



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

计算机系统设计实验报告

PA1 - 开天辟地的篇章：最简单的计算机

邵琦

年级：2020 级

专业：计算机科学与技术

指导教师：卢冶

2023 年 3 月 21 日

目录

一、 实验目的	1
二、 实验内容	1
三、 阶段一	1
(一) 实现正确的寄存器结构体	1
(二) 实现单步执行, 打印寄存器, 扫描内存	4
1. 单步执行	5
2. 打印寄存器	5
3. 扫描内存	6
四、 阶段二	7
(一) 词法分析	7
(二) 递归求值	10
五、 阶段三	17
(一) 实现监视点池的管理	17
(二) 实现监视点	20
六、 遇到的问题以及解决办法	22
(一) git log 丢失问题	22
七、 问题	23
(一) 究竟要执行多久?	23
(二) 谁来指示程序的结束?	23
(三) 为什么要使用 static?	24
(四) 一点也不能长?	25
(五) 随心所欲的断点	25
(六) NEMU 的前世今生	26
八、 必答题	26
(一) i386 手册	26
1. EFLAGS	26
2. ModR/M	27
3. mov	28
(二) shell 命令	29
(三) gcc	30

一、 实验目的

- 1) 熟悉 32 位 GNU/Linux 平台。
- 2) 熟悉 git 分布式版本控制系统的使用。
- 3) 熟悉 nemu 计算机系统的原理及程序运行的流程。
- 4) 熟悉 GNU/Linux 中 vim 等各种工具的使用。
- 5) 实现 nemu 的寄存器结构和简易调试器功能。

二、 实验内容

PA1 的主要内容为实现简易的调试器，大致包含以下三个步骤：

- 1) 实现 nemu 系统的寄存器结构，使系统可以正常运行。实现调试器的基本功能，包含单步执行和打印寄存器状态。
- 2) 实现调试器的表达式求值功能。
- 3) 实现调试器有关监视点的功能。

三、 阶段一

(一) 实现正确的寄存器结构体

寄存器结构：除程序计数器 eip 外，32 位、16 位、8 位寄存器各 8 个，其结构如下图所示：

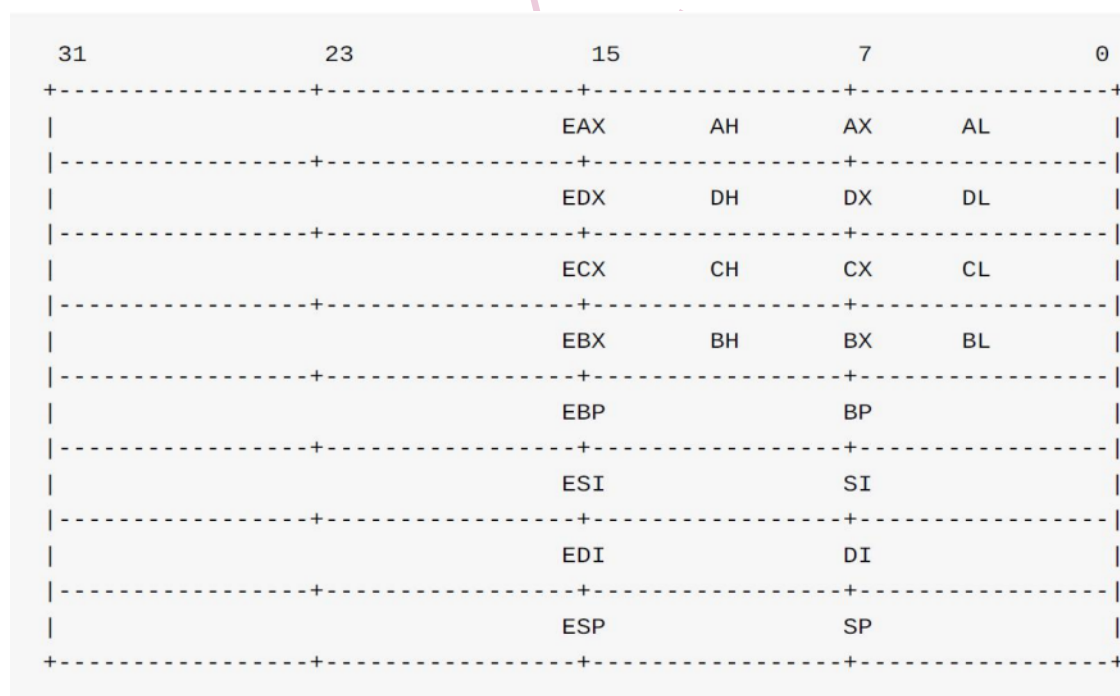


图 1: 寄存器结构

其中，EAX,EDX,ECX,EBX,EBP,ESI,EDI,ESP 是 32 位寄存器；
AX,DX,CX,BX,BP,SI,DI,SP 是 16 位寄存器；

AL,DL,CL,BL,AH,DH,CH,BH 是 8 位寄存。但它们在物理上并不是相互独立的, 例如 EAX 的低 16 位是 AX, 而 AX 又分成 AH 和 AL。这样的结构有时候在处理不同长度的数据时能提供一些便利。

我们可以通过相关文件看到相关寄存器的相关变量、初始化以及相关函数:

相关变量以及相关函数

```
1 //variable      init in reg.c
2 extern const char* regsl [];
3 extern const char* regsw [];
4 extern const char* regsb [];
5
6 #define reg_l(index) (cpu.gpr[check_reg_index(index)]._32)    //return regsl[
7                          i]
8 #define reg_w(index) (cpu.gpr[check_reg_index(index)]._16)    //return regsw[
9                          i]
10 #define reg_b(index) (cpu.gpr[check_reg_index(index) & 0x3]._8[index >> 2])
11                          //return regsb[i]
```

初始化

```
1 CPU_state cpu;
2
3 const char *regsl [] = {"eax", "ecx", "edx", "ebx", "esp", "ebp", "esi", "edi"
4                          };
5 const char *regsw [] = {"ax", "cx", "dx", "bx", "sp", "bp", "si", "di"};
6 const char *regsb [] = {"al", "cl", "dl", "bl", "ah", "ch", "dh", "bh"};
```

