PA2（上） - 指令系统

17307130178 宁晨然

一、概述

本次 PA2.1 实验需要完成 i386 指令系统部分的阅读、框架代码的阅读、基本指令的实现三个板块。重点在于理解 i386 系统取指、译码、执行的过程，并且根据流程实现基本指令，其中还涉及到了 EFLAGS 寄存器的结构、初始化和更新。还需要完成打印变量、打印栈帧。

本实验难点在于框架代码的阅读，由于接触陌生的框架，需要将每个宏定义、函数引用的关系找好，找到对应的定义和文件组织方式。还有 EFLAGS 需要多查询 i386 系统的实现方式。

二、实验过程

（一）、i386 指令系统的了解

本实验的最终目的是实现机器能够完成执行指令，所以需要了解机器执行指令前的所有过程。首先看到 i386 对于指令的分析我有些迷惑，正如介绍中写的，i386 不惜牺牲大量的操作指南空间来构造一个“指令集极其复杂”但“代码密度低”的世界。

在阅读完[x86 指令系统简介]和 i386 中部分相关内容后，我简单对指令系统有了了解，此处简述，之后我按照这个思路进行指令的实现：

实现程序需要①取指②译码③执行

先讲译码，根据 decode.h 文件等定义的，其实译码工作的大部分模板已经提供，我只需要选择其中的模板例如 r2i 等，就可以根据指令 opcode 确定之后需要如何译码。而译码的过程实质是程序读入 eip 所指向的二进制序列，读入多少、如何存放（译码过程）取决于之前读入的指令或者（modRM）。

取指过程，根据 i386 附录 3 的 opcodeMap 可以获取信息，每个指令，例如 sub/mov 等都（大部分）唯一对应一个字节（本实验没有接触两个或三个字节的 opcode），所以根据指令的 index 可以在 opcodemap 中取到“菜谱”，而 opcodemap 就类似于“菜单”。但是对于每个指令又有很多变形，例如已经实现的 mov 指令，有很多根据 mov 的立即数或者寄存器变化的实例。

执行过程，就是实现本指令需要做的事情。可能实现的有算术、杂项、移动等等操作，根据 i386 的每个 instruction 的 description 可以写出代码。

对于指令集的细节不予赘述，在实现指令过程中写出。

（二）、框架代码

本次需要阅读的代码内容大部分在 cpu 文件中。整体结构在指南中写出，下面分析细节。

(1):include/cpu

①helper.h:

定义 make_helper()最关键的函数，这个函数的内容即指令的实现，返回值为本次操作读入的字节数。instr_fecth()用于取指令，在程序执行时，可以通过这个函数获得第一个字节的 opcode 判断现在的指令数，决定之后的操作；也可以在译码过程中用于读入之后的操作，例如 modRM。index()用于译码和执行指令，本次实验应该不需要写这个函数相关内容。

此处还有个关键的全局变量 Operands ops_decoded，各处访问时可以直接使用

```
typedef struct {
    uint32_t CF:1;
    uint32_t :1;
    uint32_t PF:1;
    uint32_t :1;
    uint32_t AF:1;
    uint32_t :1;
    uint32_t ZF:1;
    uint32_t SF:1;
    uint32_t TF:1;
    uint32_t IF:1;
    uint32_t DF:1;
    uint32_t OF:1;
    uint32_t IOPL:2;
    uint32_t NF:1;
    uint32_t :1;
    uint32_t RF:1;
    uint32_t VM:1;
    uint32_t :14;
} EFLAGS ;
```

op_src/op_dest/op_src2。一般在算术等指令中，需要对读入的操作数/寄存器进行加减乘除逻辑运算等，可以直接使用该变量。（变量定义在 operand.h 中，定义内容没什么好讲的）。

```
typedef union {
    union {
        uint32_t _32;
        uint16_t _16;
        uint8_t _8[2];
    } gpr[8];

    /* Do NOT change the order of the GPRs' definitions. */
    struct
    {
        uint32_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
        swaddr_t eip;
        EFLAGS eflags;
    };

} CPU_state;
```

