南京大学


南京大学计算机科学与技术系

计算机系统基础
Programming Assignment

PA 2-1 – 指令解码与执行

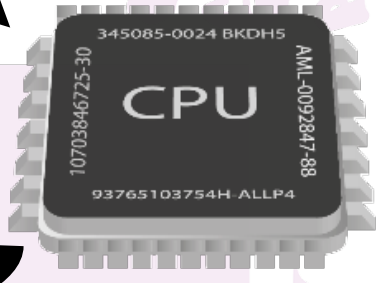
2022年3月24日

南京大学《计算机系统基础》课程组

目录

- 程序执行的宏观过程与模拟
- 单条指令的解码与NEMU实现

计算机执行程序的过程



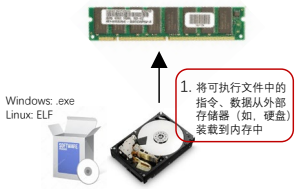
2. 循环往复地取指令、取操作数、执行、写操作数（若需要写）

1. 将可执行文件中的指令、数据从外部存储器（如，硬盘）装载到内存中

Windows: .exe
Linux: ELF

PA 2-2 深入探讨
PA 2-1 简化实现

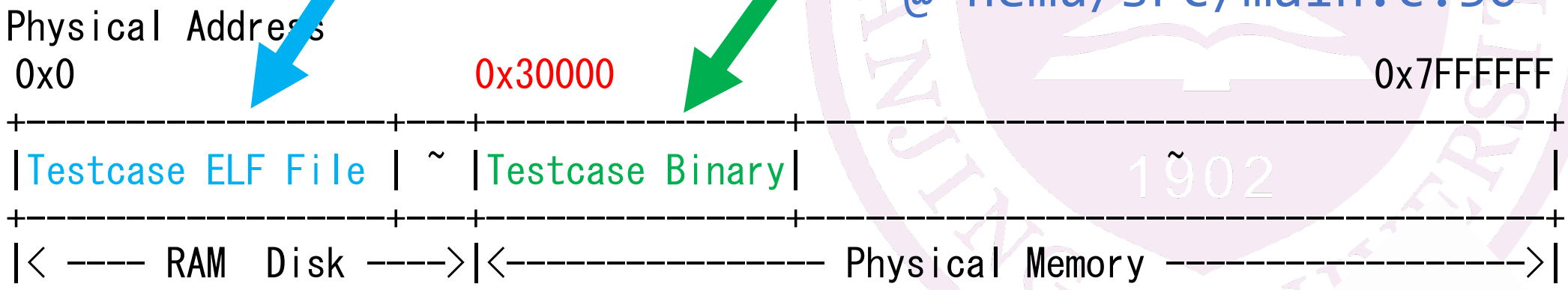
NEMU模拟程序执行：1. 装载程序



1.1 NEMU初始化模拟内存（PA 2-1的装载）

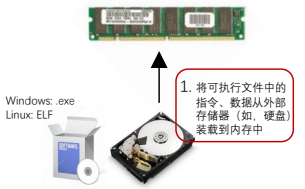
testcase/bin/add　　testcase/bin/add.img

直接拷贝　　load_exec()
@ nemu/src/main.c:30



带有RAM Disk时的NEMU模拟内存划分方式

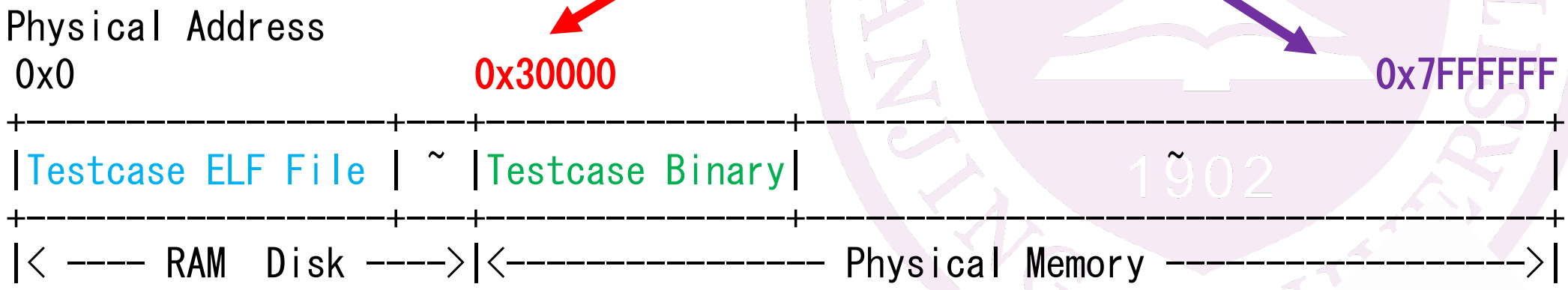
NEMU模拟程序执行：1. 装载程序



1.2 NEMU初始化CPU

```
init_cpu()
@ nemu/src/cpu/cpu.c:17
```

初始化EIP和ESP



带有RAM Disk时的NEMU模拟内存划分方式

nemu/src/cpu/cpu.c

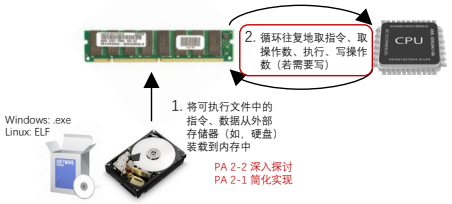
```
void exec(uint32_t n) {
    ...
    while( n > 0 && nemu_state == NEMU_RUN) {
        ...
        instr_len = exec_inst();
