
南京大学

# PA-2-1：指令解码与执行

课程名称：计算机系统基础

姓名：孙文博

学号：201830210

邮箱：201830210@smail.nju.edu.cn

实验时间：2022.3.24 - 2022.4.14

# 一、 实验目的

1. 复习理论课中指令部分的相关知识；

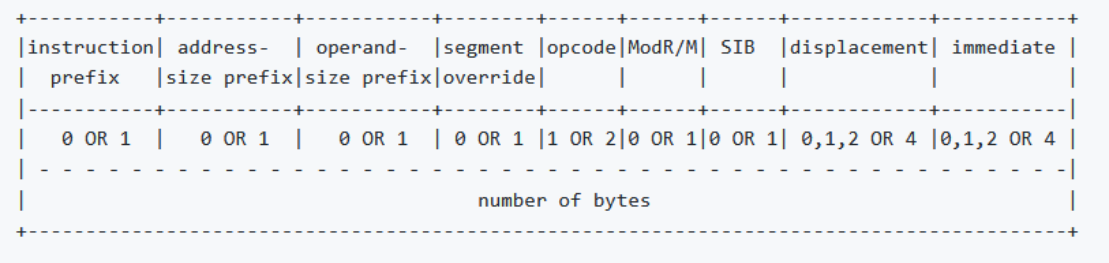
2. 了解 NEMU 的框架代码和宏定义；

3. 实现对指令的解码与执行的模拟。

# 二、 实验背景

在计算机中，一切信息都按照某种规则编码为由 01 串组成的数据形式，在 PA-1 中所介绍的整数和浮点数是如此，指挥机器运行的指令也是如此。每一条指令都指明了计算机所需要进行的一步操作（操作码）和操作的对象（操作数）。如果我们使用机器语言写程序，那么一个程序就是一个指令的序列，这个序列规定了计算机解决问题所需要执行的各个步骤。如果我们使用高级语言（如 C 语言）写程序，则往往需要使用编译器将高级语言程序编译成指令序列后再交给机器去执行。

NEMU 模拟的是 IA-32 体系结构，其指令集体系结构为对应的 i386 架构，格式如下：




其中包括指令的操作码（opcode），前缀（prefix），操作数寻址方式（ModR/M、SIB、displacement）和立即数（Immediate）等。对于单条指令的解码与执行的具体步骤在实验手册中详细给出，我们需

要理解如何通过 i386 手册查询得到各条指令信息的过程，重点关注 NEMU 然后通过 C 语言模拟这个过程。

# 三、 实验过程

## 1. 理解框架代码

PA-2-1 中有大量的已经搭好的框架代码，基本实现了一个可以反复执行指令的 CPU，这里我们列举最主要的框架代码，首先是 instr_helper.h 文件中包括了不同指令对应的宏：