②reg.h *此处添加了新的 eflags

之前在“重组寄存器”一章中已经使用过，这个文件中包含了所有寄存器的定义和访问方法。此时我需要新增一个新的寄存器 EFLAGS，我的做法是先 typedef 一个 EFLAGS 类型，然后如同 eip 一样直接添加在后面一个 struct 中，可以**不与前面共享位置**。注意左图**使用位域**的方式组织 eflags，访问我考虑到其他寄存器是使用 reg_l(index)的 define 的函数访问，但对于这个 eflags 没有必要单独定义访问。直接使用

访问： cpu.eflags.CF 即可访问和修改。（或许不安全，但个人认为此处没必要复杂）

(2):include/cpu/exec

①hepler.h：定义了很多关于指令执行的函数，例如 make_helper_v 的意思即，一个 make_helper 函数，但需要根据 is_operand_size_16 判断是_w 还是_l。此处出现了 print_asm，即打印汇编代码，还有模板类 print_asm_template 会根据译码直接打印。

②template-start.h：提供了读入输入的接口 MEM_R/MEM_W，其实实质是内存读入和输入，但是可以根据 DATA_TYPE 进行变化，使用方式可以为：

```
#define DATA_TYPE 2
#include “template-start.h”
#undef
```

比直接写几种 datatype 方便。可以通过 REG(index)访问/修改寄存器。

(3):include/cpu/decode

decode.h:定义了各种译码的模板，可以根据名字简单判断该模板的译码方式，例如 decode_i2rm_b 即 读入一个立即数和寄存器/内存，并且两者都是一个字节。之后的指令名称取名字时需要根据 decode 模板取名，就可以不用关心译码细节。并且这个 decode 模板也会影响 printf_asm_template，一旦译码确定，汇编语言代码也确定。

(3):src/cpu/exec

①exec.c:所有的 opcode 和 exec 函数均在此处。

make_helper(exec)就是执行函数。

make_group()实际上是有 modRM 的情况，对于一个 opcode，如果存在多个指令的情况，可能需要读入下一位才可以判断这个 opcode 真正的样子，例如本次实验中需要实现的 sub/cmp 都是 0x83，但是根据 ModRM 可以知道他们在 make_group 中的不同位置（共八个）。

helper_fun opcode_table [256]是一个字节的 opcode，本次实现的指令均在此。

正常 i386 中没有 inv 和 nemu_trap，是为了实验能够运转起来，先把所有没有实现的指令写成 inv，然后使用 nemu_trap 来强制结束程序。

②opcode:分成了几个文件夹，其中包含各种指令。所有的指令.h 需要包括在 all-instr.h。

(4):src/monitor/debug

①elf.c:定义了读取 elf 的函数 void load_elf_tables(int argc, char *argv[])，此时已经解析好了 elf，保存在了符号表 symtab 和字符串表 strtab。该文件头文件包含了<elf.h>，在网上可以查看细节，可知 nr_symtab_entry 是 symtab 符号表中符号个数，

Elf32_Sym*为 symtab 的结构，可以通过 symtab->st_name 等方法获取各种信息。

```
typedef struct
{
  Elf32_Word    st_name;        /* Symbol name (string tbl index) */
  Elf32_Addr    st_value;       /* Symbol value */
  Elf32_Word    st_size;        /* Symbol size */
  unsigned char st_info;        /* Symbol type and binding */
  unsigned char st_other;       /* Symbol visibility */
  Elf32_Section st_shndx;       /* Section index */
} Elf32_Sym;
```

