

## PA2 目录-1613415 潘巧巧

指令执行流程概述 (PA2.1、2.2)	2
CPU 执行指令的流程	2
关于 RTL	2
RTL 寄存器	2
RTL 指令	2
NEMU 的框架代码的指令执行流程	3
查阅 opcode_table 并进行宏的转换流程	4
译码函数和执行函数实现过程	5
关于 AM	10
Differential Testing	10
原理	10
实现	10
一键回归测试	11
输入输出 (PA2.3)	12
I/O 编制方式	12
IOE	12
串口	13
时钟	13
键盘	14
VGA	15
回答问题	17
遇到的坑	25
复习的知识和学到的经验	30

# 指令执行流程概述（PA2.1、2.2）

## CPU 执行指令的流程

①取值：从 eip 指针指向的内存取出一条指令进 CPU

②译码：查表获知该指令的操作数和操作码

③执行

（④更新 eip）

上述流程不断循环直至程序结束

## 关于 RTL

### RTL 寄存器

1、 8 个通用寄存器

```
enum { R_EAX, R_ECX, R_EDX, R_EBX, R_ESP, R_EBP, R_ESI, R_EDI }; //32位 8个
```

2、 源操作数 1、源操作数 2、目的操作数

```
#define id_src (&decoding.src) // 源操作数1  
#define id_src2 (&decoding.src2) // 源操作数2  
#define id_dest (&decoding.dest) // 目的操作数
```

3、 临时寄存器

4、 0 寄存器

```
rtlreg_t t0, t1, t2, t3; //临时寄存器  
const rtlreg_t tzero = 0; //0寄存器
```

### RTL 指令

1、 基本指令：只使用一条机器指令来实现相应的功能，同时也不需要临时寄存器，最基本的 x86 指令中的最基本的操作。

- ❖ 立即数读入 `rtl_li`
- ❖ 算术运算和逻辑运算, 包括寄存器-寄存器类型 `rtl_(add|sub|and|or|xor|shl|shr|sar|slt|sltu)` 和立即数-寄存器类型 `rtl_(add|sub|and|or|xor|shl|shr|sar|slt|sltu)i`
- ❖ 内存的访存 `rtl_lm` 和 `rtl_sm`
- ❖ 通用寄存器的访问 `rtl_lr_(b|w|l)` 和 `rtl_sr_(b|w|l)`

2、 伪指令：通过 RTL 基本指令或者已经实现的 RTL 伪指令来实现。

- ❖ 带宽度的通用寄存器访问 `rtl_lr` 和 `rtl_sr`
- ❖ EFLAGS 标志位的读写 `rtl_set_(CF|OF|ZF|SF|IF)` 和 `rtl_get_(CF|OF|ZF|SF|IF)`
- ❖ 其它常用功能, 如数据移动 `rtl_mv`, 符号扩展 `rtl_sext` 等

## NEMU 的框架代码的指令执行流程

- 1、程序总 `main()` 函数先调用 `init_monitor()` 经过一系列的初始化，再将返回值当做参数运行用户界面主循环函数 `ui_mainloop()`，`is_batch_mode` 是命令行参数控制的内容。在 `ui_mainloop()` 中进入 `cpu_exec()`，再来到 `exec_wrapper()`。以上过程在 PA1 实验报告中有详细说明。
- 2、首先看 `exec_wrapper()`，该函数主要处理内容是更新指令执行完成之后 `eip` 的更新和统计必要的执行信息。将该函数中的 `#ifdef` 合起来，看核心代码其实只有 3 行，易知整个指令执行流程的核心是在 `exec_real()` 函数

```
void exec_wrapper(bool print_flag) {
#ifdef DEBUG
#endif

    decoding.seq_eip = cpu.eip; // 1、将当期eip保存到全局译码信息成员中
    exec_real(&decoding.seq_eip); // 2、将其作为参数传入exec_real函数
                                   // exec_real返回的时候，decoding.seq_eip会指向下一条指令

#ifdef DEBUG
#endif

#ifdef DIFF_TEST
    uint32_t eip = cpu.eip;
#endif

    update_eip(); //更新eip

#ifdef DIFF_TEST
#endif
}
```

- 3、根据《NK 实验 PA 流程》，`exec_real()` 函数中有 3 步核心操作

- 1) 取指: `instr_fetch()` 函数

```
// 1、取值。得到指令，解释成opcode，记录到全局译码信息decoding中
// 2、在exec.c
static inline uint32_t instr_fetch(vaddr_t *eip, int len) {
    uint32_t instr = vaddr_read(*eip, len);
#ifdef DEBUG
    uint8_t *p_instr = (void *)&instr;
    int i;
    for (i = 0; i < len; i++) {
        decoding.p += sprintf(decoding.p, "%02x ", p_instr[i]);
    }
#endif
    (*eip) += len;
    return instr;
}
```

其中涉及到一个重要概念 `opcode`。

我们拥有一个数组 `opcode_table`，即译码查找表，该表通过操作码 `opcode` 来索引，共 512 个元素，初始时填写 `EMPTY`。每一个 `opcode` 对应相应指令的译码函数、执行函数、以及操作数宽度。这个数组的元素类型是 `struct opcode_entry`。

```
typedef struct {
    DHelper decode;
    EHelper execute;
    int width;
} opcode_entry;
```

D-decode-译码函数

E-execute-执行函数

可知这个 struct 保存解码和执行相关信息的存储

- 2) 设置宽度, `set_width()`。根据 opcode 查阅 `opcode_table`, 得到操作数的宽度信息, 并通过调用 `set_width()` 函数将其记录在全局译码信息 `decoding` 中。  
操作数宽度只有 1、2、4 三种, 如果操作数宽度为 1 则需要自己设置, 否则在该函数中会被自动设置。

```
// 2、查表得知宽度后调用该函数记录到全局译码变量decoding中
static inline void set_width(int width) {
    if (width == 0) {
        width = decoding.is_operand_size_16 ? 2 : 4;
    }
    decoding.src.width = decoding.dest.width = decoding.src2.width = width;
}
```

- 3) 译码和执行函数 `idex()`。该函数参数 2 位 `struct opcode_entry` 的指针 `*e`, 调用了 `struct` 中的两个成员——译码查找表中的相应的译码函数和执行函数。

```
// 3、译码和执行
static inline void idxex(vaddr_t *eip, opcode_entry *e) {
    /* eip is pointing to the byte next to opcode */
    if (e->decode)
        e->decode(eip);
    e->execute(eip);
}
```

## 查阅 opcode\_table 并进行宏的转换流程

——以 dummy.c 中自帶已完成的 mov 指令為例

- 1、在 dummy 的反汇编中的第一行代码查阅到 opcode 为 bd

dummy-x86-nemu.txt (~/.桌面/PA/ics2018/nexus-am/tests/cputest/build) - gedit

打开(O) 保存(S)

/home/pqq/桌面/PA/ics2018/nexus-am/tests/cputest/build/dummy-x86-nemu: 文件格式 elf32-i386

Disassembly of section .text:

00100000: <\_start>

100000: bd 00 00 00 00 mov \$0x0,%ebp

100005: dc 00 7c 00 00 mov \$0x7c00,%esp

10000a: e8 01 00 00 00 call 100010 <\_trm\_init>

10000f: 90 nop

- 2、于是程序查找到 opcode\_table 中的第 0xbd 个元素

```

* 0xb4 */ IDEXW(mov_I2r, mov, 1), IDEXW(mov_I2r, mov, 1), IDEXW(mov_I2r, mov, 1), IDEXW(mov_I2r, mov, 1),
* 0xb8 */ IDEX(mov_I2r, mov), IDEX(mov_I2r, mov), IDEX(mov_I2r, mov), IDEX(mov_I2r, mov),
* 0xbc */ IDEX(mov_I2r, mov), IDEX(mov_I2r, mov), IDEX(mov_I2r, mov), IDEX(mov_I2r, mov),
* 0xc0 */ IDEXW(gp2_Ib2E, gp2, 1), IDEX(gp2_Ib2E, gp2), EMPTY, EX(ret),
* 0xc4 */ EMPTY, EMPTY, IDEXW(mov_I2E, mov, 1), IDEX(mov_I2E, mov),

```

- ### 3、根据下列宏知

- 1) **IDEX(mov\_l2r,mov)**转换为 **IDEXW(mov\_l2r,mov,0)**
- 2) 再根据 concat 函数的拼接功能转换为 **IDEXW(decode mov\_l2r, exec mov, 0)**

```
#define IDEXW(id, ex, w)    {concat(decode_, id), concat(exec_, ex), w}
#define IDEX(id, ex)       IDEXW(id, ex, 0)
#define EXW(ex, w)         {NULL, concat(exec_, ex), w}
#define EX(ex)             EXW(ex, 0)
#define EMPTY              EX(inv)
```

- 3) 根据以下两个宏，替换为
- ```
IDEXW(  
void decode_mov_l2r (vaddr_t *eip),  
void exec_mov(vaddr_t *eip)  
)
```

```
#define make_DHelper(name) void concat(decode_, name) (vaddr_t *eip)
```

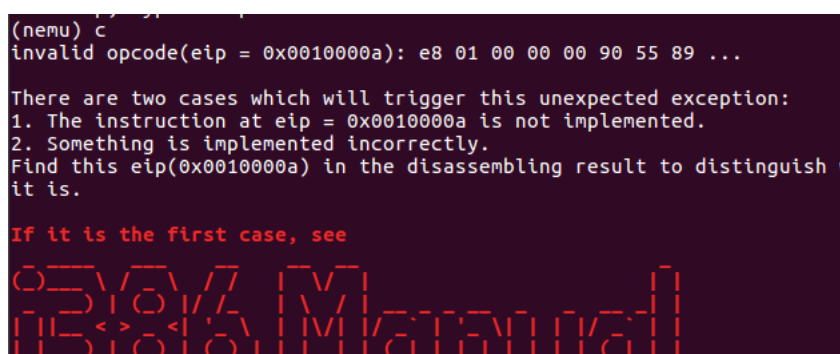
```
#define make_EHelper(name) void concat(exec_, name) (vaddr_t *eip)
```

根据以上分析知，在 idex 中的语句 `e->decode`，实际上经过宏的转换后是执行了函数 `decode_mov_l2r (vaddr_t *eip)`

## 译码函数和执行函数实现过程

——以 dummy.c 中需要完成的第一个指令 `call` 为例

- 1、在 `nexus-am/tests/cputest` 目录下键入 `make ARCH=x86-nemu ALL=dummy run` 编译 dummy 程序，并启动 NEMU 运行它。
- 2、根据运行后的报错信息和反汇编代码或 [i386 手册](#) 查找当前缺失的指令集。例如刚开始 PA2 时运行 dummy.c 程序出现如下报错。



(nemu) c  
invalid opcode(eip = 0x0010000a): e8 01 00 00 90 55 89 ...  
  