        cpu.eip += instr_len;
        n--;
        ...
    }
    ...
}

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
}
```

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

NEMU模拟程序执行：2. 执行程序



nemu/src/cpu/cpu.c

```
void exec(uint32_t n)
{
        …
        nemu_state = NEMU_RUN;
        while (n > 0 && nemu_state == NEMU_RUN)
        {       // 执行eip指向的指令
                instr_len = exec_inst();
                // (
                cpu.(
                n--;
                …
        }
        …
}
```

```
int exec_inst()
{
        uint8_t opcode = 0;
        opcode = instr_fetch(cpu.eip, 1);
        int len = opcode_entry[opcode](cpu.eip, opcode);
        return len;
}
```

解码并执行该指令，返回指令长度

how to?

nemu/src/cpu/cpu.c

目录

- 程序执行的宏观过程与模拟
- 单条指令的解码与NEMU实现

指令的解码

EIP

▼

内存中的指令数据：8b 94 83 00 11 00 00 8b 45 f4

如何理解？怎么模拟？

指令的解码

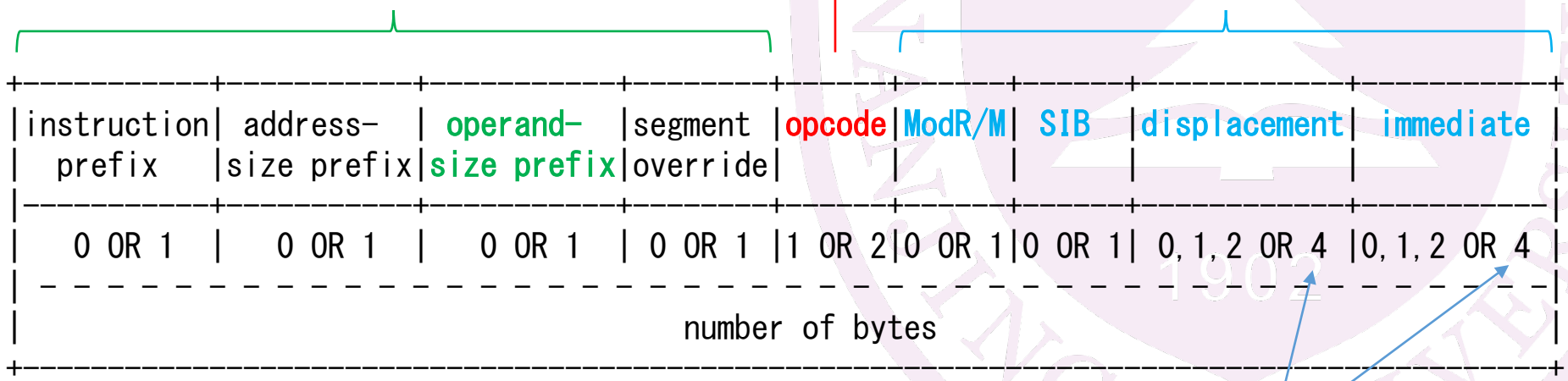
EIP

内存中的指令数据：8b 94 83 00 11 00 00 8b 45 f4

操作码
(ModR/M中也可能包含一点)