通过 strtab+symtab->name 可以获得符号真正的名字。get_var()和 get_stack()函数可以写在这个文件里。

②expr.c:之前实现了很多表达式，这次需要添加 p 变量。

（三）、指令细节

(1):sub 指令与 cmp 指令

```
/* 0x83 */
make_group(group1_sx_v,
    add_rm_imm_v, inv, inv, inv,
    inv, sub_rm_imm_v, inv, cmp_rm_imm_v)
```

sub 指令与 cmp 指令，在 mov-c.txt 中的汇编指令代码都是 0x83，说明需要通过 ModRM 进一步判断，所以按照左图放置指令。

```
83 ec 10                sub     $0x10,%esp    中的 REG 为 101=5
83 f8 01                cmp     $0x1,%eax     中的 REG 为 111=7
```

```
1 #include "cpu/exec/helper.h"
2
3 #define DATA_BYTE 1
4 #include "sub-template.h"
5 #undef DATA_BYTE
6
7 #define DATA_BYTE 2
8 #include "sub-template.h"
9 #undef DATA_BYTE
10
11 #define DATA_BYTE 4
12 #include "sub-template.h"
13 #undef DATA_BYTE
14
15 /* for instruction encoding overloading */
16
17 make_helper_v(sub_rm_imm)
```

```
SUB — Integer Subtraction

Opcode      Instruction      Clocks   Description

2C  ib      SUB AL,imm8      2        Subtract immediate byte from AL
2D  iw      SUB AX,imm16     2        Subtract immediate word from AX
2D  id      SUB EAX,imm32    2        Subtract immediate dword from EAX
80  /5 ib   SUB r/m8,imm8    2/7      Subtract immediate byte from r/m byte
81  /5 iw   SUB r/m16,imm16  2/7      Subtract immediate word from r/m word
81  /5 id   SUB r/m32,imm32  2/7      Subtract immediate dword from r/m dword
83  /5 ib   SUB r/m16,imm8   2/7      Subtract sign-extended immediate byte from r/m word
83  /5 ib   SUB r/m32,imm8   2/7      Subtract sign-extended immediate byte from r/m dword
28  /r      SUB r/m8,r8      2/6      Subtract byte register from r/m byte
29  /r      SUB r/m16,r16    2/6      Subtract word register from r/m word
29  /r      SUB r/m32,r32    2/6      Subtract dword register from r/m dword
2A  /r      SUB r8,r/m8      2/7      Subtract byte register from r/m byte
2B  /r      SUB r16,r/m16    2/7      Subtract word register from r/m word
2B  /r      SUB r32,r/m32    2/7      Subtract dword register from r/m dword
```

```
1 #include "cpu/exec/template-start.h"
2 #include "cpu/flag.h"
3 #define instr sub
4
5 static void do_execute () {
6     DATA_TYPE result = op_dest->val - op_src->val;
7     OPERAND_W(op_dest, result);
8     /* TODO: Update EFLAGS. */
9     update_ZF(result);
10    update_SF(result,4);
11    update_PF(result&0xff);
12
13    /*TODO: Update CF & OF*/
14    update_CF(op_dest->val < op_src->val);
15    update_OF(result);
16    print_asm_template2();
17 }
18
19 make_instr_helper(rm_imm)
20
21 #include "cpu/exec/template-end.h"
```

```
1 #ifndef __SUB_H__
2 #define __SUB_H__
3
4 make_helper(sub_rm_imm_b);
5 make_helper(sub_rm_imm_v);
6
7 #endif
```

sub 指令

先查阅 i386 的指令解释，可以发现 83 为寄存器和立即数的操作，此处可以使用 rm_imm 的译码（其实也可以仿照 mul 中使用 rm 的译码），并且仿照 arith 里另一个运算 mul 的方式，组织成 sub.h/sub-template.h/sub.c 的方式，代码分别如图。

sub 指令需要更新 ZF/SF/PF/CF/OF，关于 flags 后面单独叙述。

```
1 #ifndef __CMP_H__
2 #define __CMP_H__
3
4 make_helper(cmp_rm_imm_b);
5 make_helper(cmp_rm_imm_v);
6
7 #endif
```