There are two cases which will trigger this unexpected exception:  
1. The instruction at eip = 0x0010000a is not implemented.  
2. Something is implemented incorrectly.  
Find this eip(0x0010000a) in the disassembling result to distinguish it is.  
  
If it is the first case, see  
[O32/M32 Manual](#)

- ① 根据反汇编 `dummy-x86-nemu-txt` 中查阅“`e8 01 00 00` ……”的代码可知这是一个 `call` 指令

```
00100000 <_start>:  
100000: bd 00 00 00 00      mov     $0x0,%ebp  
100005: bc 00 7c 00 00      mov     $0x7c00,%esp  
10000a: e8 01 00 00 00      call    100010 <_trm_init>  
10000f: 90                  nop
```

- ② 根据 [i386 手册](#) 的附录 A (opcode map) 查阅第 e 行第 8 列可知这是一个 `call` 指令

One-Byte Opcode Map

|   | 0                                         | 1       | 2      | 3        | 4      | 5       | 6                                                         | 7            | 8                                         | 9       | A                                           | B       | C      | D       | E       | F                  |
|---|-------------------------------------------|---------|--------|----------|--------|---------|-----------------------------------------------------------|--------------|-------------------------------------------|---------|---------------------------------------------|---------|--------|---------|---------|--------------------|
| 0 | ADD                                       |         |        |          |        |         | PUSH                                                      | POP          |                                           | OR      |                                             |         |        |         |         | PUSH 2-byte escape |
|   | Eb, Gb                                    | Ev, Gv  | Gb, Eb | Gv, Ev   | Al, Ib | eAX, Iv | ES                                                        | ES           | Eb, Gb                                    | Ev, Gv  | Gb, Eb                                      | Gv, Ev  | Al, Ib | eAX, Iv | CS      |                    |
| 1 | ADC                                       |         |        |          |        |         | PUSH                                                      | POP          |                                           | SBB     |                                             |         |        |         |         | PUSH POP           |
|   | Eb, Gb                                    | Ev, Gv  | Gb, Eb | Gv, Ev   | Al, Ib | eAX, Iv | SS                                                        | SS           | Eb, Gb                                    | Ev, Gv  | Gb, Eb                                      | Gv, Ev  | Al, Ib | eAX, Iv | DS      | DS                 |
| 2 | AND                                       |         |        |          |        |         | SEG                                                       | DAA          |                                           | SUB     |                                             |         |        |         |         | SEG DAS            |
|   | Eb, Gb                                    | Ev, Gv  | Gb, Eb | Gv, Ev   | Al, Ib | eAX, Iv | ES                                                        |              | Eb, Gb                                    | Ev, Gv  | Gb, Eb                                      | Gv, Ev  | Al, Ib | eAX, Iv | CS      |                    |
| 3 | XOR                                       |         |        |          |        |         | SEG                                                       | AAA          |                                           | CMP     |                                             |         |        |         |         | SEG AAS            |
|   | Eb, Gb                                    | Ev, Gv  | Gb, Eb | Gv, Ev   | Al, Ib | eAX, Iv | ES                                                        |              | Eb, Gb                                    | Ev, Gv  | Gb, Eb                                      | Gv, Ev  | Al, Ib | eAX, Iv | CS      |                    |
| 4 | INC general register                      |         |        |          |        |         | DEC general register                                      |              |                                           |         |                                             |         |        |         |         |                    |
|   | eAX                                       | eCX     | eDX    | eBX      | eSP    | eBP     | eSI                                                       | eDI          | eAX                                       | eCX     | eDX                                         | eBX     | eSP    | eBP     | eSI     | eDI                |
| 5 | PUSH general register                     |         |        |          |        |         | POP into general register                                 |              |                                           |         |                                             |         |        |         |         |                    |
|   | eAX                                       | eCX     | eDX    | eBX      | eSP    | eBP     | eSI                                                       | eDI          | eAX                                       | eCX     | eDX                                         | eBX     | eSP    | eBP     | eSI     | eDI                |
| 6 | PUSHA                                     | POPA    | BOUND  | ARPL     | SEG    | SEG     | Operand Size                                              | Address Size | PUSH                                      | IMUL    | PUSH                                        | IMUL    | INSB   | INSW/D  | OUTSB   | OUTSW/D            |
|   | Gv, Ma                                    | Ew, Rw  | EW, RW | FS       | GS     |         |                                                           |              | Id                                        | GvEvIv  | Id                                          | GvEvIv  | Yb, DX | Yb, DX  | Dx, Xb  | Dx, Xv             |
| 7 | Short displacement jump of condition (Jb) |         |        |          |        |         |                                                           |              | Short displacement jump on condition (Jb) |         |                                             |         |        |         |         |                    |
|   | JO                                        | JNO     | JB     | JNB      | JZ     | JNZ     | JBE                                                       | JNBE         | JS                                        | JNS     | JP                                          | JNP     | JL     | JNL     | JLE     | JNLE               |
| 8 | Immediate Grpl                            |         |        | Grpl     |        |         | TEST                                                      |              |                                           | XCHG    |                                             |         | MOV    |         |         | POP                |
|   | Eb, Ib                                    | Ev, Iv  |        | Eb, Ib   | Ev, Iv |         | Eb, Gb                                                    | Ev, Gv       |                                           | Eb, Gb  | Ev, Gv                                      | Gb, Eb  | Gv, Ev | Ew, Sw  | Gv, M   | Sw, Ew             |
| 9 | NOP                                       |         |        |          |        |         | XCHG word or double-word register with eAX                |              |                                           |         |                                             |         |        |         |         |                    |
|   |                                           | eCX     | eDX    | eBX      | eSP    | eBP     | eSI                                                       | eDI          | CSW                                       | CMD     | CALL                                        | WAIT    | PUSHF  | POFF    | SAHF    | LAHF               |
|   |                                           |         |        |          |        |         |                                                           |              | Ap                                        |         |                                             |         | Fv     | Fv      |         |                    |
| A | MOV                                       |         |        |          |        |         | MOVSB                                                     | MOVSW/D      | CMPSB                                     | CMPSW/D | TEST                                        |         | STOSB  | STOSW/D | LODSB   | LODSW/D            |
|   | AL, Ob                                    | eAX, Ov | Ob, AL | Ov, eAX  | Xb, Yb | Xv, Yv  | Xb, Yb                                                    | Xv, Yv       |                                           |         | AL, Id                                      | eAX, Iv | Yb, AL | Iv, eAX | AL, Xb  | eAX, Xv            |
|   |                                           |         |        |          |        |         |                                                           |              |                                           |         |                                             |         |        |         | AL, Xb  | eAX, Xv            |
| B | MOV immediate byte into byte register     |         |        |          |        |         | MOV immediate word or double into word or double register |              |                                           |         |                                             |         |        |         |         |                    |
|   | AL                                        | CL      | DL     | BL       | AH     | CH      | DH                                                        | BH           | eAX                                       | eCX     | eDX                                         | eBX     | eSP    | eBP     | eSI     | eDI                |
| C | Shift Grp2                                |         |        | RET near | LES    | LDS     | MOV                                                       |              | ENTER                                     | LEAVE   | RET far                                     |         | INT    | INT     | INT     | IRET               |
|   | Eb, Ib                                    | Ev, Iv  | Iw     |          | Gv, Mp | Gv, Mp  | Eb, Ib                                                    | Ev, Iv       | Iw, Ib                                    |         | Iw                                          |         | 3      | Ib      |         |                    |
| D | Shift Grp2                                |         |        |          |        |         | AMM                                                       | AAD          |                                           | XLAT    | ESC (Escape to coprocessor instruction set) |         |        |         |         |                    |
|   | Eb, l                                     | Ev, l   | Eb, Cl | Ev, Cl   |        |         |                                                           |              |                                           |         |                                             |         |        |         |         |                    |
| E | LOOPE                                     | LOOPE   | LOOP   | JCXZ     | IN     |         | OUT                                                       |              | CALL                                      | JNP     |                                             | IN      |        | OUT     |         |                    |
|   | Jb                                        | Jb      | Jb     |          | AL, Ib | eAX, Ib | Id, AL                                                    | Id, eAX      | Av                                        | Jv      | Ap                                          | Jb      | AL, DX | eAX, DX | DX, AL  | DX, eAX            |
| F | LOCK                                      |         | REPNE  | REPE     | HLT    | CMC     | Unary Grp3                                                |              | CMC                                       | STC     | CLI                                         | STI     | CLD    | STD     | INC/DEC | Indirect           |
|   |                                           |         |        |          |        |         | Eb                                                        | Ev           |                                           |         |                                             |         |        |         | Grp4    | Grp5               |

### 3、在填写 opcode\_table 数组

- ① 找到 opcode\_table 数组的第 e8 个数据，目前是 EMPTY

```
/* 0xC4 */ EMPTY,
/* 0xE8 */ EMPTY,
```

- ② 我们需要根据需求在这里填入一个宏——IDEX(id,ex)，现在需要确定 id 和 ex 具体需要填入什么内容。
- ③ 关于 id，在 decode.c 中，框架代码已经帮我们实现好了各种参数下的相关 make\_DopHelper 函数，现在我们需要做的就是确定下我们的 id 填写什么。
- ④ 虽然此时我们仍然不知道 IDEX(id,ex)中的 id 需要填什么，此时需要模仿已知的 mov 函数。mov 函数在 opcode\_table 中填写为 IDEX(mov\_I2r,mov)。我们可以在 decode.h 中看到许多 make\_DHelper 函数的定义，他们的参数就是可以填写到这里的 id 的选项。

```

make_DHelper(I2E);
make_DHelper(I2a);
make_DHelper(I2r);
make_DHelper(SI2E);
make_DHelper(SI_E2G);
make_DHelper(I_E2G);
make_DHelper(I_G2E);
make_DHelper(I);
make_DHelper(r);
make_DHelper(E);
make_DHelper(gp7_E);
make_DHelper(test_I);
make_DHelper(SI);
make_DHelper(G2E);
make_DHelper(E2G);

make_DHelper(mov_I2r);
make_DHelper(mov_I2E);
make_DHelper(mov_G2E);
make_DHelper(mov_E2G);
make_DHelper(lea_M2G);

make_DHelper(gp2_1_E);
make_DHelper(gp2_cI2E);
make_DHelper(gp2_Ib2E);

make_DHelper(02a);
make_DHelper(a20);

make_DHelper(J);

make_DHelper(push_SI);

make_DHelper(in_I2a);
make_DHelper(in_dx2a);
make_DHelper(out_a2I);
make_DHelper(out_a2dx);

```

查阅 i386 手册附录 A，可以查到各个字母的意思。

#### Codes for Addressing Method

- A Direct address; the instruction has no modR/M byte; the address of the operand is encoded in the instruction; no base register, index register, or scaling factor can be applied; e.g., far JMP (EA).
- C The reg field of the modR/M byte selects a control register; e.g., MOV (0F20, 0F22).
- D The reg field of the modR/M byte selects a debug register; e.g., MOV (0F21, 0F23).
- E A modR/M byte follows the opcode and specifies the operand. The operand is either a general register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.
- F Flags Register.
- G The reg field of the modR/M byte selects a general register; e.g., ADD (00).
- I Immediate data. The value of the operand is encoded in subsequent bytes of the instruction.
- J The instruction contains a relative offset to be added to the instruction pointer register; e.g., JMP short, LOOP.
- M The modR/M byte may refer only to memory; e.g., BOUND, LES, LDS, LSS, LFS, LGS.
- O The instruction has no modR/M byte; the offset of the operand is coded as a word or double word (depending on address size attribute) in the instruction. No base register, index register, or scaling factor can be applied; e.g., MOV (A0-A3).

