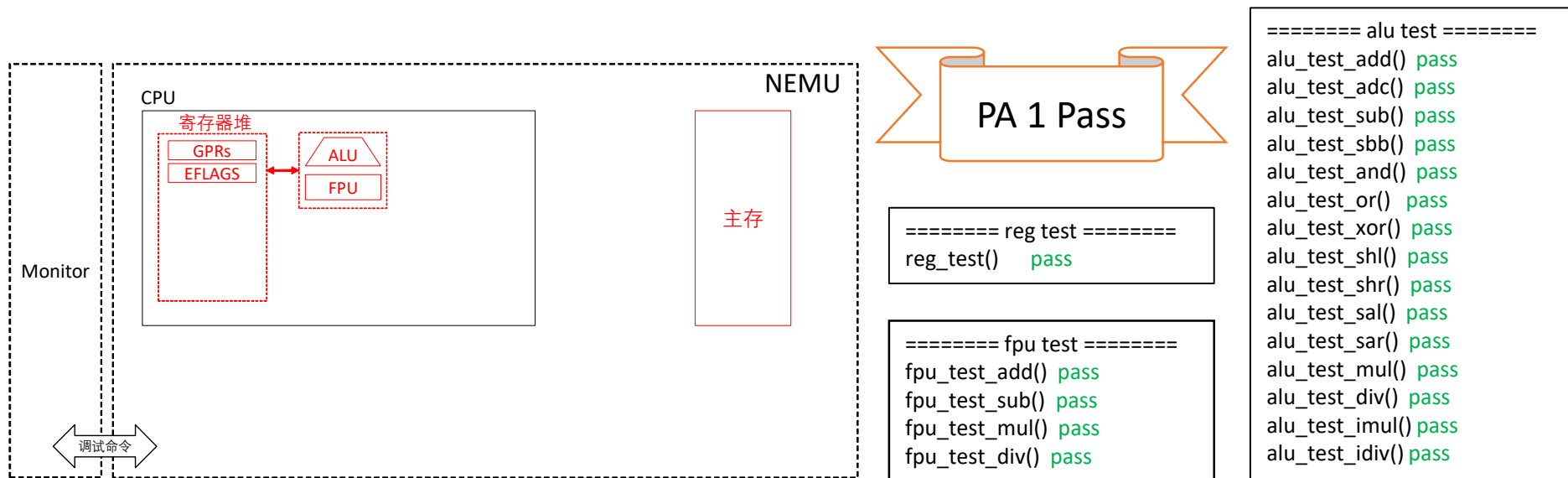


计算机系统基础  
Programming Assignment

# PA 2 程序的执行（第一课） ——PA 2-1 指令解码与执行

2017年9月22日

# 前情提要



迟交将受到该阶段10%分数的惩罚

- 数据的存储
  - 寄存器、主存（内存）
    - 内存vaddr\_read/write()接口教程上少写了sreg参数，现在可以不理睬该参数，PA 3才会涉及
- 数据的表示和运算
  - 整数：带符号、无符号 -> 各种运算 -> 标志位寄存器
  - 带小数的实数：IEEE 754浮点数 -> 各种运算和规格化
- 群公告中发布了几则针对框架代码的补丁

# 前情提要

解决计算问题的步骤  
程序处理的对象  
执行程序的器件  
储存正在处理的数据  
储存马上要用的数据  
存储大量的数据

计算机	餐厅
程序	菜谱
数据	食材
CPU	大厨
CPU内部的寄存器	灶台上的锅
主存	厨房里的冰箱
硬盘	仓库

做菜步骤  
做菜加工的对象  
执行菜谱的人  
放置正在加工的食材  
储存马上用的菜谱和食材  
啥都放这里

# 前情提要



解决计算问题的步骤  
程序处理的对象  
执行程序的器件  
储存正在处理的数据  
储存马上要用的数据  
存储大量的数据

计算机	餐厅
程序	菜谱
数据	食材
CPU	大厨
CPU内部的寄存器	灶台上的锅
主存	厨房里的冰箱
硬盘	仓库

做菜步骤  
做菜加工的对象  
执行菜谱的人  
放置正在加工的食材  
储存马上用的菜谱和食材  
啥都放这里

## PA 2-1 指令解码与执行

- 指令的编码方式
- 指令序列的执行流程
- NEMU模拟指令译码和执行（用于精简指令实现的宏）
- NEMU运行测试用例
- 备用内容：可选任务PA 2-3.1表达式求值