cmp.h

cmp 指令

内容和 sub 指令极其相似，只是这里不需要将结果存储起来，更新 flag 如图。

```
1 #include "cpu/exec/template-start.h"
2 #include "cpu/flag.h"
3
4 #define instr cmp
5
6 static void do_execute () {
7     DATA_TYPE result = op_dest->val - op_src->val;
8     /* TODO: Update EFLAGS. */
9     update_ZF(result);
10    update_SF(result,4);
11    update_PF(result&0xff);
12
13    /*TODO: Update CF & OF*/
14    update_CF(op_dest->val < op_src->val);
15    update_OF(result);
16
17    print_asm_template2();
18 }
19
20 make_instr_helper(rm2r)
21 make_instr_helper(rm_imm)
22
23
24 #include "cpu/exec/template-end.h"
```

```
1 #include "cpu/exec/helper.h"
2
3 #define DATA_BYTE 1
4 #include "cmp-template.h"
5 #undef DATA_BYTE
6
7 #define DATA_BYTE 2
8 #include "cmp-template.h"
9 #undef DATA_BYTE
10
11 #define DATA_BYTE 4
12 #include "cmp-template.h"
13 #undef DATA_BYTE
14
15 /* for instruction encoding overloading */
16
17 make_helper_v(cmp_rm_imm)
```

cmp-template.h

这里需要注意：do_execute 实际上是模板类，_v 会自动识别现在的操作数是几个字节，并根据字节调用正确的 template-start.h，所以不需要担心其他情况，因为已经写好了模板。这里我并没有实际实现 OF，因为该实验中实际只用了 ZF/CF。

使用 print_asm_template2 可以输出指令+操作数 1+操作数 2

(2):push 指令

```
30 make_helper(push) {
31     int index = instr_fetch(eip , 1) - 0x50;
32     REG(R_ESP) = REG(R_ESP) - 4;
33     MEM_W(REG(R_ESP), reg_l(index));
34     print_asm("push %%%s",regsl[index]);
35     return 1;
36 }
```

接下来的 push/call/je 我都放进了 misc.c 中，因为其实这几个指令的模板实现意义不大，本身没有几个指令集，所以放入杂项与 nop/lea 并列很合理。

push 指令：misc.h 文件中存放好 make_helper(push) 头文件定义。.c 文件中写代码如上。push 指令的操作如下图

```
118 /* 0x54 */  inv, push, inv, inv,
```

55　　push　　%ebp

```
ESP ← ESP - 4;
(SS:ESP) ← (SOURCE); (* dword assignment *)
```

根据 i386 的指南可知 push 操作即先栈顶更新后把寄存器值放入栈中。由于 push 指令是 0x50+R/M，则需要通过 instr_fetch(eip,1) 先获取原先的指令数减去 0x50 得到当前压栈的寄存器 index，然后 esp 更新（-4），将 reg(index)放入 esp 所指向的内存中。

(3):call 指令

```
make_helper(call) {
    //push eip
    REG(R_ESP) = REG(R_ESP) - 4;
    MEM_W(REG(R_ESP), eip);
    uint32_t rel32 = MEM_R(eip + 1);
    cpu.eip = cpu.eip + rel32;
    print_asm("call %x\n",cpu.eip + 5);
    return 5;
}
```

同样是在 misc.c 中定义 call 指令，先在 i386 中查看 call 指令的定义。

```
Push(EIP);
EIP ← EIP + rel32;
```

e8 00 00 00 00　　call　　100012 <main>

```
/* 0xe8 */  call, inv, inv, inv,
```

根据操作，先模拟一遍 push(eip)，类似刚才叙述的 PUSH。然后需要读取一个字节的偏移量 rel32，加上 eip 得到新的跳转地址。返回值为读入的字节，有 5 个字节。

注意： *这里不能使用 eip 这个变量，这个变量是每次运行结束程序后更新的 eip，而不是实时更新的 cpu 中的存放的 eip 值。*

(4):test 指令

```
void init_eflags(){
    cpu.eflags.CF = 0;
    cpu.eflags.PF = 0;
    cpu.eflags.AF = 0;
    cpu.eflags.ZF = 0;
    cpu.eflags.SF = 0;
    cpu.eflags.TF = 0;
    cpu.eflags.IF = 0;
    cpu.eflags.DF = 0;
    cpu.eflags.OF = 0;
}
```