- ⑤ 根据实验手册提示，当前我们只需要实现 CALL rel32 的形式，在 i386 手册中找到 call produce 页。



## CALL — Call Procedure

| Opcode | Instruction   | Clocks       | Description                                          |
|--------|---------------|--------------|------------------------------------------------------|
| E8 cw  | CALL rel16    | 7+m          | Call near, displacement relative to next instruction |
| FF /2  | CALL r/m16    | 7+m/10+m     | Call near, register indirect/memory indirect         |
| 9A cd  | CALL ptr16:16 | 17+m,pm=34+m | Call intersegment, to full pointer given             |
| 9A cd  | CALL ptr16:16 | pm=52+m      | Call gate, same privilege                            |
| 9A cd  | CALL ptr16:16 | pm=86+m      | Call gate, more privilege, no parameters             |
| 9A cd  | CALL ptr16:16 | pm=94+4x+m   | Call gate, more privilege, x parameters              |
| 9A cd  | CALL ptr16:16 | ts           | Call to task                                         |
| FF /3  | CALL m16:16   | 22+m,pm=38+m | Call intersegment, address at r/m dword              |
| FF /3  | CALL m16:16   | pm=56+m      | Call gate, same privilege                            |
| FF /3  | CALL m16:16   | pm=90+m      | Call gate, more privilege, no parameters             |
| FF /3  | CALL m16:16   | pm=98+4x+m   | Call gate, more privilege, x parameters              |
| FF /3  | CALL m16:16   | 5 + ts       | Call to task                                         |
| E8 cd  | CALL rel32    | 7+m          | Call near, displacement relative to next instruction |
| FF /2  | CALL r/m32    | 7+m/10+m     | Call near, indirect                                  |
| 9A cp  | CALL ptr16:32 | 17+m,pm=34+m | Call intersegment, to pointer given                  |
| 9A cp  | CALL ptr16:32 | pm=52+m      | Call gate, same privilege                            |
| 9A cp  | CALL ptr16:32 | pm=86+m      | Call gate, more privilege, no parameters             |
| 9A cp  | CALL ptr32:32 | pm=94+4x+m   | Call gate, more privilege, x parameters              |
| 9A cp  | CALL ptr16:32 | ts           | Call to task                                         |
| FF /3  | CALL m16:32   | 22+m,pm=38+m | Call intersegment, address at r/m dword              |
| FF /3  | CALL m16:32   | pm=56+m      | Call gate, same privilege                            |
| FF /3  | CALL m16:32   | pm=90+m      | Call gate, more privilege, no parameters             |
| FF /3  | CALL m16:32   | pm=98+4x+m   | Call gate, more privilege, x parameters              |
| FF /3  | CALL m16:32   | 5 + ts       | Call to task                                         |

根据 i386 手册，知道该指令的作用为“Call near, displacement relative to next instruction”。

E8 后面的 cd 的意思在 i386 手册 17.2.2

### 17.2.2.1 Opcode

The "Opcode" column gives the complete object code produced for each form of the instruction. When possible, the codes are given as hexadecimal bytes, in the same order in which they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

**/digit**: (digit is between 0 and 7) indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.

**/r**: indicates that the ModR/M byte of the instruction contains both a register operand and an r/m operand.

**cb, cw, cd, cp**: a 1-byte (cb), 2-byte (cw), 4-byte (cd) or 6-byte (cp) value following the opcode that is used to specify a code offset and possibly a new value for the code segment register.

**ib, iw, id**: a 1-byte (ib), 2-byte (iw), or 4-byte (id) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words and doublewords are given with the low-order byte first.

**+rb, +rw, +rd**: a register code, from 0 through 7, added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte. The codes are—

于是我们现在可以大致推断出要填入的参数为 J。

- ⑥ 现在填写第二个参数 ex。在 exec 文件夹下有许多待 TODO() 的 make\_EHelper() 函数，于是我们选择一个相关函数，即 call。故 ex 的参数填 call。
- ⑦ 综上，我们在目标 EMPTY 处填入 IDEX(J,call)。

```
/* 0xe0 */ EMPTY, EMPTY,
/* 0xe4 */ EMPTY, EMPTY,
/* 0xe8 */ IDEX(J, call),
/* 0xec */ IDEX(J, call),
/* 0xf0 */ EMPTY, EMPTY,
```

- ⑧ 运行后发现 make\_EHelper(call)未定义，需要到 all-instr.h 文件中添加函数定义。



```

dummy
pq@pqq-virtual-machine:~/桌面/PA/ics2018/nexus-am/tests/cputest$ make ARCH=x86-
nemu ALL=dummy run
Building dummy [x86-nemu]
Building am [x86-nemu]
make[2]: *** 没有指明目标并且找不到 makefile。 停止。
+ CC src/cpu/exec/exec.c
src/cpu/exec/exec.c:12:42: error: 'exec_call' undeclared here (not in a function
)
#define EXW(ex, w)          {NULL, concat(exec_, ex), w}

```

```

make_EHelper(mov);
make_EHelper(operand_size);
make_EHelper(inv);
make_EHelper(nemu_trap);
//TODO
make_EHelper(call);
make_EHelper(push);
make_EHelper(pop);

```

- ⑨ 再次运行 dummy 发现遇到 TODO(), 找到目标文件 28 行发现 call 指令的执行函数没有实现

```

am/tests/cputest/build/dummy x86-nemu
Welcome to NEMU!
[src/monitor/monitor.c,32,welcome] Build time: 21:11:54, Jun  4 2019
For help, type "help"
(nemu) c
please implement me
nemu: src/cpu/exec/control.c:28: exec_call: Assertion `0' failed.
Makefile:46: recipe for target 'run' failed

```

- ⑩ 根据 i386 手册实现 call 指令。有时还需要补充很多 rtl 指令。

```

make_EHelper(call) {
    // the target address is calculated at the decode stage
    //TODO();
    //decoding.is_jump=1;
    //decoding.jump_eip = id_dest->val+*eip;

    rtl_push(&decoding.seq_eip);
    decoding.is_jump=1;

    print_asm("call %x", decoding.jump_eip);
}

```

```

static inline void rtl_push(const rtlreg_t* src1) {
    cpu.esp = cpu.esp - 4;
    vaddr_write(cpu.esp,4,*src1);
    //TODO();
}

```

- ⑪ 再次运行 dummy.c, 发现 call 指令成功执行, 此时卡在下一个未实现的指令上。于是重复本节, 实现所有指令。直到成功运行所有文件。

```

dummy
pq@pqq-virtual-machine:~/桌面/PA/ics2018/nexus-am/tests/cputest$ make ARCH=x86-
nemu ALL=dummy run
Building dummy [x86-nemu]
Building am [x86-nemu]
make[2]: *** 没有指明目标并且找不到 makefile。 停止。
+ CC src/cpu/exec/control.c
+ LD build/nemu
[src/monitor/diff-test/diff-test.c,96,init_difftest] Connect t
y
[src/monitor/monitor.c,67,load_img] The image is /home/pq/桌面
-am/tests/cputest/build/dummy-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,32,welcome] Build time: 21:11:54, Jun
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100052

```

## 关于 AM

- 1、 功能：把计算机相关的需求抽象成统一的 API 提供给程序。为真机提供抽象。
- 2、 AM 组成

AM = TRM + IOE + ASYE + PTE + MPE

- TRM(Turing Machine) - 图灵机, 为计算机提供基本的计算能力
- IOE(I/O Extension) - 输入输出扩展, 为计算机提供输出输入的能力
- ASYE(Asynchronous Extension) - 异步处理扩展, 为计算机提供处理中断异常的能力
- PTE(Protection Extension) - 保护扩展, 为计算机提供存储保护的能力
- MPE(Multi-Processor Extension) - 多处理器扩展, 为计算机提供多处理器通信的能力 (MPE 超出了 ICS 课程的范围, 在 PA 中不会涉及)

- 3、 PA2.3 要在 AM 中实现 IOE 的抽象

## Differential Testing

### 原理

- 1、 让在 NEMU 中执行的每条指令也在真机 (QEMU) 中执行一次, 然后对比 NEMU 和真机的状态, 如果 NEMU 和真机的状态不一致, 我们就捕捉到 error 了。让 NEMU 和 QEMU 逐条指令地执行同一个客户程序。双方每执行完一条指令, 就检查各自的寄存器和内存的状态, 如果发现状态不一致, 就马上报告错误, 停止客户程序的执行。
- 2、 测试函数会在 exec\_wrapper() 的最后被调用, 在 NEMU 中执行完一条指令后, 就在 difftest\_step() 中让 QEMU 执行相同的指令, 然后读出 QEMU 中的寄存器。

### 实现

- 1、 在 nemu/include/common.h 中定义宏 DIFF\_TEST

```
#define DEBUG
#define DIFF_TEST
/* You will define this
```