在 reg.c 文件中的 reg_test() 函数中我们检验 32、16、8 位寄存器满足图 1 所示的寄存器结构, 通过与操作和移位运算, 保证寄存器的正确性。确保可直接通过寄存器名访问 32 位寄存器的内容。

reg_test() 函数

```
1 void reg_test() {
2     srand(time(0));
3     uint32_t sample[8];
4     uint32_t eip_sample = rand();
5     cpu.eip = eip_sample;
6
7     int i;
8     for (i = R_EAX; i <= R_EDI; i++) {
9         sample[i] = rand();
10        reg_l(i) = sample[i];
11        assert(reg_w(i) == (sample[i] & 0xffff));
12    } //and operation & shift operation
13    //ensure reg correct
14}
```

```
15  assert(sample[R_EAX] == cpu.eax);
16  assert(sample[R_ECX] == cpu.ecx);
17  assert(sample[R_EDX] == cpu.edx);
18  assert(sample[R_EBX] == cpu.ebx);
19  assert(sample[R_ESP] == cpu.esp);
20  assert(sample[R_EBP] == cpu.ebp);
21  assert(sample[R_ESI] == cpu.esi);
22  assert(sample[R_EDI] == cpu.edi);
23
24  assert(eip_sample == cpu.eip);
```

由测试函数 `reg_test()` 可知, 需实现 `gpr[8]` 的 Union 结构, 并且要求 `eax,ecx,edx` 等 32 位寄存器与 `gpr[8]` 开始于同一内存首地址。具体实现如下:

实现正确的寄存器结构体

```
1  typedef struct {
2  union{
3      union{
4          uint32_t _32;
5          uint16_t _16;
6          uint8_t _8[2];
7      } gpr[8];
8      struct
9      {
10
11          /* Do NOT change the order of the GPRs' definitions. */
12
13          /* In NEMU, rtlreg_t is exactly uint32_t. This makes RTL instructions
14           * in PA2 able to directly access these registers.
15           */
16          rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
17      };
18  };
19  vaddr_t eip;
20
21 } CPU_state;
```

(二) 实现单步执行, 打印寄存器, 扫描内存

命令	格式	使用举例	说明
帮助(1)	help	help	打印命令的帮助信息
继续运行(1)	c	c	继续运行被暂停的程序
退出(1)	q	q	退出 NEMU
单步执行	si [N]	si 10	让程序单步执行 N 条指令后暂停执行, 当 N 没有给出时, 缺省为 1
打印程序状态	info SUBCMD	info r info w	打印寄存器状态 打印监视点信息
表达式求值	p EXPR	p \$eax + 1	求出表达式 EXPR 的值, EXPR 支持的运算请见调试中的表达式求值小节
扫描内存(2)	x N EXPR	x 10 \$esp	求出表达式 EXPR 的值, 将结果作为起始内存地址, 以十六进制形式输出连续的 N 个 4 字节
设置监视点	w EXPR	w *0x2000	当表达式 EXPR 的值发生变化时, 暂停程序执行
删除监视点	d N	d 2	删除序号为 N 的监视点

图 2: 简易调试器

我们根据输入的命令, 在 `cmd_table` 结构中, 调用相关的处理函数。

```

cmd_table

1 static struct {
2     char *name;
3     char *description;
4     int (*handler) (char *);
5 } cmd_table [] = {
6     { "help", "Display informations about all supported commands", cmd_help },
7     { "c", "Continue the execution of the program", cmd_c },
8     { "q", "Exit NEMU", cmd_q },
9
10    /* TODO: Add more commands */
11
12    { "si", "args:[N]; execute [N] instructions step by step", cmd_si},
13    { "info", "args:r/w; print information about registers or watchpoint",
14        cmd_info},
15    { "x", "x [N] [EXPR]; scan the memory", cmd_x},
16    { "p", "expr", cmd_p},
17    { "w", "set the watchpoint", cmd_w},
18    { "d", "delete the watchpoint", cmd_d}

```

```
18 };
```

1. 单步执行

单步执行需要对输入的参数进行分类讨论，如无参数，默认执行 1 步，若参数为负，则返回错误参数提示。参数正确时，调用 `cpu_exec` 函数执行对应的步数。

单步执行

```
1 static int cmd_si(char *args)
2 {
3     uint64_t N=0;
4     if(args==NULL)
5     {
6         N=1;
7     }
8     else
9     {
10        int temp=sscanf(args,"%llu",&N);
11        if(temp<=0)
12        {
13            printf("args error in cmd_si\n");
14            return 0;
15        }
16    }
17    cpu_exec(N);
18    return 0;
19 }
```

2. 打印寄存器

打印寄存器也需要检查参数的正确性，若参数错误直接返回报错信息。当参数为“r”时，我们遍历 `regsl`, `regsw`, `regsb`，分别使用 `reg_l`, `reg_w`, `reg_b` 函数进行输出。当参数为“w”时，调用 `print_wp` 函数（在之后实现），输出监视点。

打印寄存器

```
1 static int cmd_info(char *args)
2 {
3     char s;
4     if(args==NULL)
5     {
6         printf("args error in cmd_info : miss args\n");
7         return 0;
8     }
9     int temp = sscanf(args,"%c",&s);
10    if(temp<=0)
11    {
12        //fail
13        printf("args error in cmd_info\n");
```

```

14     return 0;
15 }
16 if(s=='r')
17 {
18     //print reg
19     int i;
20     //32bit
21     for(i=0;i<8;i++)
22     {
23         printf("%s    0x%x\n",regsl[i],reg_l(i));
24     }
25     printf("eip    0x%x\n",cpu.eip);
26     //16bit
27     for(i=0;i<8;i++)
28     {
29         printf("%s    0x%x\n",regsw[i],reg_w(i));
30     }
31     //8bit
32     for(int i=0;i<8;i++)
33     {
34         printf("%s    0x%x\n",regsb[i],reg_b(i));
35     }
36     return 0;
37 }
38 if(s=='w')
39 {
40     //print monitor information
41     print_wp();
42     return 0;
43 }
44 printf("args error in cmd_info\n");
45 return 0;
46 }

```

3. 扫描内存

扫描内存也需要检查参数的正确性，若参数错误直接返回报错信息。我们读入长度和地址后，使用 `vaddr_read` 函数进行地址存储的数值读入，对每 8 位进行分割，倒序输出。

扫描内存

```

1 static int cmd_x(char *args)
2 {
3     //get data
4     int addr;
5     int len;
6     sscanf(args,"%d 0x%x",&len,&addr);
7     int i;
8     printf("memory:\n");

```



```
9 //memory scanning
10 for(i=0;i<len;i++)
11 {
12     printf("0x%x ",addr);
13     uint32_t val=vaddr_read(addr,4);
14     uint8_t *data=(uint8_t *)&val;
15     printf("0x");
16     for(int j=3;j>=0;j--)
17     {
18         printf("%02x",data[j]);
19     }
20     printf("\n");
21     addr+=4;
22 }
23 return 0;
24 }
```