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

// macro for generating the implementation of an instruction with one operand
// 用一个操作数生成指令的宏
#define make_instr_impl_1op(inst_name, src_type, suffix)
                                                    \
    make_instr_func(concat5(inst_name, _, src_type, _, suffix))
                                                    \
    {
                                                    \
        int len = 1;
                                                        \
        concat(decode_data_size_, suffix)
                                                        \
        concat3(decode_operand, _, src_type)
                                                    \
        print_asm_1(#inst_name, opr_src.data_size == 8 ? "b" : (opr_src.data_size
            == 16 ? "w" : "l"), len, &opr_src); \
        instr_execute_1op();
                                                        \
        return len;
                                                        \
    }
```

它将同一条指令，不同操作数类型的函数统一到一起，可以用一行宏定义展开(属于是把宏玩明白了)，最大程度减少了代码的重复。此外 instr_helper.h 函数中还定义了各种操作数类型和操作数长度以及条件判断部分（condition）。

其次是 opcode.c 文件，这里包含了当前可以实现的所有指令的操作码，包括 0f 开头的双字节操作码部分以及组 group 操作码部分：

opcode.c　网络正常　心情如何　顺利　遇到挑战　受挫　绝望

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

instr_func opcode_entry[256] = {
    /* 0x00 - 0x03*/ add_r2rm_b, add_r2rm_v, add_rm2r_b, add_rm2r_v,
    /* 0x04 - 0x07*/ inv, add_i2a_v, inv, inv,
    /* 0x08 - 0x0b*/ inv, or_r2rm_v, or_rm2r_b, inv,
    /* 0x0c - 0x0f*/ inv, inv, inv, opcode_2_byte/*两字节操作码*/,
    /* 0x10 - 0x13*/ inv, adc_r2rm_v, inv, inv,
    /* 0x14 - 0x17*/ inv, inv, inv, inv,
    /* 0x18 - 0x1b*/ sbb_r2rm_b, inv, inv, sbb_rm2r_v,
    /* 0x1c - 0x1f*/ inv, inv, inv, inv,
    /* 0x20 - 0x23*/ inv, and_r2rm_v, and_rm2r_b, inv,
    /* 0x24 - 0x27*/ inv, and_i2a_v, inv, inv,
    /* 0x28 - 0x2b*/ inv, sub_r2rm_v, sub_rm2r_b, sub_rm2r_v,
    /* 0x2c - 0x2f*/ inv, sub_i2a_v, inv, inv,
    /* 0x30 - 0x33*/ inv, xor_r2rm_v, inv, inv,
    /* 0x34 - 0x37*/ inv, inv, inv, inv,
    /* 0x38 - 0x3b*/ inv, cmp_r2rm_v, cmp_rm2r_b, cmp_rm2r_v,
    /* 0x3c - 0x3f*/ cmp_i2a_b, cmp_i2a_v, inv, inv,
    /* 0x40 - 0x43*/ inc_r_v, inc_r_v, inc_r_v, inc_r_v,
    /* 0x44 - 0x47*/ inc_r_v, inc_r_v, inc_r_v, inc_r_v,
    /* 0x48 - 0x4b*/ dec_r_v, dec_r_v, dec_r_v, dec_r_v,
    /* 0x4c - 0x4f*/ dec_r_v, dec_r_v, dec_r_v, dec_r_v,
    /* 0x50 - 0x53*/ push_r_v, push_r_v, push_r_v, push_r_v,
    /* 0x54 - 0x57*/ push_r_v, push_r_v, push_r_v, push_r_v,
    /* 0x58 - 0x5b*/ pop_r_v, pop_r_v, pop_r_v, pop_r_v,
    /* 0x5c - 0x5f*/ pop_r_v, pop_r_v, pop_r_v, pop_r_v,
    /* 0x60 - 0x63*/ push_a, pop_a, inv, inv,
    /* 0x64 - 0x67*/ inv, inv, data_size_16, inv,
    /* 0x68 - 0x6b*/ push_i_v, inv, push_i_b, inv,
    /* 0x6c - 0x6f*/ inv, inv, inv, inv,
    /* 0x70 - 0x73*/ inv, inv, jb_short_, jae_short_,
    /* 0x74 - 0x77*/ je_short_, jne_short_, jna_short_, ja_short_,
```

此外还有 nodrm.c，operand.c 等等，在此不一一列举了。

## 2. 实现各条指令

理解了各种头文件、宏定义等框架代码，接下来就轮到我们一条条读指令、查 i386 手册、写指令的时候了（有点像人形 CPU）。一开始进度较慢，写完一条指令就 make test 一次，后来发现指令之间的联系也很密切，一条指令往往稍加改变也可以变成另一条指令（如 add, sub, and, or 等等），这里列举最经典的 add 指令内容：

```
#include "cpu/instr.h"
/*
Put the implementations of `add' instructions here.
*/

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

make_instr_impl_2op(add, r, rm, b);
make_instr_impl_2op(add, r, rm, v);
make_instr_impl_2op(add, rm, r, b);
make_instr_impl_2op(add, rm, r, v);
make_instr_impl_2op(add, i, rm, v);
make_instr_impl_2op(add, i, a, v);
// make_instr_impl_2op(add, i, rm, bv);

make_instr_func(add_i2rm_bv)
{
    int len = 1;
    opr_dest.data_size = data_size;
    len +=modrm_rm(eip+1,&opr_dest);
    opr_src.data_size=8;
    opr_src.type = OPR_IMM;
    opr_src.addr = eip+len;

    operand_read(&opr_src);
```