各种前缀，我们只模拟operand-size prefix，其值为0x66

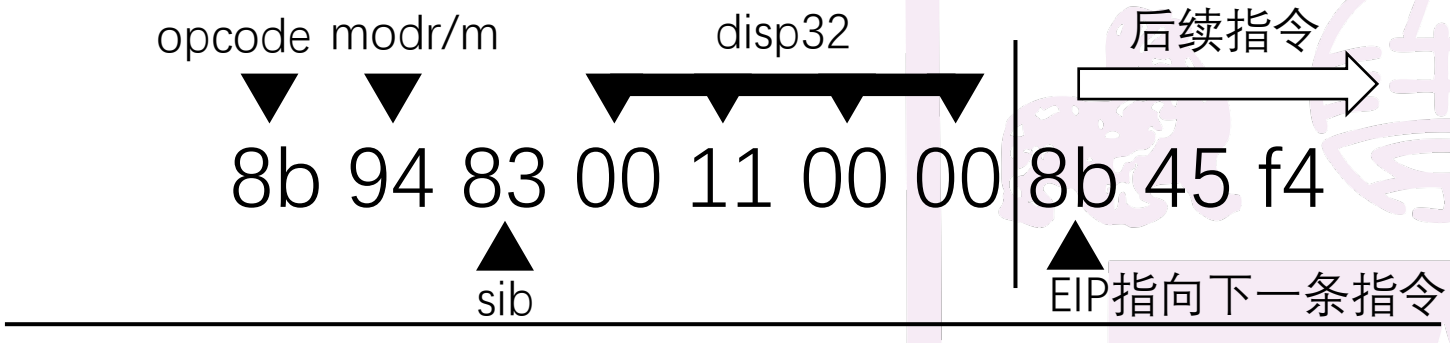
各种找到操作数（寻址）的办法



最大是4，体现我们是32位机

按i386指令集体系结构的规定

指令的解码



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
```

目录

- 程序执行的宏观过程与模拟
- 单条指令的解码与NEMU实现
  - 操作码的解码方式
  - 单条指令的实现方法
  - 利用框架代码中的宏来高效、简洁地实现多条指令

NEMU模拟指令解码和执行

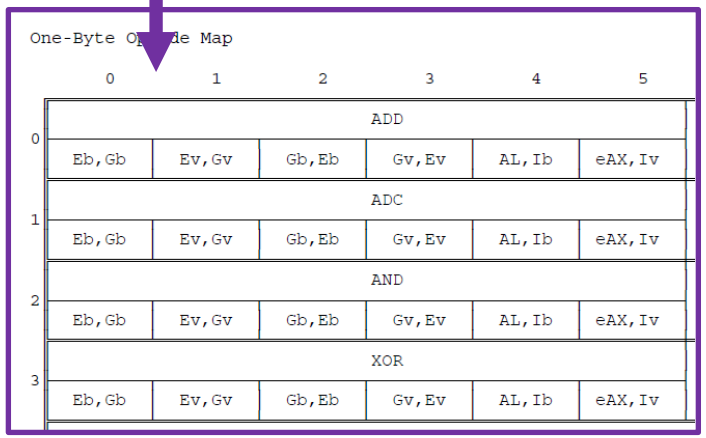
```
void exec(uint32_t n) {
        ...
        while( n > 0 && nemu_state == NEMU_RUN) {
                ...
                instr_len = exec_inst();
                cpu.eip += instr_len;
                n--;
                ...
        }
        ...
}

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
}
```

指令解码与执行

NEMU模拟指令解码和执行

```
switch(opcode)
{
case 0x00: add_r2rm_b(); break;
case 0x01: add_r2rm_v(); break;
…
case 0x10: adc_r2rm_b(); break;
…
case 0x8b: mov_rm2r_v(); break;
…
case 0xc7: mov_i2rm_v(); break;
…
};
```

对应



等价

```
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
}
```

NEMU模拟指令解码和执行

- 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] = { … }
```