首先先需要组织结构体中的 eflags，见上面的 reg.h 文件。在 restart()需要先初始化 eflags:使用 init_eflags()，添加到 restart()函数中。

test 指令也同 sub 类似，组织了 test.h/test.c/test-template.h 三个文件。

```
1 #include "cpu/exec/template-start.h"
2 #include "cpu/flag.h"
3
4 #define instr test
5
6 static void do_execute () {
7     DATA_TYPE result = op_src->val & op_dest->val;
8     //printf("src: %x, src2 : %x\n ,dest : %x\n",op_sr
9     /* TODO: Update EFLAGS. */
10    cpu.eflags.CF = 0;
11    cpu.eflags.OF = 0;
12    update_ZF(result);
13    update_SF(result,4);
14    update_PF(result&0xff);
15    print_asm_template2();
16 }
17
18 make_instr_helper(rm2r)
19
20 #include "cpu/exec/template-end.h"
```

```
1 #ifndef __TEST_H__
2 #define __TEST_H__
3
4 make_helper(test_rm2r_b);
5 make_helper(test_rm2r_v);
6
7
8 #endif
```

```
1 #include "cpu/exec/helper.h"
2
3 #define DATA_BYTE 1
4 #include "test-template.h"
5 #undef DATA_BYTE
6
7 #define DATA_BYTE 2
8 #include "test-template.h"
9 #undef DATA_BYTE
10
11 #define DATA_BYTE 4
12 #include "test-template.h"
13 #undef DATA_BYTE
14
15 /* for instruction encoding overloading */
16
17 make_helper_v(test_rm2r)
```

其中 test 指令的 i386 解释为。将两个操作数进行 AND 操作，用来更新符号位。CF 和 OF 都置零，不需要保存计算出来的 result 结果。

85 c0　test　　%eax,%eax 中 85 填充 test 指令。

```
130 /* 0x84 */  inv, test_rm2r_v, inv, inv,
```

(5):je 指令

```
make_helper(je){
    if(cpu.eflags.ZF==1)
    {
        signed char rel = swaddr_read(cpu.eip + 1, 1);
        cpu.eip += rel;
        print_asm("je %x",cpu.eip + 2);
    }
    return 2;
}
```

je 的指令，在 i386 中解释为如果 ZF 为 1 则跳转后面操作数的偏移量。故翻译成代码即左图。同样需要注意修改的 eip 一定是 cpu.eip 不是 eip 本身。由于后面的立即数只有一个字节，返回值为 2.

```
#ifndef __MISC_H__
#define __MISC_H__

make_helper(nop);
make_helper(int3);
make_helper(lea);
make_helper(push);
make_helper(call);
make_helper(je);

#endif
```

```
/* 0x74 */  je, inv, inv, inv,
```

注意： *这里需要有符号单字节扩展，可以存为 signed char 自动扩展成 uint32_t，实现有符号数的转换。*

74 06　　je　　10006a <main+0x58>

(6):EFLAGS

```
#ifndef __FLAG_H__
#define __FLAG_H__

#include "cpu/reg.h"

/* CF */
static inline void update_CF(int result){
    if(result == 1) cpu.eflags.CF = 1;
    else cpu.eflags.CF = 0;
}

/* PF */
static inline void update_PF(uint8_t result){
    int t1 = 0x55;
    int t2 = 0x33;
    int t3 = 0x0f;
    result = result&0xff;
    int count = (result&t1) + ((result>>1)&t1);
    count = (count&t2) + ((count>>2)&t2);
    count = (count + (count>>4))&t3;
    cpu.eflags.PF = !(count%2);
}

/* ZF */
static inline void update_ZF(uint32_t result){
    if(result == 0) cpu.eflags.ZF = 1;
    else cpu.eflags.ZF=0;
}

/* SF */
static inline void update_SF(uint32_t result, int width){
    result = result >> (width*8-1);
    cpu.eflags.SF = result;
}

/* OF */
static inline void update_OF(uint32_t result){

}

#endif
```