# 指令的编码方式 & 指令序列的执行流程

# 指令的编码方式

- 菜谱中的步骤以自然语言写成

将油倒入锅中  
加热油至七成热

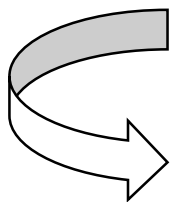
动作序列

1. 倒入 油, 锅
2. 加热 油
3. 比较 油温, 七成热
4. j1 3

- 机器是看不懂文字的 – 只懂0和1
- 机器也无法理解自然语言 – 必须规定格式

抽象

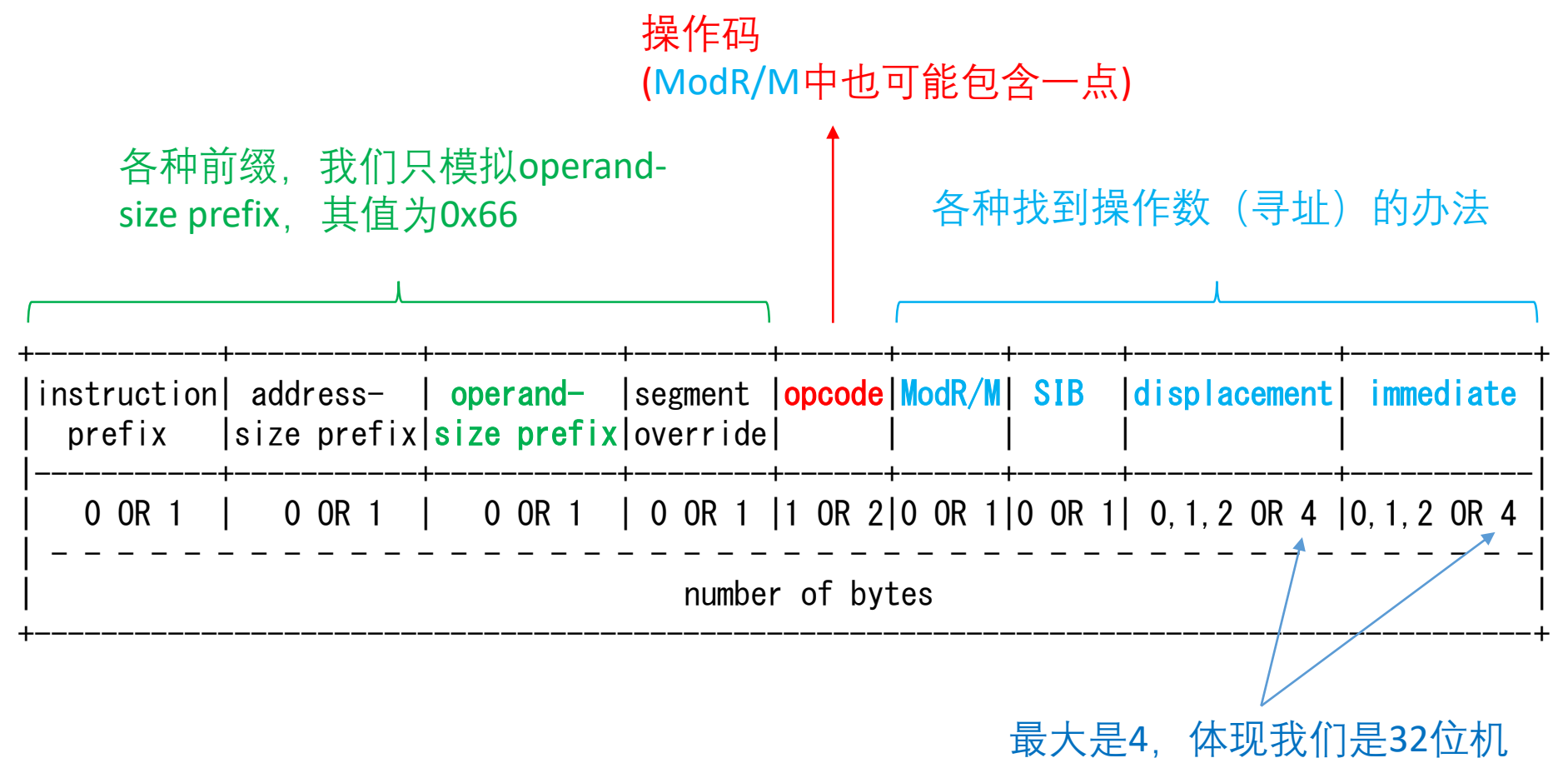
编码



动作  
操作码

对象1, 对象2, 对象3, ...  
操作数1, 操作数2, 操作数3, ...

# 指令的编码方式





# 指令的编码方式

- 理解指令译码的过程  
(假设此时EIP取初始值, 为某程序第一条指令)

EIP



8b 94 83 00 11 00 00 8b 45 f4

---

- 不是0x66, 操作数32位, 0x8b为操作码
- 查i386手册

# 指令的编码方式

- 理解指令译码的过程  
(假设此时EIP取初始值, 为某程序第一条指令)

opcode



8b 94 83 00 11 00 00 8b 45 f4

---

1. 不是0x66, 操作数32位, 0x8b为操作码
2. 查i386手册 – Appendix A – 0x8b对应MOV Gv, Ev
  - Intel格式, 表示把一个Ev类型操作数MOV到Gv类型操作数里
  - objdump和gdb中采用的AT&T格式指令操作数顺序正好相反
  - 若有需要, 到i386手册Chapter 17细查
3. Ev和Gv都说明后面跟ModR/M字节

# 指令的编码方式

- 理解指令译码的过程  
(假设此时EIP取初始值, 为某程序第一条指令)

opcode    modr/m



8b 94 83 00 11 00 00 8b 45 f4

---

- 不是0x66, 操作数32位, 0x8b为操作码
- 查i386手册 – Appendix A – 0x8b对应MOV Gv, Ev
- Ev和Gv都说明后面跟ModR/M字节

0x94 =	10	010	100
	MOD	REG/OPCODE	R/M

# 指令的编码方式

- 理解指令译码的过程  
(假设此时EIP取初始值, 为某程序第一条指令)

opcode    modr/m



8b 94 83 00 11 00 00 8b 45 f4

- 
- 不是0x66, 操作数32位, 0x8b为操作码
  - 查i386手册 – Appendix A – 0x8b对应MOV Gv, Ev

3. Ev和Gv都说明后面跟ModR/M字节

根据Gv和没有0x66前缀, 010表示edx

0x94 =

10

010

100

MOD

REG/OPCODE

R/M

# 指令的编码方式

- 理解指令译码的过程  
(假设此时EIP取初始值, 为某程序第一条指令)

opcode    modr/m



8b 94 83 00 11 00 00 8b 45 f4

- 
- 不是0x66, 操作数32位, 0x8b为操作码
  - 查i386手册 – Appendix A – 0x8b对应MOV Gv, Ev

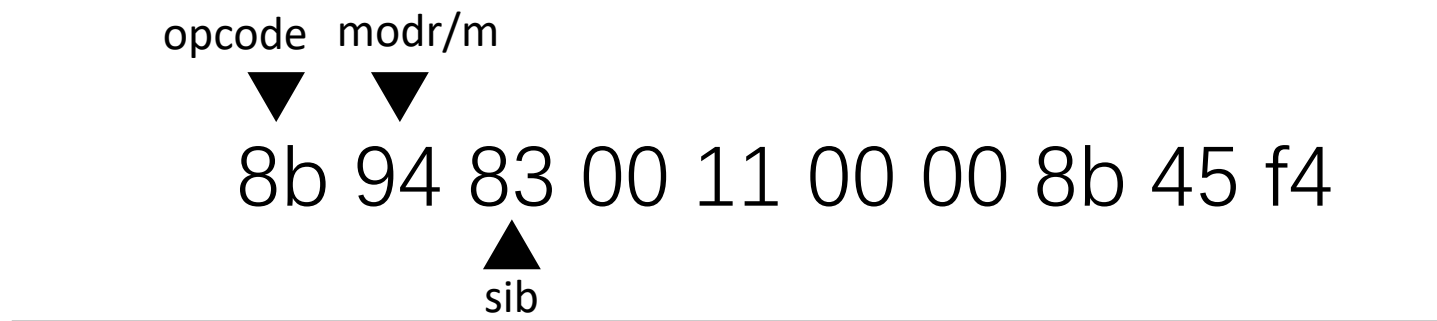
3. Ev和Gv都说明后面跟ModR/M字节

查i386手册表17-3, 发现是内存地址disp32[--][--], 还有SIB字节

0x94 =	10	010	100
	MOD	REG/OPCODE	R/M

# 指令的编码方式

- 理解指令译码的过程  
(假设此时EIP取初始值, 为某程序第一条指令)



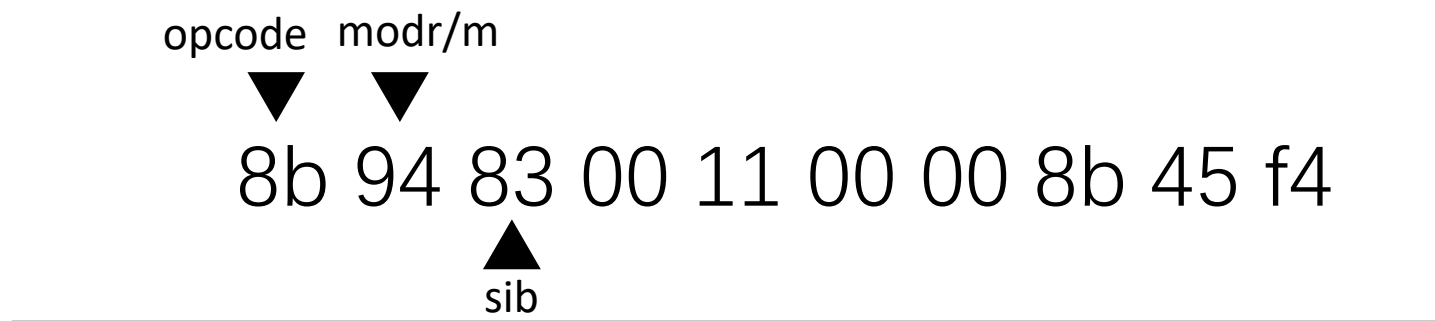
- 不是0x66, 操作数32位, 0x8b为操作码
- 查i386手册 – Appendix A – 0x8b对应MOV Gv, Ev
- Ev和Gv都说明后面跟ModR/M字节
- 根据Mod + R/M域决定有SIB字节 (内存地址disp32[--][--])