四、 阶段二

(一) 词法分析

我们先搞定 token 的定义与声明，按正则表达式的方式去描述：

token 的定义与声明

```
1 enum {
2     TK_NOTYPE = 256,
3     TK_EQ,
4     TK_NUMBER,
5     TK_HEX,
6     TK_REG,
7     TK_NEQ,
8     TK_AND,
9     TK_OR,
10    TK_NEGATIVE,
11    TK_DEREF,
12    TK_GEQ,
13    TK_LEQ,
14    TK_L,
15    TK_R
16
17    /* TODO: Add more token types */
18
19 };
20
21 static struct rule {
22     char *regex;
23     int token_type;
```

```

24 } rules [] = {
25
26     /* TODO: Add more rules.
27      * Pay attention to the precedence level of different rules.
28      */
29
30     {" +", TK_NOTYPE},    // spaces
31     {"\\+", '+'},        // plus
32     {"==", TK_EQ},        // equal
33     {"0x[0-9A-Fa-f]+", TK_HEX},
34     {"0|[1-9][0-9]*", TK_NUMBER},
35     {"\\$(eax|ecx|edx|ebx|esp|ebp|esi|edi|eip|ax|cx|dx|bx|sp|bp|si|di|al|cl|dl|
        bl|ah|ch|dh|bh)", TK_REG},
36     {"!=", TK_NEQ},
37     {"&&", TK_AND},
38     {"\\|\\|", TK_OR},
39     {">=", TK_GEQ},
40     {"<=", TK_LEQ},
41     {"<<", TK_L},
42     {">>", TK_R},
43     {"!", '!'},
44     {"<", '<'},
45     {">", '>'},
46     {"\\|", '|'},
47     {"\\^", '^'},
48     {"&", '&'},
49     {"\\-", '-'},
50     {"\\*", '*'},
51     {"/", '/'},
52     {"\\(", '('},
53     {"\\)", ')'}
54 };

```

之后，我们在 `make_token` 函数中，实现将识别出的 token 记录到 `tokens` 数组中。

我们对表达式中每个符号进行遍历，使用 `rules` 进行 token 的识别，若为空格，则进行下一个符号的词法分析，若为其他，在 `tokens` 数组中记录其类型，当为十进制整数、十六进制整数、寄存器时，还需对 `tokens` 数组中的 `str` 进行赋值，十六进制整数需去除 `0x`，寄存器需去除 `$`。并对 token 数量进行记录，如没有匹配，则返回匹配错误信息。

make_token 函数

```

1 static bool make_token(char *e) {
2     int position = 0; // current processing location
3     int i;
4     regmatch_t pmatch;
5
6     nr_token = 0;
7
8     while (e[position] != '\0') {

```

```

9      /* Try all rules one by one. */
10     for (i = 0; i < NR_REGEX; i++) {
11         if (regexec(&re[i], e + position, 1, &pmatch, 0) == 0 && pmatch.rm_so
12             == 0) {
13             char *substr_start = e + position;
14             int substr_len = pmatch.rm_eo;
15
16             Log("match rules[%d] = \"%s\" at position %d with len %d: %.*s",
17                 i, rules[i].regex, position, substr_len, substr_len, substr_start
18                 );
19             position += substr_len;
20
21             /* TODO: Now a new token is recognized with rules[i]. Add codes
22              * to record the token in the array 'tokens'. For certain types
23              * of tokens, some extra actions should be performed.
24              */
25
26             if (substr_len > 32)
27             {
28                 assert(0);
29             }
30             if (rules[i].token_type == TK_NOTYPE)
31             {
32                 break;
33             }
34             else
35             {
36                 tokens[nr_token].type = rules[i].token_type;
37                 switch (rules[i].token_type)
38                 {
39                     case TK_NUMBER:
40                         strncpy(tokens[nr_token].str, substr_start, substr_len);
41                         *(tokens[nr_token].str + substr_len) = '\0';
42                         break;
43                     case TK_REG:
44                         strncpy(tokens[nr_token].str, substr_start + 1, substr_len - 1);
45                         *(tokens[nr_token].str + substr_len - 1) = '\0';
46                         break;
47                     case TK_HEX:
48                         strncpy(tokens[nr_token].str, substr_start + 2, substr_len - 2);
49                         *(tokens[nr_token].str + substr_len - 2) = '\0';
50                         break;
51                 }
52                 nr_token++;
53                 break;
54             }
55         }
56     }
57 }

```

```
55
56     if (i == NR_REGEX) {
57         printf("no match at position %d\n%s\n%*.s^\n", position, e, position, "
58             ");
59         return false;
60     }
61
62     return true;
63 }
```

(二) 递归求值

为了实现递归求值，我们需要进行一些前置工作。

首先是 `check_parentheses` 函数，我们使用其来判断表达式是否被一对匹配的括号包围着，同时检查表达式的左右括号是否匹配。若符号要求函数返回 `true`，否则函数返回 `false`。

左右括号的匹配可使用计数器来实现。我们使用计数器 `count` 来统计“未被匹配的左括号”的数目。在确定表达式的 `p`、`q` 位置为左、右括号后，遍历 `[p-1,q-1]` 的表达式，如果遍历期间 `count<0` 则说明出现多余的“`)`”，如果遍历结束时 `count>0` 则说明出现多余的“`(`”，则匹配失败，返回 `false`。只有当遍历结束后 `count==0` 才说明匹配成功，返回 `true`。

check_parentheses 函数

```
1 bool check_parentheses(int p, int q)
2 {
3     if(p>=q)
4     {
5         printf("error:p>=q in check_parentheses\n");
6         return 0;
7     }
8     if(tokens[p].type!='(' || tokens[q].type!=')')
9     {
10        return 0;
11    }
12    int count=0;
13    for(int i=p+1;i<q;i++)
14    {
15        if(tokens[i].type=='(')
16        {
17            count++;
18        }
19        if(tokens[i].type==')')
20        {
21            if(count!=0)
22            {
23                count--;
24            }
25            else
```

```

26     {
27         return 0;
28     }
29 }
30 }
31 if(count==0)
32 {
33     return 1;
34 }
35 else
36 {
37     return 0;
38 }
39 }

```

之后我们定义 `priority` 函数，通过输入的 `tokens` 数组位置的符号类型，从而返回相应的优先级。优先级越高，返回值越大。

priority 函数

```

1 //operator priority
2 int priority(int i)
3 {
4     if(tokens[i].type==TK_NEGATIVE||tokens[i].type==TK_DEREF||tokens[i].type=='!' )
5     {
6         return 7;
7     }
8     else if(tokens[i].type=='*' || tokens[i].type=='/')
9     {
10        return 6;
11    }
12    else if(tokens[i].type=='+' || tokens[i].type=='-')
13    {
14        return 5;
15    }
16    else if(tokens[i].type==TK_L||tokens[i].type==TK_R)
17    {
18        return 4;
19    }
20    else if(tokens[i].type=='<' || tokens[i].type=='>' || tokens[i].type==TK_GEQ||
21            tokens[i].type==TK_LEQ)
22    {
23        return 3;
24    }
25    else if(tokens[i].type==TK_EQ||tokens[i].type==TK_NEQ)
26    {
27        return 2;
28    }
29    else if(tokens[i].type=='&' || tokens[i].type=='|' || tokens[i].type=='^')