## 3. 测试运行

经过了九九八十一难（实际不止 81 条指令）后，我们可以进行测试：

```
nemu: HIT GOOD TRAP at eip = 0x00030125
NEMU2 terminated
./nemu/nemu --autorun --testcase matrix-mul
NEMU load and execute img: ./testcase/bin/matrix-mul.img  elf: ./testcase/bin/matrix-mul
nemu: HIT GOOD TRAP at eip = 0x00030172
NEMU2 terminated
./nemu/nemu --autorun --testcase mul-longlong
NEMU load and execute img: ./testcase/bin/mul-longlong.img  elf: ./testcase/bin/mul-longlong
nemu: HIT GOOD TRAP at eip = 0x0003013a
NEMU2 terminated
./nemu/nemu --autorun --testcase prime
NEMU load and execute img: ./testcase/bin/prime.img  elf: ./testcase/bin/prime
nemu: HIT GOOD TRAP at eip = 0x00030093
NEMU2 terminated
./nemu/nemu --autorun --testcase shuixianhua
NEMU load and execute img: ./testcase/bin/shuixianhua.img  elf: ./testcase/bin/shuixianhua
nemu: HIT GOOD TRAP at eip = 0x00030114
NEMU2 terminated
./nemu/nemu --autorun --testcase sum
NEMU load and execute img: ./testcase/bin/sum.img  elf: ./testcase/bin/sum
nemu: HIT GOOD TRAP at eip = 0x00030048
NEMU2 terminated
./nemu/nemu --autorun --testcase wanshu
NEMU load and execute img: ./testcase/bin/wanshu.img  elf: ./testcase/bin/wanshu
nemu: HIT GOOD TRAP at eip = 0x00030091
NEMU2 terminated
./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
pa201830210@edb32e250119:~/pa_nju$
```

其中屏幕上的每一个 good 都凝结了无数的汗水和鲜血（😭）。

我们可以用过使用特制的 objdumpnemu 进行反汇编，查看每个测试样例的具体指令：

```
Disassembly of section .text:

00030000 <start>:
   30000:       e9 00 00 00 00          jmp    30005 <main>

00030005 <main>:
   30005:       55                      push   %ebp
   30006:       89 e5                   mov    %esp,%ebp
   30008:       83 e4 f8                and    $0xfffffff8,%esp
   3000b:       83 ec 10                sub    $0x10,%esp
   3000e:       e8 c3 00 00 00          call   300d6 <__x86.get_pc_thunk.dx>
   30013:       81 c2 ed 2f 00 00       add    $0x2fed,%edx
   30019:       d9 82 00 e0 ff ff       flds   -0x2000(%edx)
   3001f:       d9 5c 24 0c             fstps  0xc(%esp)
   30023:       d9 e8                   fld1
   30025:       d9 5c 24 08             fstps  0x8(%esp)
   30029:       d9 44 24 0c             flds   0xc(%esp)
   3002d:       d8 44 24 08             fadds  0x8(%esp)
   30031:       d9 5c 24 04             fstps  0x4(%esp)
   30035:       dd 82 08 e0 ff ff       fldl   -0x1ff8(%edx)
   3003b:       d9 44 24 04             flds   0x4(%esp)
   3003f:       da e9                   fucompp
   30041:       df e0                   fnstsw %ax
   30043:       80 e4 45                and    $0x45,%ah
   30046:       80 fc 40                cmp    $0x40,%ah
   30049:       74 06                   je     30051 <main+0x4c>
   3004b:       b8 01 00 00 00          mov    $0x1,%eax
   30050:       82                      nemu_trap
   30051:       d9 44 24 0c             flds   0xc(%esp)
   30055:       d8 4c 24 08             fmuls  0x8(%esp)
   30059:       d9 5c 24 04             fstps  0x4(%esp)
   3005d:       dd 82 10 e0 ff ff       fldl   -0x1ff0(%edx)
   30063:       d9 44 24 04             flds   0x4(%esp)
   30067:       da e9                   fucompp
   30069:       df e0                   fnstsw %ax
   3006b:       80 e4 45                and    $0x45,%ah
   3006e:       80 fc 40                cmp    $0x40,%ah
   30071:       74 06                   je     30079 <main+0x74>
```