- 2、 实现 DIFF\_TEST 函数。即添加相应的代码, 把 NEMU 的 8 个通用寄存器+eip 与从 QEMU 中读出的寄存器的值进行比较。如果发现值不一样, 就输出相应的提示信息, 并将 diff 标志设置为 true。

```

regcpy_from_nemu(mine);

if(r.eax != mine.eax) {
    diff = true;
    printf("qemus eax=0x%08x, mine eax=0x%08x, eip:0x%08x\n", r.eax, mine.eax, mine.eip);
}
if(r.ebx != mine.ebx) {
    diff = true;
    printf("qemus ebx=0x%08x, mine ebx=0x%08x, eip:0x%08x\n", r.ebx, mine.ebx, mine.eip);
}
if(r.ecx != mine.ecx) {
    diff = true;
    printf("qemus ecx=0x%08x, mine ecx=0x%08x, eip:0x%08x\n", r.ecx, mine.ecx, mine.eip);
}
if(r.edx != mine.edx) {
    diff = true;
    printf("qemus edx=0x%08x, mine edx=0x%08x, eip:0x%08x\n", r.edx, mine.edx, mine.eip);
}
if(r.esp != mine.esp) {
    diff = true;
    printf("qemus esp=0x%08x, mine esp=0x%08x, eip:0x%08x\n", r.esp, mine.esp, mine.eip);
}
if(r.ebp != mine.ebp) {
    diff = true;
    printf("qemus ebp=0x%08x, mine ebp=0x%08x, eip:0x%08x\n", r.ebp, mine.ebp, mine.eip);
}
if(r.esi != mine.esi) {
    diff = true;
    printf("qemus esi=0x%08x, mine esi=0x%08x, eip:0x%08x\n", r.esi, mine.esi, mine.eip);
}
if(r.edi != mine.edi) {
    diff = true;
    printf("qemus edi=0x%08x, mine edi=0x%08x, eip:0x%08x\n", r.edi, mine.edi, mine.eip);
}
if(r.eip != mine.eip) {
    diff = true;
    printf("qemus eip=0x%08x, mine eip=0x%08x, eip:0x%08x\n", r.eip, mine.eip, mine.eip);
}
}
if (diff) {
    nemu state = NEMU END;
}

```

## 一键回归测试

即在 nemu/目录下运行 `bash runall.sh` 来批量运行 `nexus-am/tests/cputest/` 中的所有测试，并报告每个测试用例的运行结果。

```

pq@pqq-virtual-machine:~/桌面/PA/ics2018/nemu$ bash runall.sh
NEMU compile OK
compiling testcases...
testcases compile OK
[ add-longlong] PASS!
[ add] PASS!
[ bit] PASS!
[ bubble-sort] PASS!
[ dummy] PASS!
[ fact] PASS!
[ fib] PASS!
[ goldbach] PASS!
[ hello-str] PASS!
[ if-else] PASS!
[ leap-year] PASS!
[ load-store] PASS!
[ matrix-mul] PASS!
[ max] PASS!
[ min3] PASS!
[ mov-c] PASS!
[ movsx] PASS!
[ mul-longlong] PASS!
[ pascal] PASS!
[ prime] PASS!
[ quick-sort] PASS!
[ recursion] PASS!
[ select-sort] PASS!
[ shift] PASS!
[ shuixianhua] PASS!
[ string] PASS!
[ sub-longlong] PASS!
[ sum] PASS!
[ switch] PASS!
[ to-lower-case] PASS!
[ unalign] PASS!
[ wanshu] PASS!

```

# 输入输出 (PA2.3)

## I/O 编制方式

- 1、作用：帮助 CPU 区分不同的设备
- 2、**端口映射 IO**: CPU 使用专门的 I/O 指令对设备进行访问, 并把设备的地址称作端口号。有了端口号以后, 在 I/O 指令中给出端口号, 就知道要访问哪一个设备的哪一个寄存器了。x86 提供了 in 和 out 指令用于访问设备, 其中 in 指令用于将设备寄存器中的数据传输到 CPU 寄存器中, out 指令用于将 CPU 寄存器中的数据传送到设备寄存器中。
- 3、**内存映射 IO**: 通过不同的物理内存地址给设备编址的。这种编址方式将一部分物理内存“重定向”到 I/O 地址空间中, CPU 尝试访问这部分物理内存的时候, 实际上最终是访问了相应的 I/O 设备。物理内存的地址空间和 CPU 的位宽都会不断增长, 内存映射 I/O 从来不需要担心 I/O 地址空间耗尽的问题。

## IOE

- 1、定义宏 HAS\_IOE。定义后, init\_device()函数会对设备进行初始化。重新编译后, 你会看到运行 NEMU 时会弹出一个新窗口

```
/* You will define t
#define HAS_IOE
```

- 2、在 AM 中实现相应的 API 为程序提供 IOE 的抽象
  - 1) **串口**: 是最简单的输出设备, PA 实验中只保留了数据寄存器和状态寄存器。由于 NEMU 串行模拟计算机系统的工, 串口的状态寄存器可以一直处于空闲状态, 每当 CPU 往数据寄存器中写入数据时, 串口会将数据传送到主机的标准输出。
  - 2) **时钟**: 有了时钟, 程序才可以提供时间相关的体验, timer.c 模拟了 i8253 计时器的功能。PA 中保留了“发起时钟中断”的功能。同时添加了一个自定义的 RTC(Real Time Clock), 初始化时将会注册 0x48 处的端口作为 RTC 寄存器, CPU 可以通过 I/O 指令访问这一寄存器, 获得当前时间(单位是 ms)。
  - 3) **键盘**: 键盘是最基本的输入设备。一般键盘的工作方式如下: 当按下一个键的时候, 键盘将会发送该键的通码(makecode); 当释放一个键的时候, 键盘将会发送该键的断码(break code)。keyboard.c 模拟 i8042 通用设备接口芯片的功能。其大部分功能也被简化, 只保留了键盘接口。i8042 初始化时会注册 0x60 处的端口作为数据寄存器, 注册 0x64 处的端口作为状态寄存器。每当用户敲下/释放按键时, 将会把相应的键盘码放入数据寄存器, 同时把状态寄存器的标志设置为 1, 表示有按键事件发生。CPU 可以通过端口 I/O 访问这些寄存器, 获得键盘码。在 AM 中约定通码的值为断码+0x8000。
  - 4) **VGA**: VGA 可以用于显示颜色像素, 是最常用的输出设备。vga.c 模拟了 VGA 的功能。VGA 初始化时注册了从 0x40000 开始的一段用于映射到 video memory 的物理内存。在 NEMU 中, video memory 是唯一使用内存映射 I/O 方式访问的 I/O 空间。R(red),G(green),B(blue),A(alpha)各占 8 bit。

串口

- 1) 利用 `pio_read` 和 `pio_write` 实现 `in` 和 `out` 指令。

```
make_EHelper(in) {
    //TODO();
    t0 = pio_read(id_src->val, id_src->width);
    operand_write(id_dest, &t0);
    print_asm_template2(in);

#ifdef DIFF_TEST
    diff_test_skip_qemu(); //跳过检查
#endif
}

make_EHelper(out) {
    //TODO();
    pio_write(id_dest->val, id_src->width, id_src->val);
    // print_asm_template2(out);

#ifdef DIFF_TEST
    diff_test_skip_qemu();
#endif
}
```

- 2) 定义宏 HAS\_SERIAL。

```
// Define this macro
#define HAS_SERIAL
```

- ### 3) 运行 hello.c 函数

```
[src/monitor/diff-test/diff-test.c,96,init_difftest] Connect to QEMU success  
y  
[src/monitor/monitor.c,67,load_img] The image is /home/pqq/桌面/PA/ics2018  
-am/apps/hello/build/hello-x86-nemu.bin  
Welcome to NEMU!  
[src/monitor/monitor.c,32,welcome] Build time: 21:11:54, Jun  4 2019  
For help, type "help"  
(nemu) c  
Hello World!  
Hello World!  
Hello World!  
Hello World!  
Hello World!  
Hello World!  
Hello World!  
Hello World!  
Hello World!  
Hello World!  
nemu: HIT GOOD TRAP at eip = 0x0010006e  
(nemu)
```

## 时钟

- ### 1) 实现\_uptime()

```
#define RTC_PORT 0x48 // Note that this is not standard
static unsigned long boot_time;

void _ioe_init() {
    boot_time = inl(RTC_PORT);
}

unsigned long _uptime() {
    //return 0;
    unsigned answer = inl(RTC_PORT) - boot_time;
    return answer;
}
```

## 2) 运行 timetest.c

```
pqq@pqq-virtual-machine: ~/桌面/PA/ics2018/nexus-am/tests/timetest
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
make[1]: Entering directory '/home/pqq/桌面/PA/ics2018/nemu'
./build/nemu -l /home/pqq/桌面/PA/ics2018/nexus-am/tests/timetest/build/n
.txt /home/pqq/桌面/PA/ics2018/nexus-am/tests/timetest/build/timetest-x86
in
[src/monitor/diff-test/diff-test.c,96,init_difftest] Connect to QEMU succ
y
[src/monitor/monitor.c,67,load_img] The image is /home/pqq/桌面/PA/ics201
-am/tests/timetest/build/timetest-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,32,welcome] Build time: 21:11:54, Jun  4 2019
For help, type "help"
(nemu) c
0.1 second.
2 seconds.
3 seconds.
4 seconds.
5 seconds.
6 seconds.
7 seconds.
8 seconds.
9 seconds.
10 seconds.
11 seconds.
12 seconds.
```

## 键盘

- 1、实现\_read\_key(), 根据手册提示：每当用户敲下/释放按键时，将会把相应的键盘码放入数据寄存器，同时把状态寄存器的标志设置为 1，表示有按键事件发生。

```
#define I8042_DATA_PORT 0x60 //数据寄存器
#define I8042_STATUS_PORT 0x64 //状态寄存器
int _read_key() {
    // return KEY_NONE;
    uint8_t if_happen = inb(I8042_STATUS_PORT);
    if (if_happen==1)
        return inl(I8042_DATA_PORT);
    else
        return _KEY_NONE;
}
```

- 2、运行 key\_test()。在 NEMU 中键入，小黑框中显示键入的信息

```
pqq@pqq-virtual-machine: ~/桌面/PA/ics2018/nexus-
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
.txt /home/pqq/桌面/PA/ics2018/nexus-am/tests/keytes
[src/monitor/diff-test/diff-test.c,96,init_difftest
y
[src/monitor/monitor.c,67,load_img] The image is /h
-am/tests/keytest/build/keytest-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,32,welcome] Build time: 21:1
For help, type "help"
(nemu) c
Get key: 36 I down
Get key: 36 I up
Get key: 51 L down
Get key: 51 L up
Get key: 37 O down
Get key: 37 O up
Get key: 59 V down
Get key: 59 V up
Get key: 31 E down
Get key: 31 E up
Get key: 38 P down
Get key: 38 P up
Get key: 43 A down
Get key: 43 A up
```