由于每次更新 eflags 的操作很相似，我定义了一个 flag.h 在 include/cpu/中，需要用的时候直接包含进来即可。操作函数模板为 update_xx()

①CF/OF:对于 add/sub/cmp/test 都有不同的更新方式，所以可以取决于指令的更新方式，直接传入更新值 true or false。本实验没用到 OF（没有实现）。

②PF(奇偶校验)：这个实现方法采用了 lab1-bits.c 的做法，统计低八位的 1 的个数，然后判断是否为偶数，若为偶数，PF=1；若为奇数，PF=0。

③ZF(零判断)：判断结果是否为 0，若为 0，则为 1；否则为 0。

④SF:判断高位符号位。其实可以使用宏定义 MSB().

(7):jmp/jbe(均定义在 misc.c 中)

```
make_helper(jmp_b){
    signed char rel = swaddr_read(cpu.eip + 1, 1);
    cpu.eip += rel;
    print_asm("jmp %x",cpu.eip + 2);
    return 2;
}

make_helper(jbe_b){
    if(cpu.eflags.CF == 1 || cpu.eflags.ZF == 1)
    {
        signed char rel = swaddr_read(cpu.eip + 1, 1);
        cpu.eip += rel;
        print_asm("jbe %x",cpu.eip + 2);
    }
    return 2;
}
```

①jmp:

```
EB  cb    JMP rel8        7+m             Jump short
EIP ← EIP + rel8/16/32;
```

此处是单字节扩展跳转，扩展方式同 je。只需要更新 eip 即可。返回两个字节。

②jbe:判断条件为 CF/ZF==1，剩余同 jmp short 相同。

(8):add/addl

add.c

```
#include "cpu/exec/helper.h"

#define DATA_BYTE 1
#include "add-template.h"
#undef DATA_BYTE

#define DATA_BYTE 2
#include "add-template.h"
#undef DATA_BYTE

#define DATA_BYTE 4
#include "add-template.h"
#undef DATA_BYTE

/* for instruction encoding overloading */

make_helper_v(add_r2rm)
make_helper_v(add_rm_imm)
```

add.h

```
#ifndef __ADD_H__
#define __ADD_H__

make_helper(add_r2rm_b);
make_helper(add_r2rm_v);
make_helper(add_rm_imm_v);

#endif
```

add-template.h

```
#include "cpu/exec/template-start.h"
#include "cpu/flag.h"
#define instr add

static void do_execute () {
    DATA_TYPE result = op_dest->val + op_src->val;
    OPERAND_W(op_dest, result);
    /* TODO: Update EFLAGS. */
    update_ZF(result);
    update_SF(result,4);
    update_PF(result&0xff);

    /*TODO: Update CF & OF*/
    update_CF(0);
    update_OF(result);
    print_asm_template2();
}

make_instr_helper(r2rm)
make_instr_helper(rm_imm)

#include "cpu/exec/template-end.h"
```

add 指令：

```
10009b:   83 45 fc 01             addl
```

原理同 sub 几乎一样，add.c 中定义了类似 sub_rm_imm 的 add_rm_imm，也是指令 83，放在 make_group 中。由于 45：中 REG 为 000，放在第 0 位。

对于指令 01：

```
01 /r     ADD r/m16,r16
01 /r     ADD r/m32,r32
```

，使用 r2rm 译码。由于 add 对于 CF 和 OF 的影响实验中未使用，简单置零。

(9):leave/ret

leave:

```
C9      LEAVE       4       Set ESP to EBP, then pop EBP
```

ret:

```
C3        RET         10+m          Return (near) to caller
```

```
make_helper(leave){
    REG(R_ESP) = REG(R_EBP);
    REG(R_EBP) = MEM_R(R_EBP);
    REG(R_ESP) = REG(R_ESP) + 4;
    print_asm("leave");
    return 1;
}

make_helper(ret){
    cpu.eip = MEM_R(REG(R_ESP)) + 4;
    REG(R_ESP) = REG(R_ESP) + 4;
    print_asm("ret\n");
    return 1;
}
```