```

```

29 {
30     return 1;
31 }
32 else if (tokens[i].type==TK_AND || tokens[i].type==TK_OR)
33 {
34     return 0;
35 }
36 return 10;
37 }

```

之后是我们要在一个 token 表达式中寻找 DominantOp, 返回其索引位置, 不存在时返回-1。

DominantOp 的基本规则如下:

通过上面这个简单的例子, 我们就可以总结出如何在一个 token 表达式中寻找 dominant operator 了:

- 非运算符的 token 不是 dominant operator.
- 出现在一对括号中的 token 不是 dominant operator. 注意到这里不会出现有括号包围整个表达式的情况, 因为这种情况已经在 check_parentheses() 相应的 if 块中被处理了.
- dominant operator 的优先级在表达式中是最低的. 这是因为 dominant operator 是最后一步才进行的运算符.
- 当有多个运算符的优先级都是最低时, 根据结合性, 最后被结合的运算符才是 dominant operator. 一个例子是 $1+2+3$, 它的 dominant operator 应该是右边的+.

图 3: DominantOp 的基本规则

在 findDominantOp 函数中, 我们遍历所给区间的 tokens, 遇到数字或寄存器则跳过, 遇到左括号则寻找与其匹配的右括号, 并跳过括号区间。当遇到其他符号时, 比较优先级, 返回优先级最低且位置最后的符号所在位置。

findDominantOp 函数

```

1 int findDominantOp(int p, int q)
2 {
3     int i, j;
4     int count=0;
5     int op=10;
6     int op_priority;
7     int pos=-1;
8     for (i=p; i<=q; i++)
9     {
10         if (tokens[i].type==TK_NUMBER || tokens[i].type==TK_REG || tokens[i].type==
            TK_HEX)
11         {
12             continue;
13         }
14         else if (tokens[i].type=='(')
15         {
16             count=0;

```

```

17     for (j=i+1;j<=q;j++)
18     {
19         if (tokens[j].type==' ')
20         {
21             count++;
22             i+=count;
23             break;
24         }
25         else
26         {
27             count++;
28         }
29     }
30 }
31 else
32 {
33     op_priority=priority(i);
34     if (op_priority<=op)
35     {
36         pos=i;
37         op=op_priority;
38     }
39 }
40 }
41 return pos;
42 }

```

之后我们便可进行 eval 函数求值，将所给区间的 tokens 对应的表达式进行值的计算并返回。

具体实现方法为：首先判断区间的合法性，在区间中只有一个 token 时，则只可能为十进制整数、十六进制整数或寄存器，分别进行进制转换及寄存器读取，并返回对应的值。在区间有多个 token 时，首先判断括号的合法性，并进行括号消去后的递归调用。获取区间的 DominantOp，当 DominantOp 为负号时，返回递归调用的负值，当 DominantOp 为解引用时，返回对应位置内存的值，当 DominantOp 为 '!' 时，返回递归调用的逻辑取反。当为其他多目运算符时，分别递归计算 DominantOp 前后的值，并根据计算符返回对应的计算结果。

eval 函数

```

1  int eval(int p,int q)
2  {
3      if(p>q)
4      {
5          printf("error:p>q in eval\n");
6          assert(0);
7      }
8      if(p==q)
9      {
10         int num;
11         switch(tokens[p].type)
12         {

```

```

13     case TK_NUMBER:
14         sscanf(tokens[p].str, "%d", &num);
15         return num;
16     case TK_REG:
17         for(int i=0; i<8; i++)
18         {
19             if(strcmp(tokens[p].str, regsl[i])==0)
20             {
21                 return reg_l(i);
22             }
23             if(strcmp(tokens[p].str, regsw[i])==0)
24             {
25                 return reg_w(i);
26             }
27             if(strcmp(tokens[p].str, regsb[i])==0)
28             {
29                 return reg_b(i);
30             }
31         }
32         if(strcmp(tokens[p].str, "eip")==0)
33         {
34             return cpu.eip;
35         }
36         else
37         {
38             printf("error in TK_REG in eval()\n");
39             assert(0);
40         }
41     case TK_HEX:
42         sscanf(tokens[p].str, "%x", &num);
43         return num;
44     }
45 }
46 if(p<q)
47 {
48     if(check_parentheses(p, q)==1)
49     {
50         return eval(p+1, q-1);
51     }
52     else
53     {
54         int op=findDominantOp(p, q);
55         vaddr_t addr;
56         int result;
57         switch(tokens[op].type)
58         {
59             case TK_NEGATIVE:
60                 return (-eval(p+1, q));

```



```
61     case TK_DEREF:
62         addr=eval(p+1,q);
63         result=vaddr_read(addr,4);
64         printf("addr=%u(0x%x)--->value=%d(0x%08x)\n",addr,addr,result,
65             result);
66         return result;
67     case '!':
68         result=eval(p+1,q);
69         if(result!=0)
70         {
71             return 0;
72         }
73         else
74         {
75             return 1;
76         }
77     }
78     int val1=eval(p,op-1);
79     int val2=eval(op+1,q);
80     switch(tokens[op].type)
81     {
82     case '+':
83         return (val1+val2);
84     case '-':
85         return (val1-val2);
86     case '*':
87         return (val1*val2);
88     case '/':
89         return (val1/val2);
90     case '<':
91         return (val1<val2);
92     case '>':
93         return (val1>val2);
94     case '^':
95         return (val1^val2);
96     case '&':
97         return (val1&val2);
98     case '|':
99         return (val1|val2);
100     case TK_EQ:
101         return (val1==val2);
102     case TK_NEQ:
103         return (val1!=val2);
104     case TK_AND:
105         return (val1&&val2);
106     case TK_OR:
107         return (val1||val2);
108     case TK_GEQ:
```

```

108         return (val1>=val2);
109     case TK_LEQ:
110         return (val1<=val2);
111     case TK_L:
112         return (val1<<val2);
113     case TK_R:
114         return (val1>>val2);
115     default:
116         assert(0);
117     }
118 }
119 }
120 return 0;
121 }