0x83 =            10            000            011

                     SS            INDEX            BASE

# 指令的编码方式

- 理解指令译码的过程  
(假设此时EIP取初始值, 为某程序第一条指令)



- 不是0x66, 操作数32位, 0x8b为操作码
- 查i386手册 – Appendix A – 0x8b对应MOV Gv, Ev
- Ev和Gv都说明后面跟ModR/M字节
- 根据Mod + R/M域决定有SIB字节 (内存地址disp32[--][--])

内存地址 =  $\text{disp32} + \text{ebx} + \text{eax} * 4$

0x83 =        10                000                011

查i386手册表17-4

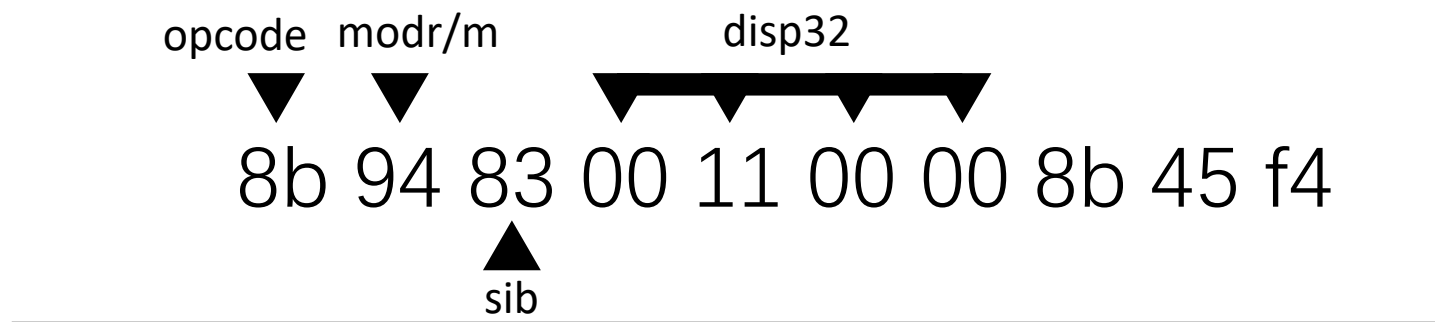
SS  
\*4

INDEX  
eax

BASE  
ebx

# 指令的编码方式

- 理解指令译码的过程  
(假设此时EIP取初始值, 为某程序第一条指令)

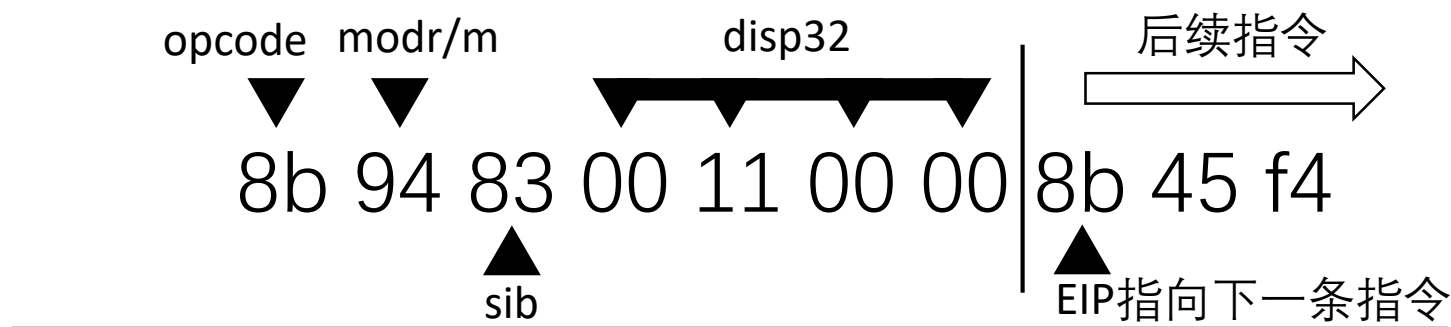


1. 不是0x66, 操作数32位, 0x8b为操作码
2. 查i386手册 – Appendix A – 0x8b对应MOV Gv, Ev
3. Ev和Gv都说明后面跟ModR/M字节
4. 根据Mod + R/M域决定有SIB字节 (内存地址disp32[--][--])
5. SIB字节后面自然还有disp32 – 32位的偏移量 (小端方式)



# 指令的编码方式

- 理解指令译码的过程  
(假设此时EIP取初始值, 为某程序第一条指令)



1. 不是0x66, 操作数32位, 0x8b为操作码
2. 查i386手册 – Appendix A – 0x8b对应MOV Gv, Ev
3. Ev和Gv都说明后面跟ModR/M字节
4. 根据Mod + R/M域决定有SIB字节 (内存地址disp32[--][--])
5. SIB字节后面自然还有disp32 – 32位的偏移量 (小端方式)
6. 该指令所有需要的信息已经获得, 对应AT&T格式汇编:

movl 0x1100(%ebx, %eax, 4), %edx

# 指令的编码方式

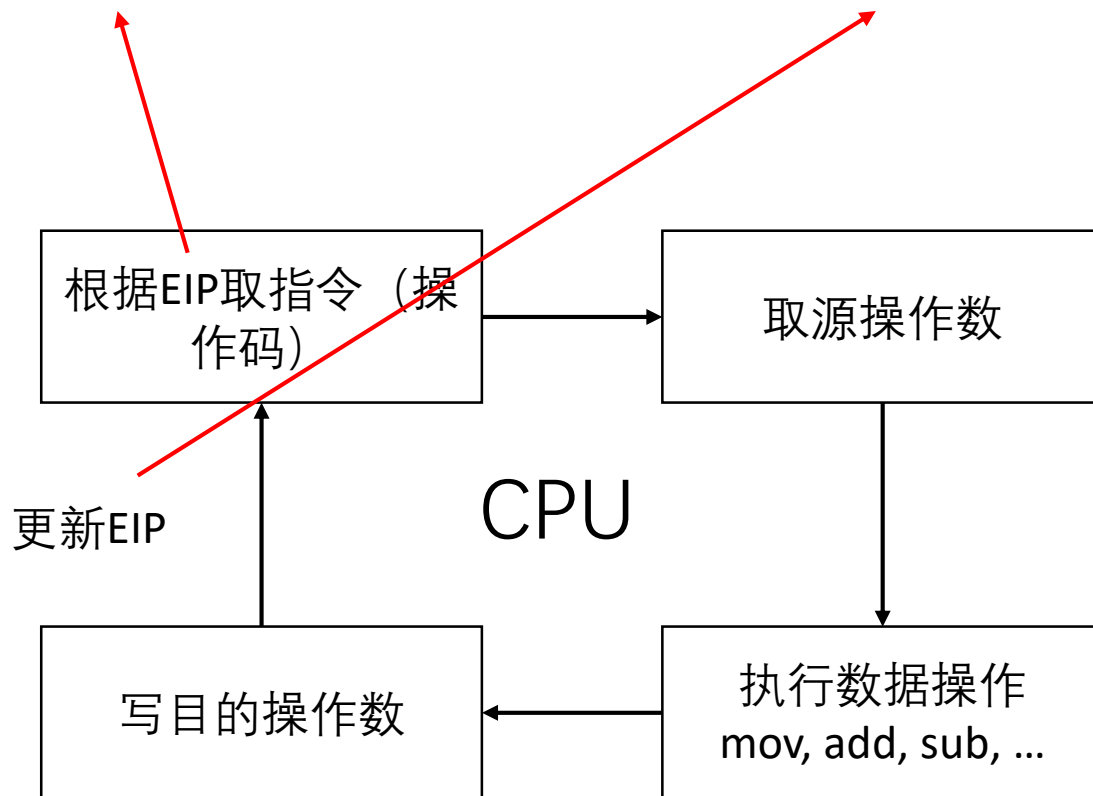
- X86有多少条指令？
  - I386手册Chapter 17
  - 手册上没有的：<http://www.felixcloutier.com/x86/>
- 指令怎么那么多？结构怎么这么复杂？
  - 复杂指令集计算机（CISC）：Intel, AMD为代表
    - 程序员选择多
    - 优化方案多
    - 可扩展性强，向下兼容性好
- 有没有简单点的？
  - 精简指令集计算机（RISC）：MIPS为代表
    - CPU实现简单，嵌入式系统欢迎
  - 一条指令集计算机（One Instruction Set Computers, OISC）
    - [https://en.wikipedia.org/wiki/One\\_instruction\\_set\\_computer](https://en.wikipedia.org/wiki/One_instruction_set_computer)

它们的计算能力是等价的（学完数理逻辑、图灵机会明白）



# 指令序列的执行流程

内存：8b 94 83 00 11 00 00 8b 45 f4



先把指令依次在内存中排好，给EIP赋一个初始值，指向第一条指令，CPU就可以循环执行每一条指令了

# NEMU模拟指令译码和执行

# NEMU模拟指令译码和执行

假设此时指令已经在内存中排好了，  
EIP初始化为第一条指令的地址，谁干的？我们最后讲

- 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  
}
```

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

怎么从main走到这个函数的？我们最后讲

循环执行指令

执行一条指令

3. 根据指令长度更新EIP，指向下一条指令

1. 取指令

2. 模拟执行

# NEMU模拟指令译码和执行

- `opcode_entry`是一个函数指针数组
  - 其中每一个元素指向一条指令的模拟函数

访问 `opcode_entry[opcode]` == 调用对应位置指向的函数  
实现某一条指令的功能  
*nemu/src/cpu/decode/opcode.c*

```
#include "cpu/instr.h"  
instr_func opcode_entry[256] = { ... }
```

*nemu/include/cpu/instr\_helper.h*

```
// the type of an instruction entry  
typedef int (*instr_func)(uint32_t eip, uint8_t opcode);
```

# 内存： C7 05 48 11 10 00 02 00 00 00

▲  
当前EIP

mov\_i2rm\_v  
是模拟C7指  
令的函数

nemu/src/cpu/instr/mov.c

```
make_instr_func(mov_i2rm_v) {  
    OPERAND rm, imm;  
    rm.data_size = data_size;  
    int len = 1;  
    len += modrm_rm(eip + 1, &rm);  
  
    imm.type = OPR_IMM;  
    imm.addr = eip + len;  
    imm.data_size = data_size;  
  
    operand_read(&imm);  
    rm.val = imm.val;  
    operand_write(&rm);  
  
    return len + data_size / 8;  
}
```

5. 循环开启下一  
条指令

```
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;  
    opcode = instr_fetch(cpu.eip, 1);  
    int len = opcode_entry[opcode](cpu.eip, opcode);  
    return len; // 返回指令长度  
}
```

1. cpu.eip指向C7

2. Opcode取出为C7

3. 访问数组即函数调用

4. 返回指令长度

```
#include "cpu/instr.h"  
instr_func opcode_entry[256] = {  
    ...  
    /* 0xc4 - 0xc7 */    inv, inv, mov_i2rm_b, mov_i2rm_v,  
    ...  
}
```

nemu/src/cpu/decode/opcode.c

# NEMU模拟指令译码和执行

- 所以PA 2-1要做的任务：执行make run或make test

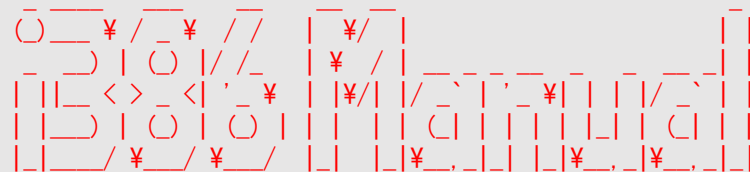
invalid opcode(eip = 0x00030033): 83 f8 01 66 c7 05 34 12 ...

There are two cases which will trigger this unexpected exception:

1. The instruction at eip = 0x00030033 is not implemented.
2. Something is implemented incorrectly.

Find this eip value(0x00030033) in the disassembling result to distinguish which case it is.

If it is the first case, see



for more details.

If it is the second case, remember:

- \* The machine is always right!
- \* Every line of untested code is always wrong!

1. 写该操作码对应的instr\_func
2. 把这个函数在nemu/include/cpu/instr.h中声明一下
3. 在opcode\_entry对应该操作码的地方把这个函数的函数名填进去替代原来的inv
4. 重复上述过程直至完成所有需要模拟的指令



# NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr\_func ?

```
// 宏展开后这一行即为 int mov_i2rm_v(uint32_t eip, uint8_t opcode) {
make_instr_func(mov_i2rm_v) {
    OPERAND rm, imm;      // OPERAND定义在nemu/include/cpu/operand.h
                           // 看教程§2-1.2.3

    rm.data = imm.val;    // 立即数放入寄存器，表示操作数的地址长度
    int len = 0;
    len += 1;

    imm.data = 0;
    imm.data = 0;
    imm.data = 0;

    operand_write(&rm);

    return len + data_size / 8; // opcode长度 + ModR/M字节扫描长度 + 立即数长度
}
```

这是一条把一个立即数mov到R/M中的指令，操作数长度为16或32位

推荐命名规则：

指令名\_源操作数类型2目的操作数类型\_长度后缀

# NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr\_func ?