## VGA

- 1、在 `paddr_read()`和 `paddr_write()`中加入对内存映射 I/O 的判断。通过 `is_mmio()`函数判断一个物理地址是否被映射到 I/O 空间。如果是，`is_mmio()`会返回映射号，否则返回-1。内存映射 I/O 的访问需要调用 `mmio_read()`或 `mmio_write()`，调用时需要提供映射号。如果不是内存映射 I/O 的访问就访问 `pmem`。

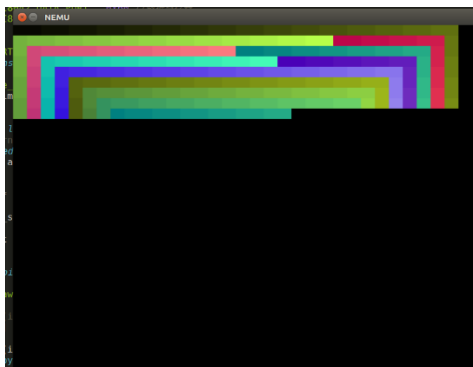
```
uint32_t paddr_read(paddr_t addr, int len) {
    //return pmem_rw(addr, uint32_t) & (~0u >> ((4 - len) << 3));
    int map_NO = is_mmio(addr);
    if (map_NO != -1)
        return mmio_read(addr, len, map_NO);
    else
        return pmem_rw(addr, uint32_t) & (~0u >> ((4 - len) << 3));
}

void paddr_write(paddr_t addr, int len, uint32_t data) {
    //memcpy(guest_to_host(addr), &data, len);
    int map_NO = is_mmio(addr);
    if (map_NO != -1)
        mmio_write(addr, len, data, map_NO);
    else
        memcpy(guest_to_host(addr), &data, len);
}
```

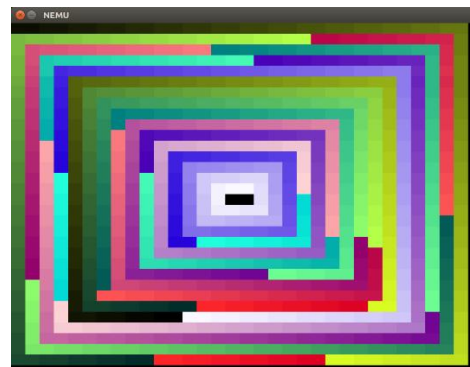
- 2、实现正确的 `_draw_rect()`

```
void _draw_rect(const uint32_t *pixels, int x, int y, int w, int h) {
    for (int i = 0; i < h; i++)
        memcpy(fb + (y + i) * _screen.width + x, pixels + i * w, w * 4);
}
```

- 3、运行 `videotest`

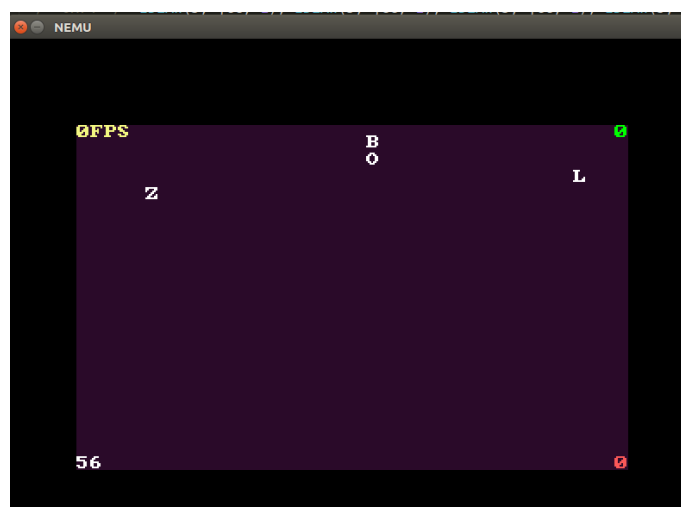


(颜色加载中)



(不断变化的画面)

- 4、运行打字小游戏



(加载无敌慢的小游戏，慢到难以输入)





# 回答问题

1、

## 立即数背后的故事

在 `decode_op_l()` 函数中通过 `instr_fetch()` 函数获得指令中的立即数。别看这里就这么一行代码，其实背后隐藏着针对字节序的慎重考虑。我们知道 x86 是小端机，当你使用高级语言或者汇编语言写了一个 32 位常数 0x1234 的时候，在生成的二进制代码中，这个常数对应的字节序列如下（假设这个常数在内存中的起始地址是 x）：

| x  | x+1 | x+2 | x+3 |
|----|-----|-----|-----|
| 34 | 12  | 00  | 00  |

而大多数 PC 机都是小端架构（我们相信没有同学会使用 IBM 大型机来做 PA），当 NEMU 运行的时候，`op_src->imm=instr_fetch(eip,4)`，这行代码会将 34 12 00 00 这个字节序列原封不动地从内存读入 `imm` 变量中，主机的 CPU 会按照小端方式来解释这一字节序列，于是会得到 0x1234，符合我们的预期结果。

Motorola 68k 系列的处理器都是大端架构的。现在问题来了，考虑以下两种情况：

- ❖ 假设我们需要将 NEMU 运行在 Motorola 68k 的机器上（把 NEMU 的源代码编译成 Motorola 68k 的机器码）
- ❖ 假设我们需要编写一个新的模拟器 NEMU-Motorola-68k，模拟器本身运行在 x86 架构中，但它模拟的是 Motorola 68k 程序的执行

在这两种情况下，你需要注意些什么问题？为什么会产生这些问题？怎么解决它们？事实上不仅仅是立即数的访问，长度大于 1 字节的内存访问都需要考虑类似的问题。我们在这里把问题统一抛出来，以后就不再单独讨论了。

2、

## 堆和栈在哪里？

我们知道代码和数据都在可执行文件里面，但却没有提到堆（heap）和栈（stack）。为什么堆和栈的内容没有放入可执行文件里面？那程序运行时刻用到的堆和栈又是怎么来的？AM 的代码是否能给你带来一些启发？

可执行文件中主要包括：

.text：存可执行文件指令

.rwdata：可读写全局初始值不为 0 的变量

.rodata：只读数据（有些编译器会把只读数据放.text 段）

.bss：可读写全局初始值不为 0 的变量

.init：给 C++ 等面向对象用的，用于调用全局类变量的构造函数

.dll：动态链接用的地址表

等等。

程序读入到内存后，先创建进程。然后操作系统会给各个段按照链接好的地址分配物理页面并映射成虚拟页面，之后操作系统会给 .bss 段单独分配一些物理页。

操作系统会把所需的 dll 映射到程序的地址空间，按照空间里的 dll 初始化.dll 所需跳转地址表。然后操作系统，执行.init 段里面的代码，防止 C++ 全局变量无法成功构造。以及给程序分配一些其它的必要资源，比如消息队列，用于通信的端口。

之后 OS 给程序初始化堆，这个堆就是 `malloc/free`，`new/delete` 对应的内存，操作系统仅仅是分配了空间段，并没有真正分配几个物理页面，同一个操作系统上每个应用程序的堆都是一样大的。注意是地址空间，不是实际内存。操作系统本身有一个堆供操作系统自己使用，这个堆在内核空间，程序看不见也访问不到。程序用到的是用户堆，每个进程有一个，进程中的每个线程都从这个堆申请内存，这个堆在用户空间。

然后分配栈地址空间, 开始创建第一个线程, 并把可执行文件头里面指出的第一条指令地址交给这个线程, 开始运行。在创建线程的过程中从栈空间里面分配一块儿空间给这个线程。

3、

### 理解 volatile 关键字

也许你从来都没听说过 C 语言中有 volatile 这个关键字, 但它从 C 语言诞生开始就一直存在。volatile 关键字的作用十分特别, 它的作用是避免编译器对相应代码进行优化。你应该动手体会一下 volatile 的作用, 在 GNU/Linux 下编写以下代码:

```
void fun() {
    volatile unsigned char *p = (void *)0x8049000;
    *p = 0;
    while(*p != 0xff);
    *p = 0x33;
    *p = 0x34;
    *p = 0x86;
}
```

然后使用 -O2 编译代码。尝试去掉代码中的 volatile 关键字, 重新使用 -O2 编译, 并对比去掉 volatile 前后反汇编结果的不同。

你或许会感到疑惑, 代码优化不是一件好事情吗? 为什么会有 volatile 这种奇葩的存在? 思考一下, 如果代码中的地址 0x8049000 最终被映射到一个设备寄存器, 去掉 volatile 可能会带来什么问题?

在 Linux 中新建一个 c 文件写入如下代码:

```
int main()
{
    volatile unsigned char *p=(void*)0x80490000;
    *p=0;
    while(*p!=0xff);
    *p=33;
    *p=34;
    *p=86;
}
```

目录下的 terminal 键入 `gcc -c test.c -o test1 -O2`

再键入 `objdump -d test1`

该文件反汇编如下:

```
pqq@pqq-virtual-machine:~/桌面$ objdump -d test1
test1:      文件格式 elf32-i386

Disassembly of section .text.startup:

00000000 <main>:
 0:  c6 05 00 00 49 80 00      movb    $0x0,0x80490000
 7:  89 f6                    mov     %esi,%esi
 9:  8d bc 27 00 00 00 00      lea     0x0(%edi,%eiz,1),%edi
10:  0f b6 05 00 00 49 80      movzbl  0x80490000,%eax
17:  3c ff                    cmp     $0xff,%al
19:  75 f5                    jne     10 <main+0x10>
1b:  c6 05 00 00 49 80 21      movb    $0x21,0x80490000
22:  31 c0                    xor     %eax,%eax
24:  c6 05 00 00 49 80 22      movb    $0x22,0x80490000
2b:  c6 05 00 00 49 80 56      movb    $0x56,0x80490000
32:  c3                      ret
```

修改 C 文件, 去掉 volatile 关键字:

```
int main()
{
    unsigned char *p=(void*)0x80490000;
    *p=0;
    while(*p!=0xff);
    *p=33;
    *p=34;
    *p=86;
}
```

目录下的 terminal 键入 `gcc -c test.c -o test2 -O2`

再键入 `objdump -d test2`

该文件反汇编如下：

```
pqq@pqq-virtual-machine:~/桌面$ objdump -d test2
test2:      文件格式 elf32-i386

Disassembly of section .text.startup:

00000000 <main>:
0:  c6 05 00 00 49 80 00    movb   $0x0,0x80490000
7:  eb fe                  jmp     7 <main+0x7>
```