```

除此以外，我们还需实现 `expr` 函数，通过判断 `make_token` 函数执行是否成功，判断是否继续计算。当 `tokens` 首位为 ‘-’ 或者 ‘*’ 时，将其类型分别改为负号及解引用，遍历 `tokens` 数组，当遇到 ‘-’ 或者 ‘*’ 时，判断其前一个不为十进制、十六进制整数或右括号时，将其类型分别改为负号及解引用。最后调用 `eval` 函数，进行表达式的递归计算。

expr 函数

```

1  uint32_t expr(char *e, bool *success) {
2      if (!make_token(e)) {
3          printf("nr_token:%d\n",nr_token);
4          *success = false;
5          return 0;
6      }
7
8      /* TODO: Insert codes to evaluate the expression. */
9      //negative & pointer
10     if(tokens[0].type=='-')
11     {
12         tokens[0].type=TK_NEGATIVE;
13     }
14     if(tokens[0].type=='*')
15     {
16         tokens[0].type=TK_REDEF;
17     }
18     for(int i=1;i<nr_token;i++)
19     {
20         if(tokens[i].type=='-')
21         {
22             if(tokens[i-1].type!=TK_NUMBER&&tokens[i-1].type!='')
23             {
24                 tokens[i].type=TK_NEGATIVE;
25             }
26         }
27         if(tokens[i].type=='*')

```

```

28     {
29         if (tokens[i-1].type!=TK_NUMBER&&tokens[i-1].type!='')
30         {
31             tokens[i].type=TK_REDEF;
32         }
33     }
34 }
35 *success=true;
36
37 return eval(0,nr_token-1);
38 }

```

五、 阶段三

(一) 实现监视点池的管理

首先，我们需要完善监视点的结构体，新增监视点表达式、监视点的值、监视点命中次数三个成员变量，并在其中新增 new_wp, free_wp, print_wp, watch_wp 四个成员变量。

监视点的结构体

```

1 typedef struct watchpoint {
2     int NO;
3     struct watchpoint *next;
4
5     /* TODO: Add more members if necessary */
6
7     int oldval;
8     char wp[32];
9     int hitnum;
10
11 } WP;
12
13 void init_wp_pool();
14 bool new_wp(char *args);
15 bool free_wp(int num);
16 void print_wp();
17 bool watch_wp();

```

在初始化函数中，我们对新增的成员变量进行初始化，并且新增变量记录下一个监视点的编号以及监视点指针。

初始化函数

```

1 void init_wp_pool() {
2     int i;
3     for (i = 0; i < NR_WP; i++) {
4         wp_pool[i].NO = i;
5         wp_pool[i].next = &wp_pool[i + 1];
6         wp_pool[i].oldval=0;

```

```

7     wp_pool[i].hitnum=0;
8 }
9 wp_pool[NR_WP-1].next = NULL;
10
11 head = NULL;
12 free_ = wp_pool;
13 used_next_no=0;
14 }

```

之后我们便可实现监视点的新增与释放函数，new_wp 以及 free_wp 函数。

在 new_wp 函数中，首先需要判断空闲链表中是否有剩余，有剩余的话将头部节点取用到监视点链表中，维护空闲链表头，对新监视点的信息进行初始化（根据所给的参数求值），当监视点列表为空时，将新监视点设为头，不为空时，则将其放到链表尾部。

new_wp 函数

```

1 bool new_wp(char *args)
2 {
3     if(free_==NULL)
4     {
5         assert(0);
6     }
7     WP* result=free_;
8     free_=free_>next;
9     result->NO=used_next_no;
10    used_next_no++;
11    result->next=NULL;
12    strcpy(result->wp,args);
13    result->hitnum=0;
14    bool success=true;
15    result->oldval=expr(result->wp,&success);
16    if(success==false)
17    {
18        printf("error in new_wp\n");
19        return 0;
20    }
21    wp_temp=head;
22    if(wp_temp==NULL)
23    {
24        head=result;
25    }
26    else
27    {
28        while(wp_temp->next!=NULL)
29        {
30            wp_temp=wp_temp->next;
31        }
32        wp_temp->next=result;
33    }

```

```

34     printf("set watchpoint success %d,oldvalue=%d\n",result->NO,result->oldval)
        ;
35     return 1;
36 }

```

在 free_wp 函数中,首先需要判断监视点链表中是否有剩余,有剩余的话将头部节点取用到监视点,将所给的编号与监视点链表中所有节点编号进行对比,若为头节点,则更新维护监视点链表头,若为中间节点,则维护链表连接。如果找到对应的监视点,则将其设为空闲链表的头部。

free_wp 函数

```

1  bool free_wp(int num)
2  {
3      WP *wp=NULL;
4      if(head==NULL)
5      {
6          printf("no watchpoint free now\n");
7          return 0;
8      }
9      if(head->NO==num)
10     {
11         wp=head;
12         head=head->next;
13     }
14     else
15     {
16         wp_temp=head;
17         while(wp_temp!=NULL&&wp_temp->next!=NULL)
18         {
19             if(wp_temp->next->NO==num)
20             {
21                 wp=wp_temp->next;
22                 wp_temp->next=wp_temp->next->next;
23                 break;
24             }
25             wp_temp=wp_temp->next;
26         }
27     }
28     if(wp!=NULL)
29     {
30         wp->next=free_;
31         free_=wp;
32         return 1;
33     }
34     return 0;
35 }

```

(二) 实现监视点

首先我们实现 print_wp 函数，打印监视点链表中所有节点的有关信息。

print_wp 函数

```

1 void print_wp()
2 {
3     if(head==NULL)
4     {
5         printf("no watchpoint\n");
6         return;
7     }
8     else
9     {
10        printf("watchpoint:\n");
11        wp_temp=head;
12        while(wp_temp!=NULL)
13        {
14            printf("NO.%d      expr:%s      hittime:%d\n",wp_temp->NO,wp_temp->wp
15                  ,wp_temp->hitnum);
16            wp_temp=wp_temp->next;
17        }
18    }

```

之后实现 watch_wp 函数，对比每一个监视点链表中的监视点表达式计算得到的值是否变化，若变化，则输出相关信息，并对命中次数进行增加，并改变该监视点的值。

watch_wp 函数

```

1 //judge if the watchpoint trigger
2 bool watch_wp()
3 {
4     bool success=true;
5     int res;
6     if(head==NULL)
7     {
8         return 1;
9     }
10    wp_temp=head;
11    while(wp_temp!=NULL)
12    {
13        res=expr(wp_temp->wp,&success);
14        if(res!=wp_temp->oldval)
15        {
16            wp_temp->hitnum++;
17            printf("watchpoint %d:%s\n",wp_temp->NO,wp_temp->wp);
18            printf("oldvalue:%d\n",wp_temp->oldval);
19            printf("newvalue:%d\n\n",res)
20            wp_temp->oldval=res;