```
// 宏展开后这一行即为 int mov_i2rm_v(uint32_t eip, uint8_t opcode) {  
make_instr_func(mov_i2rm_v) {  
    OPERAND rm, imm;      // OPERAND定义在nemu/include/cpu/operand.h  
                           // 看教程§2-1.2.3  
    rm.data_size = data_size; // data_size是个全局变量, 表示操作数的比特长度  
    int len = 1;           // opcode 长度1字节  
    len += modrm_rm(eip + 1, &rm); // 读ModR/M字节, rm的type和addr会被填写  
  
    imm.type = OPR_IMM;    // 填入立即数类型  
    imm.addr = eip + len;  // 找到立即数的地址  
    imm.data_size = data_size;  
  
    operand_read(&imm);    // 执行 mov 操作  
    rm.val = imm.val;  
    operand_write(&rm);  
  
    return len + data_size / 8; // opcode长度 + ModR/M字节扫描长度 + 立即数长度  
}
```

# NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr\_func ?

// 宏展开后这一行即为 `int mov_i2rm_v(uint32_t eip, uint8_t opcode) {`

`make_instr_func(mov_i2rm_v) {`

`OPERAND rm, imm; // OPERAND定义在nemu/include/cpu/operand.h`  
`// 看教程§2-1.2.3`

`rm.d`  
`int le`  
`len +`

`nemu/include/cpu/instr_helper.h`

`#define make_instr_func(name) int name(uint32_t eip, uint8_t opcode)`

`imm.type = OPR_IMM; // 填入立即数类型`  
`imm.addr = eip + len; // 找到立即数的地址`  
`imm.data_size = data_size;`

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

`return len + data_size / 8; // opcode长度 + ModR/M字节扫描长度 + 立即数长度`

`}`

# NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr\_func ?

```
// 宏展开后这一行即为 int mov_i2rm_v(uint32_t eip, uint8_t opcode) {  
make_instr_func(mov_i2rm_v) {  
    OPERAND rm, imm;          // OPERAND定义在nemu/include/cpu/operand.h  
                                // 看教程§2-1.2.3  
    rm.data_size = data_size; // data_size是个全局变量, 表示操作数的比特长度  
    int len = 1;              // opcode 长度1字节  
    len += modrm_rm(eip + 1, &rm); // 读ModR/M字节, rm的type和addr会被填写  
  
    imm.type = OPR_IMM;       // 填入立即数类型  
    imm.addr = eip + len;     // 找到立即数的地址  
    imm.data_size = data_size;  
  
    operand_read(&imm);       // 执行 mov 操作  
    rm.val = imm.val;  
    operand_write(&rm);  
  
    return len + data_size / 8; // opcode长度 + ModR/M字节扫描长度 + 立即数长度  
}
```

# NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr\_func ?

```
// 宏展开后这一行即为 int mov_i2rm_v(
make_instr_func(mov_i2rm_v) {
    OPERAND rm, imm;          // OPERAND定
                                // 看教程§2-1
    rm.data_size = data_size; // data_size是
    int len = 1;               // opcode 长度
    len += modrm_rm(eip + 1, &rm); // 读M

    imm.type = OPR_IMM;        // 填入立即
    imm.addr = eip + len;      // 找到立即
    imm.data_size = data_size;

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

    return len + data_size / 8; // opcode长
}
```

nemu/include/cpu/operand.h

```
enum {OPR_IMM, OPR_REG, OPR_MEM,
      OPR_CREG, OPR_SREG};
```

```
typedef struct {
```

```
    int type;
```

// addr地址, 随type不同解释也不同

```
    uint32_t addr;
```

```
    uint8_t sreg; // 现在不管
```

```
    uint32_t val;
```

// data\_size = 8, 16, 32

```
    size_t data_size;
```

```
#ifdef DEBUG
```

```
    MEM_ADDR mem_addr;
```

```
#endif
```

```
} OPERAND;
```

# NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr\_func ?

```
// 宏展开后这一行即为 int mov_i2rm_v(uint32_t eip, uint8_t opcode) {  
make_instr_func(mov_i2rm_v) {  
    OPERAND rm, imm;          // OPERAND定义在nemu/include/cpu/operand.h  
                                // 看教程§2-1.2.3  
    rm.data_size = data_size; // data_size是个全局变量, 表示操作数的比特长度  
    int len = 1;              // opcode 长度1字节  
    len += modrm_rm(eip + 1, &rm); // 读ModR/M字节, rm的type和addr会被填写  
  
    imm.type = OPR_IMM;       // 填入立即数类型  
    imm.addr = eip + len;     // 找到立即数的地址  
    imm.data_size = data_size;  
  
    operand_read(&imm);       // 执行 mov 操作  
    rm.val = imm.val;  
    operand_write(&rm);  
  
    return len + data_size / 8; // opcode长度 + ModR/M字节扫描长度 + 立即数长度  
}
```

# NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr\_func ?

// 宏展开后这一行即为 `int mov_i2rm_v(uint32_t eip, uint8_t opcode) {`

`make_instr_func(mov_i2rm_v) {`

`OPERAND rm, imm;`

`rm.data_size = data_size;`

`int len = 1;`

`len += modrm_rm(eip + 1,`

`imm.type = OPR_IMM;`

`imm.addr = eip + len;`

`imm.data_size = data_size;`

`operand_read(&imm);`

`rm.val = imm.val;`

`operand_write(&rm);`

`return len + data_size / 8;`

`}`

`nemu/src/cpu/instr/data_size.c`

`uint8_t data_size = 32;`

`make_instr_func(data_size_16) {`

`uint8_t op_code = 0;`

`int len = 0;`

`data_size = 16;`

`op_code = instr_fetch(eip + 1, 1);`

`len = opcode_entry[op_code](eip + 1, op_code);`

`data_size = 32;`

`return 1 + len;`

`}`

// opcode长度 + ModR/M字节扫描长度 + 立即数长度

# NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr\_func ?

```
// 宏展开后这一行即为 int mov_i2rm_v(uint32_t eip, uint8_t opcode) {  
make_instr_func(mov_i2rm_v) {  
    OPERAND rm, imm;          // OPERAND定义在nemu/include/cpu/operand.h  
                                // 看教程§2-1.2.3  
    rm.data_size = data_size; // data_size是个全局变量, 表示操作数的比特长度  
    int len = 1;              // opcode 长度1字节  
    len += modrm_rm(eip + 1, &rm); // 读ModR/M字节, rm的type和addr会被填写  
  
    imm.type = OPR_IMM;       // 填入立即数类型  
    imm.addr = eip + len;     // 找到立即数的地址  
    imm.data_size = data_size;  
  
    operand_read(&imm);       // 执行 mov 操作  
    rm.val = imm.val;  
    operand_write(&rm);  
  
    return len + data_size / 8; // opcode长度 + ModR/M字节扫描长度 + 立即数长度  
}
```



# NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr\_func ?