#### 【关于关键字 volatile】

volatile 的本意是“易变的”，因为访问寄存器要比访问内存单元快的多，所以编译器一般都会作减少存取内存的优化，但有可能会读脏数据。当要求使用 volatile 声明变量值的时候，系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据。精确地说就是，遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。如果不使用 volatile，则编译器将对所声明的语句进行优化。

用 volatile 定义的变量会在程序外被改变,每次都必须从内存中读取，而不能重复使用放在 cache 或寄存器中的备份。

简洁的说就是：volatile 关键词影响编译器编译的结果，用 volatile 声明的变量表示该变量随时可能发生变化，与该变量有关的运算，不要进行编译优化，以免出错。

#### 【如果代码中的地址 0x8049000 最终被映射到一个设备寄存器】

若去掉 volatile，如第二个反汇编图，会因编译器的优化丢失部分指令。

4、

#### 如何检测多个键同时被按下

在游戏中，很多时候需要判断玩家是否同时按下了多个键，例如 RPG 游戏中的八方向行走，格斗游戏中的组合招式等等，根据键盘码的特性，你知道这些功能是如何实现的吗？

在检测到键盘有键被按下后，会调用一个 press\_key 函数

```
#define KEYDOWN_MASK 0x8000
bool keyboard_event() {
    int keycode = _read_key();
    if (keycode == _KEY_NONE) return false;

    if((keycode & KEYDOWN_MASK) != 0){
        key_code = keycode & ~KEYDOWN_MASK;
        press_key(key_code);
        return true;
    }
    else{
        for(int i = 0; i < 26; i++){
            if(keycode == letter_code[i]){
                release_key(i);
            }
        }
        return true;
    }
}
```

在该函数中，会将被按下的键对应的标志位赋值为 true

```
/* 对应键按下的标志位 */
static bool letter_pressed[26];

void press_key(int scan_code) {
    int i;
    for (i = 0; i < 26; i++) {
        if (letter_code[i] == scan_code) {
            letter_pressed[i] = true;
        }
    }
}
```

而松开按键时，会调用 release\_key 函数，这个函数会将对应键的标志位恢复为 false

```
void release_key(int index) {
    letter_pressed[index] = false;
}
```

故只需要每次触发按键函数的时候，遍历 letter\_pressed 数组，查找 true 的元素的数量，即可知道现在有几个键被按下。

5、

### 必答题

你需要在实验报告中用自己的语言，尽可能详细地回答下列问题。

- ❖ 在 nemu/include/cpu/rtl.h 中，你会看到由 static inline 开头定义的各种 RTL 指令函数。选择其中一个函数，分别尝试去掉 static，去掉 inline 或去掉两者，然后重新进行编译，你会看到发生错误。请分别解释为什么会发生这些错误？你有办法证明你的想法吗？
- ❖ 了解 Makefile 请描述你在 nemu 目录下敲入 make 后，make 程序如何组织 .c 和 .h 文件，最终生成可执行文件 nemu/build/nemu。（这个问题包括两个方面：Makefile 的工作方式和编译链接的过程。）关于 Makefile 工作方式的提示：
  - ✧ Makefile 中使用了变量，包含文件等特性
  - ✧ Makefile 运用并重写了一些 implicit rules
  - ✧ 在 man make 中搜索 -n 选项，也许对你有帮助
  - ✧ RTFM

### 1) Rtl.h 中的 RTL 指令函数

```
46 static inline void rtl_mul(rtlreg_t* dest_hi, rtlreg_t* dest_lo, const rtlreg_t* src1, const rtlreg_t* src2) {
47     asm volatile("mul %3" : "=d"(*dest_hi), "=a"(*dest_lo) : "a"(*src1), "r"(*src2));
48 }
49
50 static inline void rtl_imul(rtlreg_t* dest_hi, rtlreg_t* dest_lo, const rtlreg_t* src1, const rtlreg_t* src2) {
51     asm volatile("imul %3" : "=d"(*dest_hi), "=a"(*dest_lo) : "a"(*src1), "r"(*src2));
52 }
53
54 static inline void rtl_div(rtlreg_t* q, rtlreg_t* r, const rtlreg_t* src1_hi, const rtlreg_t* src1_lo, const rtlreg_t* src2) {
55     asm volatile("div %4" : "=a"(*q), "=d"(*r) : "d"(*src1_hi), "a"(*src1_lo), "r"(*src2));
56 }
57
58 static inline void rtl_idiv(rtlreg_t* q, rtlreg_t* r, const rtlreg_t* src1_hi, const rtlreg_t* src1_lo, const rtlreg_t* src2) {
59     asm volatile("idiv %4" : "=a"(*q), "=d"(*r) : "d"(*src1_hi), "a"(*src1_lo), "r"(*src2));
60 }
```

**Static 函数：**在函数的返回类型前加上关键字 static，函数就被定义成为静态函数。普通函数的定义和声明默认情况下是 extern 的，但静态函数只是在声明他的文件当中可见，不能被其他文件所用。因此定义静态函数有以下好处：

- <1> 其他文件中可以定义相同名字的函数，不会发生冲突。
- <2> 静态函数不能被其他文件所用。

**Inline 函数：**内联函数和普通函数最大的区别在于内部的实现方面，而不是表面形式，普通函数在被调用时，系统首先要跳跃到该函数的入口地址，执行函数体，执行完成后再返回到函数调用的地方，函数始终只有一个拷贝；而内联函数则不需要进行一个寻址的过程，当执行到内联函数时，此函数展开（很类似宏的使用），如果在 N 处调用了此内联函数，则此函数就会有 N 个代码段的拷贝。

- a) 删掉 static，保留 inline：无报错
- b) 删掉 inline，保留 static：报错



```

pqq@pqq-virtual-machine: ~/桌面/PA/ics2018/nexus-am/tests/videotest
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
make[1]: Leaving directory '/home/pqq/桌面/PA/ics2018/nexus-am'
make[1]: Entering directory '/home/pqq/桌面/PA/ics2018/nexus-am/libs/klib'
make[1]: *** 没有指明目标并且找不到 makefile。 停止。
make[1]: Leaving directory '/home/pqq/桌面/PA/ics2018/nexus-am/libs/klib'
/home/pqq/桌面/PA/ics2018/nexus-am/Makefile.compile:86: recipe for target 'klib'
failed
make: [klib] Error 2 (ignored)
make[1]: Entering directory '/home/pqq/桌面/PA/ics2018/nemu'
+ CC src/cpu/decode/decode.c
In file included from ./include/cpu/decode.h:6:0,
                  from ./include/cpu/exec.h:9,
                  from src/cpu/decode/decode.c:1:
./include/cpu/rtl.h:46:13: error: 'rtl_mul' defined but not used [-Werror=unused
-function]
static void rtl_mul(rtlreg_t* dest_hi, rtlreg_t* dest_lo, const rtlreg_t* src1,
                  ^
cc1: all warnings being treated as errors
Makefile:23: recipe for target 'build/obj/cpu/decode/decode.o' failed
make[1]: *** [build/obj/cpu/decode/decode.o] Error 1
make[1]: Leaving directory '/home/pqq/桌面/PA/ics2018/nemu'
/home/pqq/桌面/PA/ics2018/nexus-am/Makefile.app:35: recipe for target 'run' fail
ed
make: *** [run] Error 2
pqq@pqq-virtual-machine: ~/桌面/PA/ics2018/nexus-am/tests/videotest$

```

c) 删掉 inline 和 static: 报错

```

pqq@pqq-virtual-machine: ~/桌面/PA/ics2018/nexus-am/tests/videotest
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
+ CC src/cpu/exec/exec.c
+ CC src/cpu/exec/control.c
+ CC src/cpu/exec/special.c
+ CC src/cpu/exec/cc.c
+ LD build/nemu
build/obj/cpu/decode/modrm.o: 在函数'rtl_mul'中:
/home/pqq/桌面/PA/ics2018/nemu/./include/cpu/rtl.h:47: 'rtl_mul'被多次定义
build/obj/cpu/decode/decode.o:/home/pqq/桌面/PA/ics2018/nemu/./include/cpu/rtl.h
:47: 第一次在此定义
build/obj/cpu/intr.o: 在函数'rtl_mul'中:
/home/pqq/桌面/PA/ics2018/nemu/./include/cpu/rtl.h:47: 'rtl_mul'被多次定义
build/obj/cpu/decode/decode.o:/home/pqq/桌面/PA/ics2018/nemu/./include/cpu/rtl.h
:47: 第一次在此定义
build/obj/cpu/exec/logic.o: 在函数'rtl_mul'中:
/home/pqq/桌面/PA/ics2018/nemu/./include/cpu/rtl.h:47: 'rtl_mul'被多次定义
build/obj/cpu/decode/decode.o:/home/pqq/桌面/PA/ics2018/nemu/./include/cpu/rtl.h
:47: 第一次在此定义
build/obj/cpu/exec/system.o: 在函数'rtl_mul'中:
/home/pqq/桌面/PA/ics2018/nemu/./include/cpu/rtl.h:47: 'rtl_mul'被多次定义
build/obj/cpu/decode/decode.o:/home/pqq/桌面/PA/ics2018/nemu/./include/cpu/rtl.h
:47: 第一次在此定义
build/obj/cpu/exec/prefix.o: 在函数'rtl_mul'中:
/home/pqq/桌面/PA/ics2018/nemu/./include/cpu/rtl.h:47: 'rtl_mul'被多次定义
build/obj/cpu/decode/decode.o:/home/pqq/桌面/PA/ics2018/nemu/./include/cpu/rtl.h

```

1、首先，inline 函数是不能像传统的函数那样放在.c 中然后在.h 中给出接口在其余文件中调用的，因为 inline 函数其实是跟宏定义类似，不存在所谓的函数入口。

2、因为第一点，会出现一个问题，就是说如果 inline 函数在两个不同的文件中出现，也就是说一个.h 被两个不同的文件包含，则会出现重名，链接失败。所以 static inline 的用法就能很好的解决这个问题，使用 static 修饰符，函数仅在文件内部可见，不会污染命名空间。可以理解为一个 inline 在不同的.C 里面生成了不同的实例，而且名字是完全相同的。

- 关于 static 和 inline 的问题化简讨论和尝试。

新建一个文件夹，文件夹中有 main.c 文件，include 了 2 个 h 文件，调用了两个函数。

```

main.c x
#include"pqq1.h"
#include"pqq2.h"
int main()
{
    pqq1();
    pqq2();
    return 0;
}

```

pqq1.h 和 pqq2.h 中各自声明了自己的函数

```
pqq1.h  x
void pqq1();

pqq2.h  x
void pqq2();
```