```

```

21     return 0;
22 }
23 wp_temp=wp_temp->next;
24 }
25 return 1;
26 }

```

之后我们在 ui.c 中继续实现指令，我们补充实现 w 和 d 指令，分别为设置以及删除 watchpoint。w 指令，只需调用 new_wp 函数即可，d 指令需要对输入的参数转化为整型，并判断其合法性，并调用 free_wp 函数进行监视点的释放。

实现 w 和 d 指令

```

1 static int cmd_w(char *args)
2 {
3     new_wp(args);
4     return 0;
5 }
6
7 static int cmd_d(char *args)
8 {
9     int num=0;
10    int n=sscanf(args,"%d",&num);
11    if(n<=0)
12    {
13        printf("args error in cmd_d\n");
14        return 0;
15    }
16    int freewp=free_wp(num);
17    if(freewp==false)
18    {
19        printf("error:no watchpoint %d\n",num);
20    }
21    else
22    {
23        printf("delete watchpoint success %d\n",num);
24    }
25    return 0;
26 }

```

最后，我们在 cpu-exec.c 文件中添加 watchpoint 头文件，并且在 DEBUG 中添加监视点的触发判断。

cpu-exec.c 文件

```

1 #ifndef DEBUG
2     /* TODO: check watchpoints here. */
3     if (watch_wp()==false)
4     {
5         nemu_state=NEMU_STOP;
6     }

```

```
7  
8 #endif
```

六、 遇到的问题以及解决办法

(一) git log 丢失问题

在学习中，我曾遇到过 git log 丢失的问题，经过与学长的交流，得知是在 git commit 过程未完成的时候强制关闭了 git 进程导致。并且得知了解决办法。

```
shox@ubuntu:~/ics2017$ git log  
error: object file .git/objects/81/50592ada0e53ef3c0c2d01b72c69e8d18aeffc is empty  
error: object file .git/objects/81/50592ada0e53ef3c0c2d01b72c69e8d18aeffc is empty  
fatal: loose object 8150592ada0e53ef3c0c2d01b72c69e8d18aeffc (stored in .git/objects/81/50592ada0e53ef3c0c2d01b72c69e8d18aeffc) is corrupt
```

图 4: git log 丢失

How to fix Git error: object file is empty?

#git #beginners

Step 1: Make a backup of your .git

```
# Copy your git folder to git-old  
cp -a .git .git-old
```

Step 2: remove empty files

```
# Go into the git folder  
cd .git/  
# Delete all empty files in the curent folder and sub-folders  
find . -type f -empty -delete -print
```

Step 3: check if everything's right

```
# Verifies the connectivity and validity of the objects in the git folder  
git fsck --full  
# Show the repos status  
git status
```

If you have more issues, check this topic in stackoverflow it is very helpful

图 5: 解决办法

七、 问题

(一) 究竟要执行多久?

究竟要执行多久?

在 `cmd_c()` 函数中, 调用 `cpu_exec()` 的时候传入了参数-1, 你知道这是什么意思吗?

图 6: 究竟要执行多久?

我们打开 `nemu/src/monitor/cpu-exec.c` 文件, 观察 `cpu_exec()` 函数, 这个函数模仿 CPU 的工作方式, 不断执行 `n` 条指令, 直到指令执行完毕或进入 `nemu_trap`, 才退出指令执行的循环。

传入参数为 `uint64_t` 类型, 即 `unsigned long long` (64 位无符号整型) 当我们给它传入-1 时, 相当于 `uint64_t` 中的 264-1 (最大值), 因此保证函数能不断执行指令直到 `nemu_trap`。

(二) 谁来指示程序的结束?

谁来指示程序的结束?

在程序设计课上老师告诉你, 当程序执行到 `main()` 函数返回处的时候, 程序就退出了, 你对此深信不疑. 但你是否怀疑过, 凭什么程序执行到 `main()` 函数的返回处就结束了? 如果有人告诉你, 程序设计课上老师的说法是错的, 你有办法来证明/反驳吗? 如果你对此感兴趣, 请在互联网上搜索相关内容.

图 7: 谁来指示程序的结束?

谁来指示程序的结束?

```

1 #include <iostream>
2 using namespace std;
3 void func() { cout << "After main" << endl; }
4 int main()
5 {
6     atexit(&func);
7     cout << "End of main" << endl;
8     return 0;
9     system("pause");
10 }

```



图 8: 谁来指示程序的结束?

通过测试我们不难发现程序并不是执行到 `main()` 函数的返回处就结束了。

`exit()` 函数用于在程序运行的过程中随时结束程序, 其原型为: `void exit(int state)`; `exit` 的参数 `state` 是返回给操作系统或当前程序的调用程序, 返回 0 表示程序正常结束, 非 0 表示程序非正常结束。`main` 函数结束时也会隐式地调用 `exit()` 函数。`exit()` 函数运行时首先会执行由 `atexit()` 函数登记的函数, 然后会做一些自身的清理工作, 同时刷新所有输出流、关闭所有打开的流并且关闭通过标准 I/O 函数 `tmpfile()` 创建的临时文件。

`atexit()` 用于注册终止函数 (即 `main` 执行结束后调用的函数), 其原型为: `int atexit(void (*function)(void))`; 很多时候我们需要在程序退出的时候做一些诸如释放资源的操作, 但程序退出的方式有很多种, 比如 `main()` 函数运行结束、在程序的某个地方用 `exit()` 结束程序、用户通过 `Ctrl+C` 或 `Ctrl+break` 操作来终止程序等等, 因此需要有一种与程序退出方式无关的方法来进行程序退出时的必要处理。方法就是用 `atexit()` 函数来注册程序正常终止时要被调用的函数。

(三) 为什么要使用 `static`?

温故而知新

框架代码中定义 `wp_pool` 等变量的时候使用了关键字 `static`, `static` 在此处的含义是什么? 为什么要在此处使用它?

图 9: 为什么要使用 `static`?

`static` 定义全局静态变量, 该变量只能在本文件中进行访问, 保护了数据的安全性。这里 `static` 表示 `wp_pool` 为静态全局变量, 此处使用 `static` 是为了避免 `wp_pool` 在程序运行中被误修改。其他源文件想对该变量进行操作时, 需要通过封装的函数来实现。

(四) 一点也不能长?

一点也不能长?