```
// 宏展开后这一行即为 int mov_i2rm_v(uint32_t eip, uint8_t opcode) {  
make_instr_func(mov_i2rm_v) {  
    OPERAND rm, imm;          // OPERAND定义在nemu/include/cpu/operand.h  
                                // 看教程§2-1.2.3  
    rm.data_size = data_size; // data_size是个全局变量, 表示操作数的比特长度  
    int len = 1;              // opcode 长度1字节  
    len += modrm_rm(eip + 1, &rm); // 读ModR/M字节, rm的type和addr会被填写  
  
    imm.type = OPR_IMM;      // 填  
    imm.addr = eip + len;    // 找  
    imm.data_size = data_size;  
  
    operand_read(&imm);      // 执  
    rm.val = imm.val;  
    operand_write(&rm);  
  
    return len + data_size / 8; // opcode长度 + ModR/M字节扫描长度 + 立即数长度  
}
```

[nemu/src/cpu/decode/modrm.c](#)

```
int modrm_rm(uint32_t eip, OPERAND * rm);
```

就是查表过程变成代码  
会将传入的rm的type和addr（包括sreg）填好

# NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr\_func ?

```
// 宏展开后这一行即为 int mov_i2rm_v(uint32_t eip, uint8_t opcode) {  
make_instr_func(mov_i2rm_v) {  
    OPERAND rm, imm;          // OPERAND定义在nemu/include/cpu/operand.h  
                                // 看教程§2-1.2.3  
    rm.data_size = data_size; // data_size是个全局变量, 表示操作数的比特长度  
    int len = 1;              // opcode 长度1字节  
    len += modrm_rm(eip + 1, &rm); // 读ModR/M字节, rm的type和addr会被填写  
  
    imm.type = OPR_IMM;       // 填入立即数类型  
    imm.addr = eip + len;     // 找到立即数的地址  
    imm.data_size = data_size;  
  
    operand_read(&imm);       // 执行 mov 操作  
    rm.val = imm.val;  
    operand_write(&rm);  
  
    return len + data_size / 8; // opcode长度 + ModR/M字节扫描长度 + 立即数长度  
}
```

# NEMU模拟指令译码和执行

nemu/src/cpu/decode/operand.c

```
void operand_read(OPERAND * opr) {
    switch(opr->type) {
        case OPR_MEM: ...
        case OPR_IMM:
            opr->val = vaddr_read(opr->addr, SREG_CS, 4);
            break;
        case OPR_REG:
            if(opr->data_size == 8) {
                opr->val = cpu.gpr[opr->addr % 4]._8[opr->addr / 4];
            } else {
                opr->val = cpu.gpr[opr->addr]._32;
            }
            break;
        case OPR_CREG: ...
        case OPR_SREG: ...

        // deal with data size
        switch(opr->data_size) {
            case 8: opr->val = opr->val & 0xff; break;
            case 16: opr->val = opr->val & 0xffff; break;
            case 32: break;
            default: ...
        }
    }
}
```

# NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr\_func ?

```
// 宏展开后这一行即为 int mov_i2rm_v(uint32_t eip, uint8_t opcode) {  
make_instr_func(mov_i2rm_v) {  
    OPERAND rm, imm;          // OPERAND定义在nemu/include/cpu/operand.h  
                                // 看教程§2-1.2.3  
    rm.data_size = data_size; // data_size是个全局变量, 表示操作数的比特长度  
    int len = 1;              // opcode 长度1字节  
    len += modrm_rm(eip + 1, &rm); // 读ModR/M字节, rm的type和addr会被填写  
  
    imm.type = OPR_IMM;        // 填入立即数类型  
    imm.addr = eip + len;      // 找到立即数的地址  
    imm.data_size = data_size;  
  
    operand_read(&imm);        // 执行 mov 操作  
    rm.val = imm.val;  
    operand_write(&rm);  
  
    return len + data_size / 8; // opcode长度 + ModR/M字节扫描长度 + 立即数长度  
}
```

执行mov操作并且  
写目的操作数

# NEMU模拟指令译码和执行

- 怎么写某操作码对应的instr\_func ?

```
// 宏展开后这一行即为 int mov_i2rm_v(uint32_t eip, uint8_t opcode) {  
make_instr_func(mov_i2rm_v) {  
    OPERAND rm, imm;          // OPERAND定义在nemu/include/cpu/operand.h  
                                // 看教程§2-1.2.3  
    rm.data_size = data_size; // data_size是个全局变量, 表示操作数的比特长度  
    int len = 1;              // opcode 长度1字节  
    len += modrm_rm(eip + 1, &rm); // 读ModR/M字节, rm的type和addr会被填写  
  
    imm.type = OPR_IMM;        // 填入立即数类型  
    imm.addr = eip + len;      // 找到立即数的地址  
    imm.data_size = data_size;  
  
    operand_read(&imm);        // 执行 mov 操作  
    rm.val = imm.val;  
    operand_write(&rm);  
  
    return len + data_size / 8; // opcode长度 + ModR/M字节扫描长度 + 立即数长度  
}
```

返回指令长度

# NEMU模拟指令译码和执行

- 查表得知

- `mov_i2rm_v`其实是对应操作码为0xC7的指令
- 在实现完`mov_i2rm_v`的函数体之后

1. 把`mov_i2rm_v`在`nemu/include/cpu/instr.h` (或者自己创建一个`mov.h`) 中声明一下 (或者在`instr.h`中`include mov.h`)
2. 把`mov_i2rm_v`填入`opcode_entry[0xC7]`这个位置, 替换原来的`inv`

# NEMU模拟指令译码和执行

- 针对这个框架有一些要特别注意的地方

*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--;  
        ...  
    }  
    ...  
}
```

这一步非常机械，对于某些指令，如特殊的jmp、ret中涉及到跳转到某一个绝对的地址（而非相对下一条指令起始地址的偏移量）时，要在实现时灵活指定指令长度为0，来规避  
`cpu.eip += instr_len`

# NEMU模拟指令译码和执行

- 精简指令实现的宏

	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

大量指令操作相同，只是操作数的类型和长度不同。



# NEMU模拟指令译码和执行

- 精简指令实现的宏
  - 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
- 但是代码复制是很糟糕的！
  - `alu_test.c` – bad example!

## 20170911有关div测试代码的修正说明

在原框架代码中的nemu/src/cpu/test/alu\_test.c中针对div的测试用例，在随机测试部分误用了针对idiv的测试代码。修复方案为改为针对div的测试代码。具体请参见群文件：20170911有关div测试代码的修正说明.txt

讲师 汪亮 发表于 09-11 20:50 86人已读

你猜我是怎么写错的？

# NEMU模拟指令译码和执行

- 精简指令实现的宏：许多指令的实现流程固定
  1. 声明操作数OPERAND
  2. 设置操作数长度data\_size
  3. 根据操作数类型进行解码 decode
  4. 进行数据操作（能否同类指令共享，不同指令区分定制？）
  5. 返回指令长度

# NEMU模拟指令译码和执行

- 于是在[nemu/include/cpu/instr\\_helper.h](#)中我们给出了用于精简指令实现的宏

```
#define make_instr_impl_1op(inst_name, src_type, suffix) ...  
#define make_instr_impl_1op_cc(inst_name, src_type, suffix, cc) ...  
#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) ...  
#define make_instr_impl_2op_cc(inst_name, src_type, dest_type, suffix, cc) ...
```