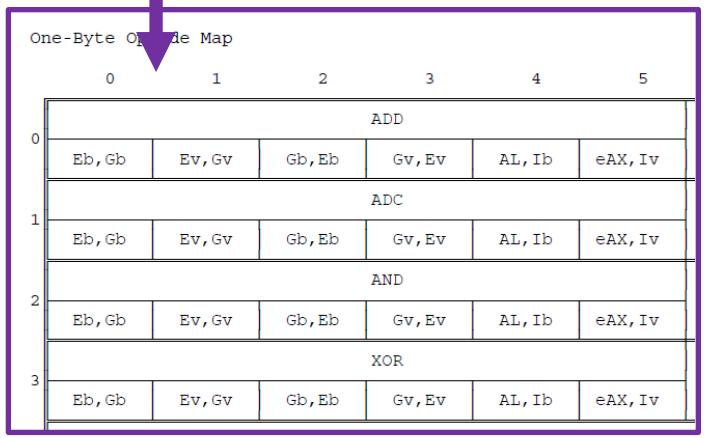
nemu/include/cpu/instr_helper.h

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

NEMU模拟指令解码和执行

```
switch(opcode)
{
case 0x00: add_r2rm_b(); break;
case 0x01: add_r2rm_v(); break;
...
case 0x10: adc_r2rm_b(); break;
...
case 0x8b: mov_rm2r_v(); break;
...
case 0xc7: mov_i2rm_v(); break;
...
};
```

对应



等价

内存： C7 05 48 11 10 00 02 00 00 00

```
instr_func opcode_entry[256] = {
    ...
    /* 0xbc - 0xbf*/ mov_i2r_v, mov_i2r_v, mov_i2r_v, mov_i2r_v,
    /* 0xc0 - 0xc3*/ group_2_b, group_2_v, inv, inv,
    /* 0xc4 - 0xc7*/ inv, inv, mov_i2rm_b, mov_i2rm_v,
    /* 0xc8 - 0xcb*/ inv, inv, inv, inv,
    ...
};
```

```
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
}
```

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;
}
```

4. 返回指令长度

5. 循环开启下一条指令

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;
    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

```
#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

特殊的操作码编码方式

- 双字节opcode：第一个字节是0x0f

```
instr_func opcode_entry[256] = {
    /* 0x08 - 0x0b*/ inv, inv, inv, inv,
    /* 0x0c - 0x0f*/ inv, inv, inv, opcode_2_byte,
    /* 0x10 - 0x13*/ inv, inv, inv, inv,
    /* 0x14 - 0x17*/ inv, inv, inv, inv,
};
```

nemu/src/cpu/decode/opcode.c

```
instr_func opcode_2_byte_entry[256] = {
    /* 0x00 - 0x03*/ inv, group_7, inv, inv,
    /* 0x04 - 0x07*/ inv, inv, inv, inv,
    /* 0x08 - 0x0b*/ inv, inv, inv, inv,
    /* 0x0c - 0x0f*/ inv, inv, inv, inv,
    /* 0x10 - 0x13*/ inv, inv, inv, inv,
    /* 0x14 - 0x17*/ inv, inv, inv, inv,
    ...
};
```

nemu/src/cpu/instr/opcode_2_byte.c

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

extern bool has_prefix;

make_instr_func(opcode_2_byte)
{
        int len = 1;
        has_prefix = true;
        uint8_t op = instr_fetch(eip + 1, 1);
#ifdef NEMU_REF_INSTR
        len += __ref_opcode_2_byte_entry[op](eip + 1, op);
#else
        len += opcode_2_byte_entry[op](eip + 1, op);
#endif
        has_prefix = false;
        return len;
}
```

特殊的操作码编码方式

- group中的指令