我们知道 `int3` 指令不带任何操作数, 操作码为 1 个字节, 因此指令的长度是 1 个字节. 这是必须的吗? 假设有一种 `x86` 体系结构的变种 `my-x86`, 除了 `int3` 指令的长度变成了 2 个字节之外, 其余指令和 `x86` 相同. 在 `my-x86` 中, 文章中的断点机制还可以正常工作吗? 为什么?

图 10: 一点也不能长?

是的。`int3` 指令的操作码为 `0xCC`, 它是一个单字节的操作码, 后面没有操作数, 因此指令的长度为 1 个字节。指令长度是处理器解码和执行指令的关键信息之一。处理器必须能够准确地确定指令的长度, 以便正确地解析指令并执行正确的操作。因此, `int3` 指令的长度为 1 个字节是必须的。

在 `x86` 体系结构中, `int3` 指令被用作软件断点。在标准的 `x86` 指令集中, `int3` 指令的长度为 1 字节。在 `my-x86` 中, 由于 `int3` 指令的长度变成了 2 个字节, 因此在原有的 `x86` 调试器中, 可能会在执行到 `int3` 指令时读取错误的数, 导致断点机制不能正常工作。

为了使断点机制正常工作, 调试器需要能够识别 `my-x86` 的特殊指令长度, 并对其进行特殊处理。否则, 调试器将不能准确地识别 `int3` 指令, 并将继续执行下一条指令, 从而导致断点机制失效。

因此, 如果调试器能够正确地处理 `my-x86` 的指令长度变化, 断点机制仍然可以正常工作。但如果调试器无法适应 `my-x86` 的变化, 则断点机制将不能正常工作。

(五) 随心所欲的断点

随心所欲”的断点

如果把断点设置在指令的非首字节(中间或末尾), 会发生什么? 你可以在 GDB 中尝试一下, 然后思考并解释其中的缘由。

图 11: 随心所欲的断点

在 `x86` 指令中, 指令的非首字节通常表示指令的操作数或修饰符, 而不是指令的起始地址。因此, 在 GDB 中将断点设置在指令的非首字节上, 实际上是将断点设置在指令的某个操作数或修饰符上, 而不是指令的起始地址。当程序执行到这个断点时, CPU 会尝试执行这个指令的操作数或修饰符, 而不是执行整个指令。因此, 这个断点可能不会导致程序停止, 而是导致程序以不可预测的方式执行错误的操作数或修饰符。

例如, 在 GDB 中, 我们可以使用命令 `b *0x080483f7+2` 来将断点设置在指令的非首字节上。这个指令是一个 `mov` 指令, 它的操作数在指令的第 3 个字节中。当程序执行到这个断点时, CPU 会尝试执行这个操作数, 而不是执行整个指令。如果这个操作数是一个无效的地址, 程序就会发生崩溃。

(六) NEMU 的前世今生

NEMU 的前世今生

你已经对 NEMU 的工作方式有所了解了. 事实上在 NEMU 诞生之前, NEMU 曾经有一段时间并不叫 NEMU, 而是叫 NDB (NJU Debugger), 后来由于某种原因才改名为 NEMU. 如果你想知道这一段史前的秘密, 你首先需要了解这样一个问题: 模拟器 (Emulator) 和调试器 (Debugger) 有什么不同? 更具体地, 和 NEMU 相比, GDB 到底是如何调试程序的?

图 12: NEMU 的前世今生

模拟器是一种软件工具, 它可以在一种硬件平台上模拟另一种硬件平台。它能够执行另一种硬件平台上的指令, 并将其转换成主机平台上的指令, 从而使主机平台能够运行另一种硬件平台的应用程序。模拟器通常用于开发和测试不同平台的应用程序。调试器是一种用于调试程序的工具, 它可以帮助开发人员识别和修复程序中的错误。调试器通常提供了各种功能, 如设置断点、单步执行、查看和修改变量值等, 以帮助开发人员找到程序中的错误。

NEMU 是一种模拟器, 它能够模拟 x86 计算机的硬件平台, 从而使用户能够在不同的操作系统环境中运行和调试 x86 计算机上的程序。NEMU 还包含了调试器的功能, 例如单步执行、设置断点和查看变量等。GDB 是一种典型的调试器, 它能够在 x86 计算机上调试程序。GDB 提供了各种功能, 如设置断点、单步执行、查看和修改变量值等, 以帮助开发人员找到程序中的错误。与 NEMU 不同, GDB 不提供模拟器的功能, 它只能调试运行在本地 x86 计算机上的程序。

八、 必答题

你需要在实验报告中回答下列问题:

①查阅 i386 手册理解了科学查阅手册的方法之后, 请你尝试在 i386 手册中查阅以下问题所在的位置, 把需要阅读的范围写到你的实验报告里面:

- ✧ EFLAGS 寄存器中的 CF 位是什么意思?
- ✧ ModR/M 字节是什么?
- ✧ mov 指令的具体格式是怎么样的?

②shell 命令完成 PA1 的内容之后, nemu/ 目录下的所有 .c 和 .h 文件总共有多少行代码? 你是使用什么命令得到这个结果的? 和框架代码相比, 你在 PA1 中编写了多少行代码? (Hint: 目前 2017 分支中记录的正好是做 PA1 之前的状态, 思考一下应该如何回到“过去”?) 你可以把这条命令写入 Makefile 中, 随着实验进度的推进, 你可以很方便地统计工程的代码行数, 例如敲入 make count 就会自动运行统计代码行数的命令。再来个难一点的, 除去空行之外, nemu/ 目录下的所有 .c 和 .h 文件总共有多少行代码?

③使用 man 打开工程目录下的 Makefile 文件, 你会在 CFLAGS 变量中看到 gcc 的一些编译选项。请解释 gcc 中的 -Wall 和 -Werror 有什么作用? 为什么要使用 -Wall 和 -Werror?

图 13: 必答题

(一) i386 手册

1. EFLAGS

我们在 i386 手册中搜索 EFLAGS 相关信息。发现 2.3.4.1 节和附录 c 有提及相关内容。

2.3.4.1 Status Flags

The status flags of the **EFLAGS** register allow the results of one instruction to influence later instructions. The arithmetic instructions use OF, SF, ZF, AF, PF, and CF. The SCAS (Scan String), CMPS (Compare String), and LOOP instructions use ZF to signal that their operations are complete. There are instructions to set, clear, and complement CF before execution of an arithmetic instruction. Refer to Appendix C for definition of each status flag.