①leave:先更新 esp 指向 ebp，即 esp = ebp;再更新 ebp 为 prev_ebp，即 ebp = mem(ebp)；再更新 esp+4。相当于 ebp = pop()。

②ret:先更新 eip 为返回 call 前的地址(+偏移量)，再更新 esp+4，相当于 eip = pop();

（四）、打印变量/栈帧

(1):打印变量 p 变量名

```
uint32_t get_var(char* str, int* ans)
{
    int i;
    for(i=0;i<nr_symtab_entry;i++){
        Elf32_Sym* sym = symtab+i;
        if(sym->st_info == 17 && strcmp(strtab+sym->st_name,str))
        {
            *ans = 1;
            return sym->st_value;
        }
    }
    *ans = 0;
    return 0;
}
```

```
else if (tokens[p].type == TK_VAR)
{
    int ans,result = get_var(tokens[p].str,&ans);
    if(ans==0) assert(0);
    return result;
}
```

左图为 elf.c 中根据变量名获取变量值的函数。右图为 expr.c 中识别出变量类型后获取变量值的情况。

首先 expr.c 需要添加 token_type: TK_VAR，识别方式最后，为 c 语言变量规则{"[_a-zA-Z][a-zA-Z_0-9]*", TK_VAR}

然后添加变量识别情况（左图），直接根据 tokens[p].str 存的变量字符串去与符号表中每个符号对比。

```
//test1:print symtab
void pa()
{
    int i;
    for(i=0;i<nr_symtab_entry;i++)
    {
        Elf32_Sym* sym = symtab+i;
        printf("name:%s\tvalue:0x%x\tinfo:%x\tsize:0x%x\n",strtab+sym->st_name,sym->st_value,sym->st_info,sym->st_size);
    }
}
```

我先打印了一遍 symtab 中的所有信息，查看 info==0x11 时即 Type = OBJECT。并且要表示符号的名字，需要在 strtab 中找到真正的字符串名字 strtab+symtab->st_name 即真实名字，与 str 对比就能确定是否为选取符号。

我还设置了 ans 用于判断是否真正成功取值，如果没有找到变量，assert(0)。

(2):打印栈帧 bt

```
//get the whole stack
void get_stack()
{
    uint32_t ebp = cpu.ebp,i;
    char* str = (char*)malloc(sizeof(char)*100);

    func_name(cpu.eip,str);
    printf("#0\t0x%08x in %s ()\n",cpu.eip,str);
    for(i=1;;i++)
    {
        if(func_name(swaddr_read(ebp+4,4),str))
            printf("#%d\t0x%08x in %s ()\n",i,swaddr_read(ebp+4,4),str);
        if(ebp==0) break;
        else ebp = swaddr_read(ebp,4);
    }
}
```

```
static int cmd_bt(char *args)
{
    get_stack();
    return 0;
}
```

首先在 ui.c 中添加左图所示的 cmd_bt，只需要使用 get_stack()函数打印出栈帧即可。

get_stack()函数定义在 elf.c 中，用 str 开空间保存函数名称。打印栈帧的过程分解为：①查找当前地址，判断当前所在函数名，打印②寻找返回函数的地址

```
//get the function name in symtab
bool func_name(uint32_t addr, char* str)
{
    int i;
    for(i=0;i<nr_symtab_entry;i++)
    {
        Elf32_Sym* sym = symtab+i;
        if(sym->st_info==0x12&&(sym->st_value < addr && sym->st_value+sym->st_size>=addr))
        {sprintf(str,"%s",strtab+sym->st_name);return true;}
    }
    return false;
}
```

左图为根据当前地址查找所在函数名称的函数，遍历 symtab，查找满足 info==12 和地址在内部的 function，最后保存 function name

对于打印内容，先通过 eip 打印当前函数名，读取 ebp 可以返回上个函数的 ebp，存放的地址在 ebp+4 中，同理可以读取出 function name。