```
instr_func opcode_entry[256] = {
    /* 0x7c - 0x7f*/ inv, inv, inv, inv,
    /* 0x80 - 0x83*/ group_1_b, group_1_v, nemu_trap, group_1_bv,
    /* 0x84 - 0x87*/ inv, inv, inv, inv,
};
```

nemu/src/cpu/decode/opcode.c

One-Byte Opcode Map

	0	1	2	3	4
0	ADD				
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib
1	ADC				
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib
2	AND				
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib
3	XOR				
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib
4	INC general register				
	eAX	eCX	eDX	eBX	eSP
5	PUSH general registe				
	eAX	eCX	eDX	eBX	eSP
6	PUSHA	POPA	BOUND Gv,Ma	ARPL Ew,Rw	SEG =FS
7	Short displacement jump of cond				
	JO	JNO	JB	JNB	JZ
8	Immediate Grpl			Grpl	TEST
	Eb,Ib	Ev,Iv		Ev,Iv	Eb,Gb
9	NOP	XCHG word or double-word re			
		eCX	eDX	eBX	eSP

手册上指令格式有bug，以框架代码为准

特殊的操作码编码方式

- group中的指令

MODR/M BYTE

7	6	5	4	3	2	1	0
MOD		REG/OPCODE			R/M		

```
instr_func opcode_entry[256] = {
    /* 0x7c - 0x7f*/ inv, inv, inv, inv,
    /* 0x80 - 0x83*/ group_1_b, group_1_v, nemu_trap, group_1_bv,
    /* 0x84 - 0x87*/ inv, inv, inv, inv,
};
```

nemu/src/cpu/decode/opcode.c

Opcodes determined by bits 5,4,3 of modR/M byte:

Group	mod nnn R/M 000	001	010	011	100	101	110	111
1	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
2	ROL	ROR	RCL	RCR	SHL	SHR		SAR
3	TEST Ib/Iv		NOT	NEG	MUL AL/eAX	IMUL AL/eAX	DIV AL/eAX	IDIV AL/eAX
4	INC Eb	DEC Eb						
5	INC Ev	DEC Ev	CALL Ev	CALL eP	JMP Ev	JMP Ep	PUSH Ev	

One-Byte Opcode Map

	0	1	2	3	4
0	ADD				
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib
1	ADC				
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib
2	AND				
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib
3	XOR				
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib
4	INC general register				
	eAX	eCX	eDX	eBX	eSP
5	PUSH general registe				
	eAX	eCX	eDX	eBX	eSP
6	PUSHA	POPA	BOUND Gv,Ma	ARPL Ew,Rw	SEG =FS
7	Short displacement jump of cond				
	JO	JNO	JB	JNB	JZ
8	Immediate Grpl			Grpl	TEST
	Eb,Ib	Ev,Iv		Ev,Iv	Eb,Gb
9	NOP	XCHG word or double-word re			
		eCX	eDX	eBX	eSP

手册上指令格式有bug，以框架代码为准

特殊的操作码编码方

- group中的指令

```
instr_func opcode_entry[256] = {
    /* 0x7c - 0x7f*/ inv, inv, inv, inv,
    /* 0x80 - 0x83*/ group_1_b, group_1_v, ···,
    /* 0x84 - 0x87*/ inv, inv, inv, inv,
};
```

nemu/src/cpu/decode/opcode.c

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

#define make_group_impl(name)                                   \
        make_instr_func(name)                                   \
        {                                                       \
                uint8_t op_code;                                \
                modrm_opcode(eip + 1, &op_code);                \
                return concat(name, _entry)[op_code](eip, op_code); \
        }

#ifdef NEMU_REF_INSTR
#define make_group_impl_cond make_group_impl_ref
#else
#define make_group_impl_cond make_group_impl
#endif

make_group_impl_cond(group_1_b)
make_group_impl_cond(group_1_v)
```

nemu/src/cpu/instr/group.c

```
/* 0x80 */
instr_func __ref_group_1_b_entry[8] =    {__ref_add_i2rm_b, __ref_or_i2rm_b, __ref_adc_i2rm_b, __ref_sbb_i2rm_b,
__ref_and_i2rm_b, __ref_sub_i2rm_b, __ref_xor_i2rm_b, __ref_cmp_i2rm_b};

/* 0x81 */
instr_func __ref_group_1_v_entry[8] =    {__ref_add_i2rm_v, __ref_or_i2rm_v, __ref_adc_i2rm_v, __ref_sbb_i2rm_v,
__ref_and_i2rm_v, __ref_sub_i2rm_v, __ref_xor_i2rm_v, __ref_cmp_i2rm_v};
```

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;
}
```

4. 返回指令长度

5. 循环开启下一条指令

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;
    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