还有

decode\_data\_size系列

decode\_operand系列

condition系列

宏在预处理阶段被gcc处理，本质就是字符串替换，拿右边的替换左边的，换行必须打上\

# NEMU模拟指令译码和执行

- 于是在 `nemu/include/cpu/instr_helper.h` 中我们给出了用于精简指令实现的宏：举个例子

源操作数类型      长度后缀  
指令名称          目的操作数类型

```
#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) \  
    make_instr_func(concat7(inst_name, _, src_type, 2, dest_type, _, suffix)) { \// 拼接得到 mov_i2rm_v  
        int len = 1; \// opcode 占一字节  
        concat(decode_data_size_, suffix) \  
        concat3(decode_operand, _, concat3(src_type, 2, dest_type)) \  
        print_asm_2(...); \// 打印调试信息  
        instr_execute_2op(); \  
        return len; \  
    }
```

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

# NEMU模拟指令译码和执行

- 于是在 `nemu/include/cpu/instr_helper.h` 中我们给出了用于精简指令实现的宏：举个例子

源操作数类型      长度后缀  
指令名称          目的操作数类型

```
#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) \  
    make_instr_func(concat7(inst_name, _, src_type, 2, dest_type, _, suffix)) { //拼接得到mov_i2rm_v  
        int len = 1; //opcode占一字节  
        concat(decode_data_size_, suffix) \  
        concat3(decode_operand, _, concat3(src_type, 2, dest_type)) \  
        print_asm_2(...); //打印调试信息  
        instr_execute_2op(); \  
        return len; \  
    }
```

```
#define decode_data_size_v opr_src.data_size = opr_dest.data_size = data_size;
```

两个全局OPERAND类型的变量，免去创建局部变量的开销

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

# NEMU模拟指令译码和执行

- 于是在 `nemu/include/cpu/instr_helper.h` 中我们给出了用于精简指令实现的宏：举个例子

源操作数类型      长度后缀

指令名称          目的操作数类型

```
#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) \
    make_instr_func(concat7(inst_name, _, src_type, 2, dest_type, _, suffix)) { //拼接得到mov_i2rm_v
        int len = 1; //opcode占一字节
        concat(decode_data_size_, suffix) \
        concat3(decode_operand, _, concat3(src_type, 2, dest_type)) \
        print_asm_2(...); // 打印调试信息
        instr_execute_2op(); \
        return len; \
    }
```

解码两个操作数并增加len

```
#define decode_operand_i2rm \
    len += modrm_rm(eip + 1, &opr_dest); \
    opr_src.type = OPR_IMM; \
    opr_src.sreg = SREG_CS; \
    opr_src.addr = eip + len; \
    len += opr_src.data_size / 8;
```

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

# NEMU模拟指令译码和执行

- 于是在 `nemu/include/cpu/instr_helper.h` 中我们给出了用于精简指令实现的宏：举个例子

源操作数类型      长度后缀

指令名称          目的操作数类型

```
#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) \
    make_instr_func(concat7(inst_name, _, src_type, 2, dest_type, _, suffix)) {\ //拼接得到mov_i2rm_v
        int len = 1; \ //opcode占一字节
        concat(decode_data_size_, suffix) \
        concat3(decode_operand, _, concat3(src_type, 2, dest_type)) \
        print_asm_2(...); \ // 打印调试信息
        instr_execute_2op(); \
        return len; \
    }
```

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

执行函数写在 `mov.c` 中，根据指令的具体操作来实现，`static` 不可少！

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

# NEMU模拟指令译码和执行

- 于是在[nemu/include/cpu/instr\\_helper.h](#)中我们给出了用于精简指令实现的宏：举个例子

```
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)
```

[nemu/src/cpu/instr/mov.c](#)



# NEMU模拟指令译码和执行

- 于是在`nemu/include/cpu/instr_helper.h`中我们给出了用于精简指令实现的宏，一些实用信息（详细用法参阅教程，比较详尽）

`#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) ...`

- `inst_name`就是指令的名称：mov, add, sub, ...
- `src_type`和`dest_type`是源和目的操作数类型，与`decode_operand`系列宏一致：
  - rm – 寄存器或内存地址 – 对应手册E类型
  - r – 寄存器地址 – 对应手册G类型
  - i – 立即数 – 对应手册I类型
  - m – 内存地址 – 差不多对应手册M类型
  - a – 根据操作数长度对应al, ax, eax – 手册里没有
  - c – 根据操作数长度对应cl, cx, ecx – 手册里没有
  - o – 偏移量 – 差不多对应手册里的O类型
- `suffix`是操作数长度后缀，与`decode_data_size`系列宏一致：
  - b, w, l, v – 8, 16, 32, 16/32位
  - bv – 源操作数为8位，目的操作数为16/32位，特殊指令用到
  - short, near – jmp指令用到，分别指代8位和32位

你可以根据实际需要添加其他的宏或改写已有的宏

# NEMU模拟指令译码和执行

假设此时指令已经在内存中排好了，  
EIP初始化为第一条指令的地址，谁干的？我们最后讲

- 指令循环：一条接一条的执行指令

*nemu/src/cpu/cpu.c*

怎么从main  
走到这个函  
数的？我们  
最后讲

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

循环执行指令

回答最后的问题

```
int exec_inst() {  执行一条指令  
    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  
}
```

# NEMU运行测试用例

# NEMU运行测试用例

- 从 **make run** 说起

## Makefile

```
nemu:  
    $(call git_commit, "nemu")  
    cd nemu && make
```

Makefile中的第一个目标，当命令不带目标（就单纯make）时，默认执行的目标

目标

```
run: nemu do_not_call_me_testcase  
    $(call git_commit, "run")  
    ./nemu/nemu -run mov
```

前提依赖（应先于该目标达成的目标）

在前提依赖被满足的情况下，实现目标的额外步骤

# NEMU运行测试用例

- 从make run说起

## Makefile

```
nemu:  
    $(call git_commit, "nemu")  
    cd nemu && make
```

目标

```
run: nemu do_not_call_me_testcase  
    $(call git_commit, "run")  
    ./nemu/nemu -run mov
```

前提依赖（编译  
nemu和testcase）

1. 打上git记录

2. 执行./nemu/nemu -run mov

- 其中mov是测试用例的名称，对应testcase/src/里面测试用例源文件的文件名

# NEMU运行测试用例

- 从make run说起
- 执行./nemu/nemu -run mov

nemu/src/main.c

```
int main(int argc, char* argv[]) {  
  
    if(argc == 1) { reg_test(); alu_test(); fpu_test(); return 0; }  
  
    /* Read the arguments */  
    if(argc == 3) {  
        if(strcmp(argv[1], "-run")) {  
            printf("Error: %s %s %s\n", argv[0], argv[1], argv[2]);  
            printf("Usage: nemu -run <testcase>\n");  
            return 0;  
        }  
  
        strcpy(image_path, "./testcase/bin/");  
        strcat(image_path, argv[2]);  
        strcat(image_path, ".img");  
        strcpy(elf_path, "./testcase/bin/");  
        strcat(elf_path, argv[2]);  
        single_run();  
    }  
    ...  
}
```