至此，PA-2-1 圆满完成！

# 四、 思考题

1. 使用 hexdump 命令查看测试用例的.img 文件，所显示的.img 文件的内容对应模拟内存的哪一个部分？指令在机器中表示的形式是什么？

   查看 add.img 文件的内容如下：

```
pa201830210@edb32e250119:~/pa_nju/testcase/bin$ hexdump add.img
0000000 00e9 0000 5500 e589 8353 10ec 8fe8 0000
0000010 8100 efc2 002f c700 f045 0000 0000 45c7
0000020 00f8 0000 eb00 c748 f445 0000 0000 34eb
0000030 458b 8bf8 828c 0020 0000 458b 8bf4 8284
0000040 0020 0000 1c8d 8b01 f045 488d 8901 f04d
0000050 848b 4082 0000 3900 74c3 b806 0001 0000
0000060 ff82 f445 458b 83f4 07f8 c476 45ff 8bf8
0000070 f845 f883 7607 83b0 f87d 7408 b806 0001
0000080 0000 8382 f47d 7408 b806 0001 0000 b882
0000090 0000 0000 b882 0000 0000 c483 5b10 c35d
00000a0 148b c324 0000 0000 0000 0000 0000 0000
00000b0 0000 0000 0000 0000 0000 0000 0000 0000
*
0001000 0014 0000 0000 0000 7a01 0052 7c01 0108
0001010 0c1b 0404 0188 0000 0020 0000 001c 0000
0001020 efe5 ffff 009b 0000 4100 080e 0285 0d42
0001030 4405 0383 9202 41c3 0cc5 0404 0010 0000
0001040 0040 0000 f05c ffff 0004 0000 0000 0000
0001050 0000 0000 0000 0000 0000 0000 0000 0000
*
0003020 0000 0000 0001 0000 0002 0000 ffff 7fff
0003030 0000 8000 0001 8000 fffe ffff ffff ffff
0003040 0000 0000 0001 0000 0002 0000 ffff 7fff
0003050 0000 8000 0001 8000 fffe ffff ffff ffff
0003060 0001 0000 0002 0000 0003 0000 0000 8000
0003070 0001 8000 0002 8000 ffff ffff 0000 0000
0003080 0002 0000 0003 0000 0004 0000 0001 8000
0003090 0002 8000 0003 8000 0000 0000 0001 0000
00030a0 ffff 7fff 0000 8000 0001 8000 fffe ffff
00030b0 ffff ffff 0000 0000 fffd 7fff fffe 7fff
00030c0 0000 8000 0001 8000 0002 8000 ffff ffff
00030d0 0000 0000 0001 0000 fffe 7fff ffff 7fff
00030e0 0001 8000 0002 8000 0003 8000 0000 0000
00030f0 0001 0000 0002 0000 ffff 7fff 0000 8000
0003100 fffe ffff ffff ffff 0000 0000 fffd 7fff
0003110 fffe 7fff ffff 7fff fffc ffff fffd ffff
0003120 ffff ffff 0000 0000 0001 0000 fffe 7fff
0003130 ffff 7fff 0000 8000 fffd ffff fffe ffff
0003140
```

根据 NEMU 的约定，这部分内容将放在虚拟内存 0x30000 处（因为此时还没有实现 ELF 装载），且之后 NEMU 初始化会做两件事情：

1. 把测试用例镜像文件的内容直接拷贝到内存从 LOAD_OFF = 0x30000 处开始的连续区域内；
2. 将 EIP 初始化为 INIT_EIP = 0x30000。

指令在机器中的表示为一串 01 序列。

## 2. 如果去掉 instr_execute_2op () 函数前面的 static 关键字会发生什么情况？为什么？

static 函数的作用域与普通函数作用域不同,仅在本文件内。只在当前源文件中使用的函数应该说明为内部函数(static 修饰的函

数)，内部函数应该在当前源文件中说明和定义。对于可在当前源文件以外使用的函数，应该在一个头文件中说明，要使用这些函数的源文件要包含这个头文件。

而我们实现的许多条指令（如 mov，add 等）都有各种单独的 instr_execute_2op（）函数，若去掉 static 关键字，会造成多重定义，编译链接的时候会报错。

### 3. 为什么 test-float 会 fail？以后在写和浮点数相关的程序的时候要注意什么?

使用浮点栈存储浮点数时为 10 字节,而 float 类型所占空间为 4 字节，因此在内存单元和浮点栈之间进行数据传送的过程中，可能会发生精度损失，从而造成计算结果错误。

处理浮点数要注意各个部分之间的精度转换。

## 五、 总结与反思

早有耳闻的 PA-2-1 实验，其难度果然非同一般。PA 课上听的云里雾里的宏定义和各种操作，还是亲身实践后才理解是什么原理。当然，整个过程是以时间为代价的，反复查阅，反复修改，反复测试，最终可能才收获一个小小的 GOOD。

可是正所谓纸上得来终觉浅，绝知此事要躬行。在经历 PA-2-1 手撕 i386 指令的过程后，我对课本上的指令架构才有了更深入的理解，知道了哪些指令需要实现哪些功能，又为什么需要实现这些功能。

总的来说，PA-2-1 是一个令人难忘的过程，也许未来工作时的我再次看到 i386 手册，依旧会想起曾经做 PA 时那些脱发的夜晚。希望下一阶段的 PA 实验能够再接再厉吧，加油！