```
#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

目录

- 程序执行的宏观过程与模拟
- 单条指令的解码与NEMU实现
  - 操作码的解码方式
  - 单条指令的实现方法
  - 利用框架代码中的宏来高效、简洁地实现多条指令

NEMU模拟指令解码和执行

- 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 +                                                    填写

    imm.
    imm.
    imm.

    oper
    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
}
```

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模拟指令解码和执行

- 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 – 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模拟指令解码和执行

```
// 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模拟指令解码和执行

```
// 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) {
```

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;
    oper
    rm.v
    oper

    retu
}
```

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

对比一下
opcode_entry的类型

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

长度

NEMU模拟指令解码和执行

```
// 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模拟指令解码和执行

```
// 宏展开后这一行即为 int mov_i2rm_v(
make_instr_func(mov_i2rm_v) {
    OPERAND rm, imm;          // OPE
                                // 看
    rm.data_size = data_size; // data_
    int len = 1;                // opc
    len += modrm_rm(eip + 1, &rm);

    imm.type = OPR_IMM;       // 填
    imm.addr = eip + len;       // 找
    imm.data_size = data_size;

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

    return len + data_size / 8;    // op
}
```

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模拟指令解码和执行

```
// 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模拟指令解码和执行

```
// 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 +

    imm.type = OPR_IMM;
    imm.addr = eip + len;
    imm.data_size = data_siz

    operand_read(&imm);
    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/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;
}
```

NEMU模拟指令解码和执行

```
// 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模拟指令解码和执行

```
// 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_I
    imm.addr = eip + l
    imm.data_size = da

    operand_read(&im
    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变量的type和addr（包括sreg）填好
返回解析modr/m所扫描过的字节数（包括可能的SIB和disp）

NEMU模拟指令解码和执行

```
// 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模拟指令解码和执行

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: …
        }
}
```

```
// 宏展
make_
    OPE

    rm.c
    int le
    len +

    imm
    imm
    imm

    ope
    rm.v
    ope

    retu
}
```

NEMU模拟指令解码和执行

```
// 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模拟指令解码和执行

```
// 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

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;
}
```

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

nemu/src/cpu/decode/opcode.c

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

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

5. 循环开启下一条指令

容易写出低质量代码：代码克隆

```
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;
}
```

```
make_instr_func(mov_r2rm_v) {
    OPERAND rm, r;
    rm.data_size = data_size;
    r.data_size = data_size
    int len = 1;
    len += modrm_r_rm(eip + 1, &r, &rm);
    operand_read(&r);
    rm.val = r.val;
    operand_write(&rm);
    return len;
}
```

- 代码冗长难理解
- 容易引入错误

```
make_instr_func(mov_rm2r_v) {
    OPERAND rm, r;
    rm.data_size = data_size;
    r.data_size = data_size
    int len = 1;
    len += modrm_r_rm(eip + 1, &r, &rm);
    operand_read(&rm);
    r.val = rm.val;
    operand_write(&r);
    return len;
}
```

目录

- 程序执行的宏观过程与模拟
- 单条指令的解码与NEMU实现
  - 操作码的解码方式
  - 单条指令的实现方法
  - 利用框架代码中的宏来高效、简洁地实现多条指令

观察操作码编码规律

One-Byte Opcode Map

	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

操作相同，仅操作码类型长度不同

思路：将功能抽象出来

```
void execute_mov(OPERAND * opr_dest, OPERAND * opr_src)
{
    operand_read(opr_src);
    opr_dest.val = opr_src.val;
    operand_write(&opr_dest);
}

make_instr_func(mov_i2rm_v) {
    OPERAND opr_dest, opr_src;
    opr_dest.data_size = data_size;
    int len = 1;
    len += modrm_rm(eip + 1, &opr_dest);
    opr_src.type = OPR_IMM;
    opr_src.addr = eip + len;
    opr_src.data_size = data_size;
    execute_mov(&opr_dest, &opr_src);
    return len + data_size / 8;
}
```

操作数解码能不能抽象？

```
make_instr_func(mov_r2rm_v) {
    OPERAND opr_dest, opr_src;
    opr_dest.data_size = data_size;
    opr_src.data_size = data_size
    int len = 1;
    len += modrm_r_rm(eip + 1, &opr_src, &opr_dest);
    execute_mov(&opr_dest, &opr_src);
    return len;
}
```

```
make_instr_func(mov_r2rm_v) {
    OPERAND opr_dest, opr_src;
    opr_dest.data_size = data_size;
    opr_src.data_size = data_size
    int len = 1;
    len += modrm_r_rm(eip + 1, &opr_dest, &opr_src);
    execute_mov(&opr_dest, &opr_src);
    return len;
}
```