在 pqq1.c 和 pqq2.c 中实现了对应的函数，他们都 include 了 1 个 h 文件且调用了 test() 函数

```
pqq1.c  x
#include "help.h"
#include "pqq1.h"
void pqq1()
{
    int pqq1=1;
    test();
}

pqq2.c  x
#include "help.h"
#include "pqq2.h"
void pqq2()
{
    int pqq2=2;
    test();
}
```

在 help.h 文件中实现了 test 函数

```
help.h  x
static inline void test()
{
    int a=0;
    a=a+1;
}
```

我们在上述过程中提到的 3 个 h 文件+3 个 c 文件中实现了问题情况的简化，只需调整 help.h 中的 test 函数的 void 返回类型之前的 static 和 inline，即可得到相似结果，并且方便查看反汇编码，进而回答问题。

Static inline void test(): 键入编译和运行命令，成功编译和运行

```
pqq@pqq-virtual-machine:~/桌面/test1$ gcc main.c pqq1.c pqq2.c -o test
pqq@pqq-virtual-machine:~/桌面/test1$ ./test
```

a) Inline void test(): 编译失败

```
pqq@pqq-virtual-machine:~/桌面/test1$ gcc main.c pqq1.c pqq2.c -o test
/tmp/ccIC0g32.o: 在函数'pqq1'中:
pqq1.c:(.text+0xe): 对'test'未定义的引用
/tmp/cc5Uhed9.o: 在函数'pqq2'中:
pqq2.c:(.text+0xe): 对'test'未定义的引用
collect2: error: ld returned 1 exit status
```

b) Static void test(): 无报错

```
pqq@pqq-virtual-machine:~/桌面/test1$ gcc -c pqq2.c -o pqq2.o
pqq@pqq-virtual-machine:~/桌面/test1$ objdump -d pqq2.o

pqq2.o:          文件格式 elf32-i386

Disassembly of section .text:

00000000 <test>:
 0: 55                push    %ebp
 1: 89 e5             mov     %esp,%ebp
 3: 83 ec 10          sub     $0x10,%esp
 6: c7 45 fc 00 00 00 movl    $0x0,-0x4(%ebp)
 d: c7 45 fc 01 00 00 movl    $0x1,-0x4(%ebp)
14: 90                nop
15: c9                leave
16: c3                ret

00000017 <pqq2>:
17: 55                push    %ebp
18: 89 e5             mov     %esp,%ebp
1a: 83 ec 10          sub     $0x10,%esp
1d: c7 45 fc 02 00 00 movl    $0x2,-0x4(%ebp)
24: e8 d7 ff ff ff    call    0 <test>
29: 90                nop
2a: c9                leave
2b: c3                ret
```



c) Void test(): 报错显示重定义

```
pqq@pqq-virtual-machine:~/桌面/test1$ gcc main.c pqq1.c pqq2.c -o test
/tmp/ccRuev6f.o: 在函数'test'中:
pqq2.c:(.text+0x0): `test'被多次定义
/tmp/ccLLsutK.o:pqq1.c:(.text+0x0): 第一次在此定义
collect2: error: ld returned 1 exit status
```

```
pqq@pqq-virtual-machine:~/桌面/test1$ gcc -c pqq1.c -o pqq1.o
pqq@pqq-virtual-machine:~/桌面/test1$ objdump -d pqq1.o

pqq1.o:          文件格式 elf32-i386

Disassembly of section .text:

00000000 <test>:
 0: 55                push  %ebp
 1: 89 e5             mov   %esp,%ebp
 3: 83 ec 10          sub   $0x10,%esp
 6: c7 45 fc 00 00 00 movl  $0x0,-0x4(%ebp)
 d: c7 45 fc 01 00 00 movl  $0x1,-0x4(%ebp)
14: 90                nop
15: c9                leave
16: c3                ret

00000017 <pqq1>:
17: 55                push  %ebp
18: 89 e5             mov   %esp,%ebp
1a: 83 ec 10          sub   $0x10,%esp
1d: c7 45 fc 01 00 00 movl  $0x1,-0x4(%ebp)
24: e8 fc ff ff ff   call  25 <pqq1+0xe>
29: 90                nop
2a: c9                leave
2b: c3                ret
```

```
pqq@pqq-virtual-machine:~/桌面/test1$ gcc -c pqq2.c -o pqq2.o
pqq@pqq-virtual-machine:~/桌面/test1$ objdump -d pqq2.o

pqq2.o:          文件格式 elf32-i386

Disassembly of section .text:

00000000 <test>:
 0: 55                push  %ebp
 1: 89 e5             mov   %esp,%ebp
 3: 83 ec 10          sub   $0x10,%esp
 6: c7 45 fc 00 00 00 movl  $0x0,-0x4(%ebp)
 d: c7 45 fc 01 00 00 movl  $0x1,-0x4(%ebp)
14: 90                nop
15: c9                leave
16: c3                ret

00000017 <pqq2>:
17: 55                push  %ebp
18: 89 e5             mov   %esp,%ebp
1a: 83 ec 10          sub   $0x10,%esp
1d: c7 45 fc 02 00 00 movl  $0x2,-0x4(%ebp)
24: e8 fc ff ff ff   call  25 <pqq2+0xe>
29: 90                nop
2a: c9                leave
2b: c3                ret
```

2) 在 nemu 键入 make:

```
pqq@pqq-virtual-machine: ~/桌面/PA/ics2018/nemu
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
pqq@pqq-virtual-machine:~/桌面/PA/ics2018/nemu$ make
+ CC src/cpu/decode/decode.c
+ CC src/cpu/decode/modrm.c
+ CC src/cpu/intr.c
+ CC src/cpu/exec/logic.c
+ CC src/cpu/exec/system.c
+ CC src/cpu/exec/prefix.c
+ CC src/cpu/exec/arith.c
+ CC src/cpu/exec/data-mov.c
+ CC src/cpu/exec/exec.c
+ CC src/cpu/exec/control.c
+ CC src/cpu/exec/special.c
+ CC src/cpu/exec/cc.c
+ LD build/nemu
pqq@pqq-virtual-machine:~/桌面/PA/ics2018/nemu$
```

## ● 关于 make 和 Makefile

◆ Makefile: Makefile 文件描述了整个工程的编译、连接等规则。其中包括: 工程中

的哪些源文件需要编译以及如何编译、需要创建哪些库文件以及如何创建这些库文件、如何最后产生我们想要的可执行文件。尽管看起来可能是很复杂的事情，但是为工程编写 Makefile 的好处是能够使用一行命令来完成“自动化编译”，一旦提供一个（通常对于一个工程来说会是多个）正确的 Makefile。编译整个工程你所要做的唯一的一件事就是在 shell 提示符下输入 make 命令。整个工程完全自动编译，极大提高了效率。

- ◆ make: make 是一个命令工具，它解释 Makefile 中的指令（应该说是规则）。在 Makefile 文件中描述了整个工程所有文件的编译顺序、编译规则。Makefile 有自己的书写格式、关键字、函数。像 C 语言有自己的格式、关键字和函数一样。而且在 Makefile 中可以使用系统 shell 所提供的任何命令来完成想要的工作。Makefile（在其它的系统上可能是另外的文件名）在绝大多数的 IDE 开发环境中都在使用，已经成为一种工程的编译方法。
- ◆ Makefile 工作方式：
  - 1、make 会在当前目录下找名字叫“Makefile”或“makefile”的文件。
  - 2、如果找到，它会找文件中的第一个目标文件（target），并把这个文件作为最终的目标文件。
  - 3、如果目标文件不存在，或是目标文件所依赖的后面的 .o 文件的文件修改时间要比目标文件新，那么就会执行后面所定义的命令来生成目标文件。
  - 4、如果目标文件所依赖的.o 文件也不存在，那么 make 会在当前文件中找目标为.o 文件的依赖性，如果找到则再根据那一个规则生成.o 文件。
  - 5、C 文件和 H 文件是存在的，于是 make 会生成 .o 文件，然后再用 .o 文件声明 make 的任务，也就是可执行文件。

# 遇到的坑

- 1、配置失败。没好好听课，查 readme 只添加了一个环境变量 AM\_HOME



然后

```
pqq@pqq-virtual-machine:~/桌面/PA/ics2018/nexus-am/tests/cputest$ make ARCH=x86-nemu ALL=dummy run
Building dummy [x86-nemu]
+ CC tests/dummy.c
Building am [x86-nemu]
make[2]: *** 没有指明目标并且找不到 makefile。 停止。
make[2]: *** run: 没有那个文件或目录。 停止。
/home/pqq/桌面/PA/ics2018/nexus-am/Makefile.app:35: recipe for target 'run' failed
make[1]: *** [run] Error 2
Makefile:12: recipe for target 'Makefile.dummy' failed
make: [Makefile.dummy] Error 2 (ignored)
dummy
```

询问助教姐姐后把 3 个环境变量补齐

```
118
119 export NEMU_HOME=/home/lxz/下载/ics2018/nemu
120 export AM_HOME=/home/lxz/下载/ics2018/nexus-am
121 export NAVY_HOME=/home/lxz/下载/ics2018/navy-apps
~/.bashrc[1]
```

顺利运行!

```
pqq@pqq-virtual-machine: ~/桌面/PA/ics2018/nexus-am/tests/cputest
pqq@pqq-virtual-machine:~/桌面/PA/ics2018/nexus-am/tests/cputest$ make ARCH=x86-nemu ALL=dummy run
Building dummy [x86-nemu]
Building am [x86-nemu]
make[2]: *** 没有指明目标并且找不到 makefile。 停止。
[src/monitor/monitor.c,65,load_img] The image is /home/pqq/桌面/PA/ics2018/nexus-am/tests/cputest/build/dummy-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 20:03:52, May 9 2019
For help, type "help"
(nemu)
```

- 2、实现第一个指令 call 的时候，没捉摸手册，看反汇编还觉得 10010 是个立即数，替换 EMPTY 的时候传参传了 I 而不是 J……事实证明学会看手册是多么重要，悲伤蛙.jpg

```
00100000 <_start>:
00100000: bd 00 00 00 00      mov     $0x0,%ebp
00100005: bc 00 7c 00 00      mov     $0x7c00,%esp
0010000a: e8 01 00 00 00      call   100010 <_trn_init>
0010000f: 90                  nop
```

- 3、实现第一个指令 call 的时候，只是根据实验手册按自己的想象实现 call 指令，并且也能跑，如下图：

call:call 指令有很多形式，不过在 PA 中只会用到其中的几种，现在只需要实现 CALL rel32 的形式就可以了。%eip 的跳转可以通过将 decoding.is\_jump 设为 1，并将 decoding.jump\_eip 设为跳转目标地址来实现，这时在 update\_eip() 函数中会把跳转目标地址作为新的 %eip，而不是顺序意义下的下一条指令的地址