内存镜像路径

elf文件路径  
(现在不管)

# NEMU运行测试用例

- 从make run说起
- 执行./nemu/nemu -run mov

nemu/src/main.c

```
static void single_run() {  
    ...  
    restart(INIT_EIP);  
    load_image(image_path, LOAD_OFF);  
    ...  
    load_image(elf_path, 0);  
    ...  
    ui_mainloop(autorun);  
}
```

# NEMU运行测试用例

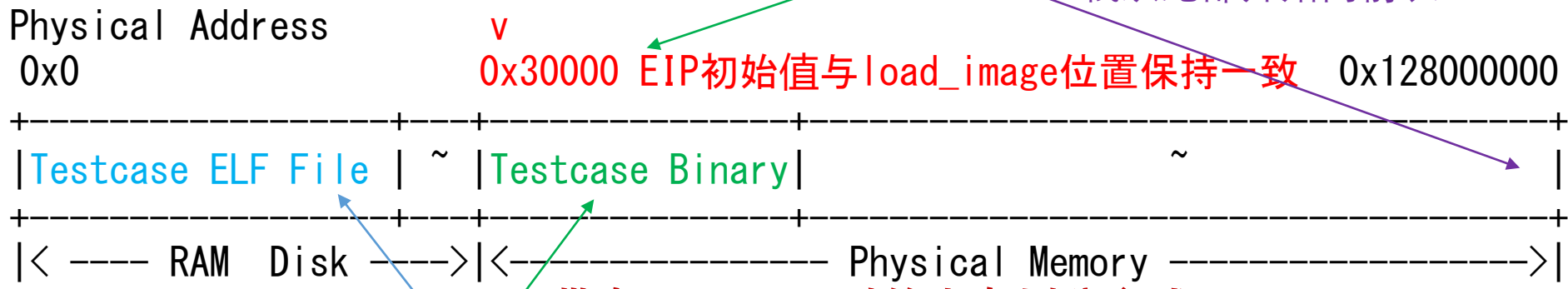
- 从make run说起
- 执行./nemu/nemu -run mov

```
single_run() -> restart() -> init_cpu() {  
    cpu.eip = init_eip; // 0x30000 和 testcase/Makefile中-Ttext=0x30000配合  
                        // 同时和load_image(image_path, LOAD_OFF)的位置配合  
    cpu.esp = (128 << 20) - 16;  
}
```

cpu.c

Initial EIP 指向第一条指令

栈从尾部开始向前长



0x30000 EIP初始值与load\_image位置保持一致

带有RAM Disk时的内存划分方式

```
single_run() -> load_image () {  
    ...  
    fread(...);  
    ...  
}
```

main.c

PA 2-1只看绿线  
PA 2-2看蓝线



# NEMU运行测试用例

- 从make run说起
- 执行./nemu/nemu -run mov

nemu/src/main.c

```
static void single_run() {  
    ...  
    restart(INIT_EIP);  
    load_image(image_path, LOAD_OFF);  
    ...  
    load_image(elf_path, 0);  
    ...  
    ui_mainloop(autorun);  
}
```

# NEMU运行测试用例

- 从 **make run** 说起
- 执行 `./nemu/nemu -run mov`
- 进入 `ui_mainloop()`, `autorun` 为 `false` 进入字符交互界面, 为 `true` 则不进入交互界面, 自动开始运行 (`make test` 就这么实现)

```
main() {  
    single_run();  
}
```

**main.c**

```
single_run() {  
    // restart()初始化EIP  
    // load_image()绿的那个把指令在内从中排好  
    ui_mainloop();  
}
```

```
void ui_mainloop(bool autorun) {
```

...

```
while(true) {
```

通过readline拿到用户命令  
查cmd\_table看是什么命令  
调用命令对应的handler

```
}
```

```
}
```

**c命令**

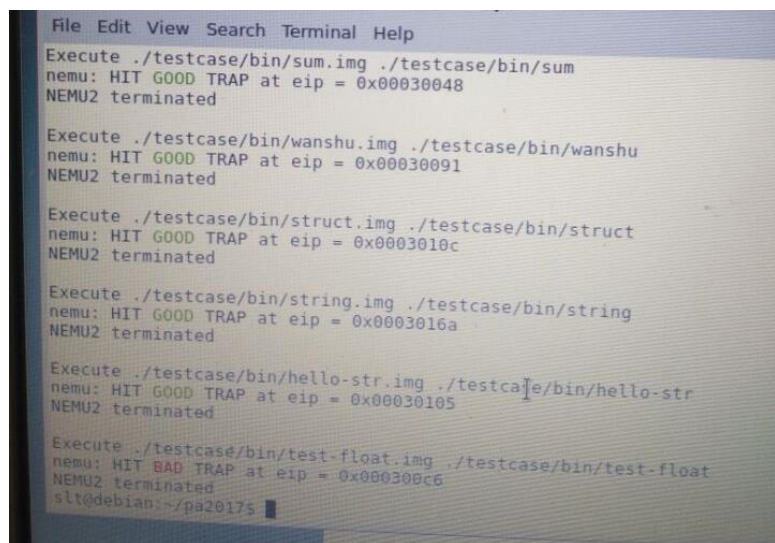
```
cmd_handler(cmd_c) {  
    // execute the program  
    exec(-1); // 进入cpu.c  
    return 0;  
}
```

**nemu/src/monitor/ui.c**

其他丰富的调试命令参见教程§2-1.2.1或阅读源代码, `EXPR`要用起来参见教程可选任务PA 2-3

# 实验目标

大佬！



```
File Edit View Search Terminal Help
Execute ./testcase/bin/sum.img ./testcase/bin/sum
nemu: HIT GOOD TRAP at eip = 0x00030048
NEMU2 terminated

Execute ./testcase/bin/wanshu.img ./testcase/bin/wanshu
nemu: HIT GOOD TRAP at eip = 0x00030091
NEMU2 terminated

Execute ./testcase/bin/struct.img ./testcase/bin/struct
nemu: HIT GOOD TRAP at eip = 0x0003010c
NEMU2 terminated

Execute ./testcase/bin/string.img ./testcase/bin/string
nemu: HIT GOOD TRAP at eip = 0x0003016a
NEMU2 terminated

Execute ./testcase/bin/hello-str.img ./testcase/bin/hello-str
nemu: HIT GOOD TRAP at eip = 0x00030105
NEMU2 terminated

Execute ./testcase/bin/test-float.img ./testcase/bin/test-float
nemu: HIT BAD TRAP at eip = 0x000300c6
NEMU2 terminated
slt@debian:~/pa2017$
```

- PA 2-1提交截止时间
  - 2017年10月19日24时
  - 一个月的时间
- 有时间可以（应该）先做 PA 2-3.1

PA 2-1到此结束

**祝大家学习快乐，身心健康！**

欢迎大家踊跃参加问卷调查

（量表一、二、三到PA 2截止时做一次，如果PA 2-1截止的时候也能做一次最好）