用于精简指令实现的宏

普通实现

会出现大量相似的重复代码

```
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;
}
```

精简实现

```
#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)
```

同样指令同样的逻辑

一行对应一条指令的实现

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

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

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

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

```
    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

```
    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

```
    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; \
            }
```

```
    // 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;
}
```

nemu/include/cpu/instr_helper.h

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
        }
```

nemu/include/cpu/instr_helper.h

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

```
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

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;
}
```

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

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

最终效果

```
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)
```

展开后约等于120行的代码

PA 2-1要做的任务：执行make run或make test_pa-2-1

需要修改Makefile来指定测试用例

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.

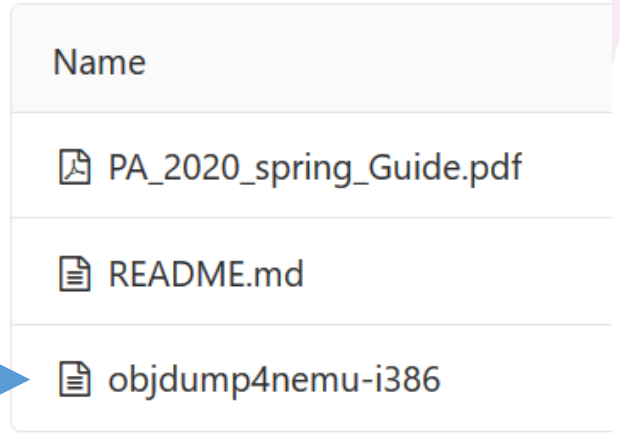
If it is the first case, see

for more details.

If it is the second case, remember:
* The machine is always right!
* Every line of untested code is always wrong!

根据eip，结合打印出来的内存内容定位需要实现的指令

配合使用我们定制的objdump工具



http://114.212.10.212/wl/pa2020_spring_guide

源码：https://gitee.com/wlicsnju/binutils4nemu

PA 2-1要做的任务：执行make run或make test_pa-2-1

1. **查i386手册得知这是一条什么指令**
   a) 先查appendix A得知指令的类型和格式
   b) 必要的话查section 17.2.1译码ModR/M和SIB字节
   c) 必要的话查section 17.2.2.11查看指令的具体含义和细节
2. **写该操作码对应的instr_func**
   a) 例如：make_instr_func(mov_i2rm_v)
3. **把这个函数在nemu/include/cpu/instr.h中声明一下**
4. **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**
5. **重复上述过程直至完成所有需要模拟的指令**

NEMU模拟指令解码和执行

- 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

实验目标

控制台

```
$ make clean
$ make test_pa-2-1
...
./nemu/nemu --autorun --testcase struct
NEMU load and execute img: ./testcase/bin/struct.img  elf: ./testcase/bin/struct
nemu: HIT GOOD TRAP at eip = 0x0003010c
NEMU2 terminated
./nemu/nemu --autorun --testcase string
NEMU load and execute img: ./testcase/bin/string.img  elf: ./testcase/bin/string
nemu: HIT GOOD TRAP at eip = 0x0003016a
NEMU2 terminated
./nemu/nemu --autorun --testcase hello-str
NEMU load and execute img: ./testcase/bin/hello-str.img  elf: ./testcase/bin/hello-str
nemu: HIT GOOD TRAP at eip = 0x00030105
NEMU2 terminated
./nemu/nemu --autorun --testcase test-float
NEMU load and execute img: ./testcase/bin/test-float.img  elf: ./testcase/bin/test-float
nemu: HIT BAD TRAP at eip = 0x000300c8
NEMU2 terminated
make[1]: Leaving directory '/home/icspa/teaching/temp_test/pa_code'
```

- PA 2-1提交截止时间待定
- 建议大家先写一些指令，发现在实现过程中不方便的地方，下一次课我们讲解框架代码中和精简指令实现的宏的有关内容

PA 2-1 结束

PA 2-1截止时间2022年4月14日24时