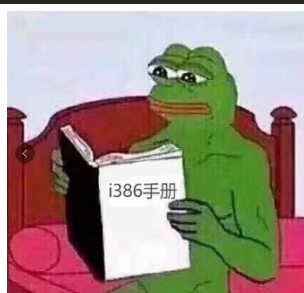
```
make_EHelper(call) {
    // the target address is calculated at the decode stage
    //TODO();
    decoding.is_jump=1;
    decoding.jump_eip = id_dest->val+*eip;
```

但后面听了学姐的课，尽早实现了 Differential Testing 之后，就，过不去了，然后老老实实重新开始去研究 i386 手册并且重写。

类似的指令还有好几个，哭。

```
Operation
IF rel16 or rel32 type of call
THEN (* near relative call *)
    IF OperandSize = 16
    THEN
        Push(IP);
        EIP ← (EIP + rel16) AND 0000FFFFH;
    ELSE (* OperandSize = 32 *)
        Push(EIP);
        EIP ← EIP + rel32;
    FI;
FI;
```

```
make_EHelper(call) {
    // the target address is calculated at the decode stage
    //TODO();
    //decoding.is_jump=1;
    //decoding.jump_eip = id_dest->val+*eip;
    rtl_push(&decoding.seq_eip);
    decoding.is_jump=1;
    print_asm("call %x", decoding.jump_eip);
}
```



事实再次证明看手册是多么重要。

- 4、执行 PA2 的时候有个奇怪的格子乱码，以为是个编码 BUG 之类的，找了好久，1 小时吧，后来助教姐姐说……不用管它，嗯。

```
(nemu) si 5
100013: 56
100014: 53
100015: 31 f6
100017: bb f9 03 00 00
10001c: 89 f0
(nemu) si 5
10001e: 89 da
100020: ee
100021: b9 fb 03 00 00
100026: b0 80
100028: 89 ca
(nemu) si 5
10002a: ee
10002b: ba f8 03 00 00
100030: b0 01
100032: ee
100033: 89 f0
(nemu) \
```

- 5、第一次理论上实现我 dummy.c 的所有所需指令后开始运行，然而跑到底之后，居然又触发了未实现的指令？

经查找是个 add 指令，我寻思着反汇编里好像没有 add 指令，但还是非常实诚地去加上 add 指令了，然后接着又继续触发下一个未实现指令，我：？？？

跟同学讨论后，发现我的 make\_DopHelper(SI)实现有错误，当时写的时候根据 TODO 的提醒和模仿上面的 make\_DopHelper(I)来写，没考虑 width，也没把 rtl\_li 模仿进来……

```
static inline make_DopHelper(I) {
    /* eip here is pointing to the immediate */
    op->type = OP_TYPE_IMM;
    op->imm = instr_fetch(eip, op->width);
    rtl_li(&op->val, op->imm);

#ifdef DEBUG
    snprintf(op->str, OP_STR_SIZE, "%0x%x", op->imm);
#endif
}

/* I386 manual does not contain this abbreviation, but it is different from
 * the one above from the view of implementation. So we use another helper
 * function to decode it.
 */
/* sign immediate */
static inline make_DopHelper(SI) {
    assert(op->width == 1 || op->width == 4);

    op->type = OP_TYPE_IMM;

    /* TODO: Use instr_fetch() to read 'op->width' bytes of memory
     * pointed by 'eip'. Interpret the result as a signed immediate,
     * and assign it to op->imm.
     */
    op->imm = instr_fetch(eip, op->width);
}
```

修改好之后就运行正确了

```
static inline make_DopHelper(SI) {
    assert(op->width == 1 || op->width == 4);

    op->type = OP_TYPE_IMM;

    /* TODO: Use instr_fetch() to read 'op->width' bytes of memory
     * pointed by 'eip'. Interpret the result as a signed immediate,
     * and assign it to op->imm.
     */
    //op->imm = instr_fetch(eip, op->width);
    //TODO();

    if (op->width == 4)
        op->imm = instr_fetch(eip, op->width);

    else if (op->width == 2)
        op->imm = (int16_t)((uint16_t)instr_fetch(eip, op->width));

    else
        op->imm = (int16_t)(int8_t)((uint8_t)instr_fetch(eip, op->width));

    rtl_li(&op->val, op->imm);
}
```

- 6、在 PA2.2 中手册提到

```
--- nexus-am/Makefile.check
+++ nexus-am/Makefile.check
@@ -7,2 +7,2 @@
-ARCH ?= native
+ARCH ?= x86-nemu
ARCH = $(shell ls $(AM_HOME)/am/arch/)
```

然后在 nexus-am/tests/cputest/ 目录下执行

```
make ALL=xxx run
```

其中 xxx 为测试用例的名称(不包含.c 后缀)。

然后愉快地结束了那一天的学习，去休息了。第二天起来，把这事儿忘了，然后就一直在 native 底下把 pa2 做完，每次运行就用的 make ARCH=x86-nemu ALL=某文件 run，然后倒也没啥问题……

于是实现 PA2.3 的时候，感慨自己技术飞涨太快，居然一路顺顺利利……？？？  
后来到了 PA3，开始出现诡异问题，程序一直跑到不该跑的地方去，比如：

```
pqq@pqq-virtual-machine:~/桌面/PA/ics2018/nanos-lite$ make run
Building nanos-lite [native]
+ CC src/syscall.c
src/syscall.c: In function 'do_syscall':
src/syscall.c:6:10: error: implicit declaration of function 'SYSCALL_ARG1' [-Werror=implicit-function-declaration]
    a[0] = SYSCALL_ARG1(r);
           ^
cc1: all warnings being treated as errors
/home/pqq/桌面/PA/ics2018/nexus-am/Makefile.compile:69: recipe for target '/home/pqq/桌面/PA/ics2018/nanos-lite/build/native/./src/syscall.o' failed
make: *** [/home/pqq/桌面/PA/ics2018/nanos-lite/build/native/./src/syscall.o] Error 1
```

然后咱也不知道为啥，咱也不敢问，就 debug 吧，毕竟误跳到的地方也是未实现的函数。于是耽误一周。

后来实在是做不出来了，再次求助温柔学姐



先问个问题，你改x86-nemu了吗，你的提示里面写的是building nanos-lite native

2019年5月23日 18:45



你先改一下x86-nemu看看效果。理论上你刚进去pa3还没执行到do-syscall应该不会报这个错的

好的吧。按学姐的提示改了到 x86 之后，回到 PA3 重新开始运行。  
可是，怎么报错的地方还是不对啊！！

```
pqq@pqq-virtual-machine:~/桌面/PA/ics2018/nanos-lite$ make run
Building nanos-lite [x86-nemu]
+ CC src/syscall.c
+ CC src/proc.c
+ CC src/fs.c
+ CC src/frandisk.c
+ CC src/mm.c
+ CC src/main.c
+ AS src/initrd.S
+ CC src/loader.c
+ CC src/lrq.c
+ CC src/device.c
make[1]: Entering directory '/home/pqq/桌面/PA/ics2018/nexus-an'
make[2]: Entering directory '/home/pqq/桌面/PA/ics2018/nexus-an/an'
Building an [x86-nemu]
+ CC arch/x86-nemu/src/pte.c
+ CC arch/x86-nemu/src/trm.c
+ CC arch/x86-nemu/src/loef.c
arch/x86-nemu/src/loef.c: In function 'read_key':
arch/x86-nemu/src/loef.c:40:23: error: 'I8042_STATUS_PORT' undeclared (first use in this function)
    uint8_t press = inb(I8042_STATUS_PORT);
                        ^
arch/x86-nemu/src/loef.c:40:23: note: each undeclared identifier is reported only once for each function it appears in
arch/x86-nemu/src/loef.c:42:15: error: 'I8042_DATA_PORT' undeclared (first use in this function)
    return inl(I8042_DATA_PORT);
                ^
arch/x86-nemu/src/loef.c:46:1: error: control reaches end of non-void function [-Werror=return-type]
}
^
cc1: all warnings being treated as errors
/home/pqq/桌面/PA/ics2018/nexus-an/Makefile.compile:69: recipe for target '/home/pqq/桌面/PA/ics2018/nexus-an/an/build/x86-nemu/./arch/x86-nemu/src/loef.o' failed
make[2]: *** [/home/pqq/桌面/PA/ics2018/nexus-an/an/build/x86-nemu/./arch/x86-nemu/src/loef.o] Error 1
make[2]: Leaving directory '/home/pqq/桌面/PA/ics2018/nexus-an/an'
makefile:6: recipe for target 'all' failed
make[1]: *** [all] Error 2
make[1]: Leaving directory '/home/pqq/桌面/PA/ics2018/nexus-an'
/home/pqq/桌面/PA/ics2018/nexus-an/Makefile.compile:84: recipe for target 'an' failed
make: *** [an] Error 2
pqq@pqq-virtual-machine:~/桌面/PA/ics2018/nanos-lite$
```

于是：



惨痛的事实告诉我们听课是多么重要。

然后 git checkout 回到 PA2, 跑一波 PA2.3, 得, 整个儿全崩了。重来吧。这个纠结了一整个晚上。

PS: 在自己刚开始弄 PA3 到处瞎 debug 后来却总是弄不好的时候, 我还以为是自己把代码写坏了。然后尝试重置 PA3, 反反复复好多次。请教学姐之后重回 PA2, 我发现 bash runall.sh 运行失败, 显示 NEMU compile failed。也不知道是自己哪里搞崩了。幸好有备份代码, 然后反复备份反复重置反复瞎搞了一小时, 给弄好了。我也不知道为啥, 卢老师说咱是搞科学的不能列玄学问题, 这份经历我就不单独列一点了。



# 复习的知识和学到的经验

- 1、学会了一些 Linux 下好用的命令，如在 ics2018 目录下打开 terminal，键入 `grep -rn "xxxxx"`，可以直接全局搜索 xxxxx 字符串在哪些文档的哪一行出现过。
- 2、对宏的运用有了超级超级超级深刻的体验。一开始一直找不到程序怎么跑，甚至不能理解 `opcode_table` 数组，用了一整个晚上把流程串起来。
- 3、学看 i386 手册……
- 4、复习了计算机组成原理，主要是计算机实现指令的流程。
- 5、再次学习了一些 C++ 里我不太会用的保留字，比如 `static` 和 `inline`，对西加加里文件的链接和运行等过程的理解更深刻了呢。
- 6、真的好难