三、实验结果

(1):add.c

```
chty627@ubuntu:~/ics2015$ make run
+ cc nemu/src/cpu/exec/arith/add.c
+ cc nemu/src/monitor/debug/elf.c
+ ld obj/nemu/nemu
+ cc testcase/src/add.c
+ ld obj/testcase/add
objcopy -S -O binary obj/testcase/add entry
obj/nemu/nemu obj/testcase/add
Welcome to NEMU!
The executable is obj/testcase/add.
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x001000b7
```

(2):mov-c.c

```
+ cc testcase/src/mov-c.c
+ ld obj/testcase/mov-c
objcopy -S -O binary obj/testcase/mov-c entry
obj/nemu/nemu obj/testcase/mov-c
Welcome to NEMU!
The executable is obj/testcase/mov-c.
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x001000cf
```

(3):p testdata / bt

```
(nemu) c

Hit breakpoint at eip = 0x00100023
(nemu) p testdata
1056800
(nemu) p *(testdata+4)
1
(nemu) bt
#0      0x00100024 in add ()
#1      0x0010006a in main ()
```

```
int add(int a, int b) {
    int c = a + b;
    set_bp();
    return c;
}
```

add 函数中设置断点

四、思考题

(1)大端机器

> 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?

答：此时需要注意大端机器与小端的存放方式不同。当读入超过 1 个字节的立即数/内存空间时，都会需要注意，起始位置，结束位置，该变量/寄存器的总长度。首先需要知道 DATA_TYPE 的大小，如果为一个字节可以如同以前一样读入，如果超过一个字节，2/4 的时候，需要遍历读入 DATA_TYPE 字节，然后反过来重组为正确的数值和地址。

(2)为什么目前让用户程序从 `main` 函数返回就会发生错误？这个错误具体是怎么发生的

```
00100000 <start>:
  100000:   bd 00 00 00 00          mov    $0x0,%ebp
  100005:   bc 00 00 00 08          mov    $0x8000000,%esp
  10000a:   83 ec 10                sub    $0x10,%esp
  10000d:   e8 17 00 00 00          call   100029 <main>
```

答：查看 add.c 的汇编代码可知 add 中是 start 函数分配了 esp/ebp 栈空间，调用 main 函数。如果 main 函数返回则返回为 start 函数，但之后没有任何指令。程序无法自然停止。

```
(nemu) c
invalid opcode(eip = 0x00000007): 00 00 00 00 00 00 00 00 ...
```

(3)消失的符号

> 消失的符号
>
> 我们在 `add.c` 中定义了宏 `NR_DATA`，同时也在 `add()` 函数中定义了局部变量 `c` 和形参 `a`，`b`，但你会发现在符号表中找不到和它们对应的表项，为什么会这样？思考一下，什么才算是一个符号(symbol)？

形参不能称之为符号，在编译过程中，只有强符号会分配空间和初值，弱符号保存符号信息，没有初值。而形参不是强符号也不是弱符号。

五、实验感受

本次实验重点在于对于 cpu 执行指令过程的理解，当理解到了取指、译码、执行三个步骤之后，需要对框架代码有很深刻的了解，才能理解清楚每个函数、宏定义、逻辑关系。

我也更加深刻的认识到了汇编语言对于理解程序执行过程的作用，我可以查看汇编语言中前面机器码的部分，然后根据指令名称查找到 i386 中对应的指令解释，根据指令的机器码查看相关的定义内容。译码部分已经有模板可以实现，实现指令部分的过程不复杂，只需要把伪代码写成 c 语言即可，困难的是 debug 中各种细节的小错误，例如 CF 未更新、取寄存器值得方法、更新寄存器的方法等等。所以需要用上次使用的 gdb 方法来调试，每次做完就需要检测，满足 KISS 原则。

难点在于对于框架代码的理解，也是花了很多时间看代码，才真的意识到了宏定义、函数引用的逻辑关系。这个本领在以后肯定很实用，因为需要阅读别人的代码。每次接触新的代码的时候都需要根据它的逻辑理清楚，然后自己试验着跑一遍程序来理解。

期待下次与你相见。