图 14: 2.3.4.1 节

Appendix C Status Flag Summary

Status Flags' Functions

Bit	Name	Function
0	CF	Carry Flag — Set on high-order bit carry or borrow; cleared otherwise.
2	PF	Parity Flag — Set if low-order eight bits of result contain an even number of 1 bits; cleared otherwise.
4	AF	Adjust flag — Set on carry from or borrow to the low order four bits of AL; cleared otherwise. Used for decimal arithmetic.
6	ZF	Zero Flag — Set if result is zero; cleared otherwise.
7	SF	Sign Flag — Set equal to high-order bit of result (0 is positive, 1 if negative).
11	OF	Overflow Flag — Set if result is too large a positive number or too small a negative number (excluding sign-bit) to fit in destination operand; cleared otherwise.

图 15: 附录 c

查阅得：CF 为 EFLAGS 寄存器的进位标志，如果运算导致最高位产生进位或者借位则为 1。否则为 0。

2. ModR/M

我们在 i386 手册中搜索 ModR/M 相关信息，发现 ModR/M 相关内容在 17.2.1 节，查阅可知，大多数可以引用内存中的操作数的指令在主操作码字节之后都有一个寻址形式的字节，即为 ModR/M 字节。其包括 MOD、REG/OPCODE、R/M 三方面内容，分别表示索引类型或者寄存器编号、寻址模式编码等信息。通过上面三个部分值的变化，以及 16 位或 32 位寻址方式，我们再根据手册中的表格可以查看具体寻址方式。

17.2.1 ModR/M and SIB Bytes

The **ModR/M** and SIB bytes follow the opcode byte(s) in many of the 80386 instructions. They contain the following information:

- The indexing type or register number to be used in the instruction
- The register to be used, or more information to select the instruction
- The base, index, and scale information

The **ModR/M** byte contains three fields of information:

- The mod field, which occupies the two most significant bits of the byte, combines with the r/m field to form 32 possible values: eight registers and 24 indexing modes

Page 241 of 421

INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986

- The reg field, which occupies the next three bits following the mod field, specifies either a register number or three more bits of opcode information. The meaning of the reg field is determined by the first (opcode) byte of the instruction.
- The r/m field, which occupies the three least significant bits of the byte, can specify a register as the location of an operand, or can form part of the addressing-mode encoding in combination with the field as described above

图 16: 17.2.1 节

3. mov

我们在 i386 手册中搜索 `mov` 相关信息，查阅可知格式为 $DEST \leftarrow SRC$ ，具体格式是 `mov 目的操作数, 源操作数`。

MOV — Move Data

Opcode	Instruction	Clocks	Description
88 /r	MOV r/m8,r8	2/2	Move byte register to r/m byte
89 /r	MOV r/m16,r16	2/2	Move word register to r/m word
89 /r	MOV r/m32,r32	2/2	Move dword register to r/m dword
8A /r	MOV r8,r/m8	2/4	Move r/m byte to byte register
8B /r	MOV r16,r/m16	2/4	Move r/m word to word register
8B /r	MOV r32,r/m32	2/4	Move r/m dword to dword register
8C /r	MOV r/m16,Sreg	2/2	Move segment register to r/m word
8D /r	MOV Sreg,r/m16	2/5,pm=18/19	Move r/m word to segment register
A0	MOV AL,moffs8	4	Move byte at (seg:offset) to AL
A1	MOV AX,moffs16	4	Move word at (seg:offset) to AX
A1	MOV EAX,moffs32	4	Move dword at (seg:offset) to EAX
A2	MOV moffs8,AL	2	Move AL to (seg:offset)
A3	MOV moffs16,AX	2	Move AX to (seg:offset)
A3	MOV moffs32,EAX	2	Move EAX to (seg:offset)
B0 + rb	MOV reg8,imm8	2	Move immediate byte to register
B8 + rw	MOV reg16,imm16	2	Move immediate word to register
B8 + rd	MOV reg32,imm32	2	Move immediate dword to register
C7iiii	MOV r/m8,imm8	2/2	Move immediate byte to r/m byte
C7	MOV r/m16,imm16	2/2	Move immediate word to r/m word
C7	MOV r/m32,imm32	2/2	Move immediate dword to r/m dword

NOTES:

moffs8, moffs16, and moffs32 all consist of a simple offset relative to the segment base. The 8, 16, and 32 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16 or 32 bits.

Operation

$DEST \leftarrow SRC;$

Description

MOV copies the second operand to the first operand.

图 17: mov 指令

(二) shell 命令

我们可以通过如下指令获取代码行数，其中第一个为全部行数，第二个为去空行后行数。

```
shox@ubuntu:~/ics2017/nemu$ find . -name "*.ch" |xargs cat|wc -l
4098
shox@ubuntu:~/ics2017/nemu$ find . -name "*.ch" |xargs cat|grep -v ^$|wc -l
3422
shox@ubuntu:~/ics2017/nemu$
```

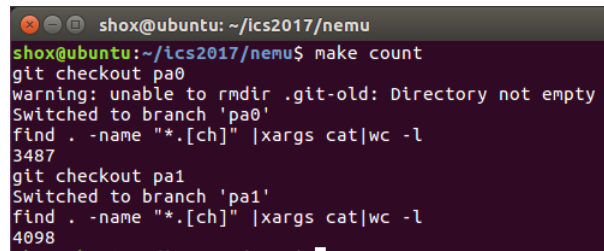
图 18: mov 指令

为统计新增代码行数，我们可以将代码切到上一个实验的分支，统计行数，在切换回本实验的分支，统计行数，相减则可得出结果。

我们可以把这条命令写入 Makefile 中，随着实验进度的推进，我们可以很方便地统计工程的代码行数，敲入 make count 就会自动运行统计代码行数的命令。

```
count:
    git checkout pa0
    find . -name "*.ch" |xargs cat|wc -l
    git checkout pa1
    find . -name "*.ch" |xargs cat|wc -l
```

图 19: Makefile

A terminal window with a dark background and light text. The prompt is 'shox@ubuntu: ~/ics2017/nemu'. The user enters 'make count'. The output shows 'git checkout pa0', a warning about '.git-old', switching to branch 'pa0', and a 'find' command result of 3487. Then the user enters 'git checkout pa1', switches to branch 'pa1', and the 'find' command result is 4098.

```
shox@ubuntu: ~/ics2017/nemu
shox@ubuntu:~/ics2017/nemu$ make count
git checkout pa0
warning: unable to rmdir .git-old: Directory not empty
Switched to branch 'pa0'
find . -name "*.ch" |xargs cat|wc -l
3487
git checkout pa1
Switched to branch 'pa1'
find . -name "*.ch" |xargs cat|wc -l
4098
```

图 20: 统计代码行数

(三) gcc

-Wall: 使 GCC 产生尽可能多的警告信息, 取消编译操作, 打印出编译时所有错误或警告信息。

-Werror: 在发生警告时停止编译操作, 即要求 GCC 将所有的警告当成错误进行处理, 从而终止编译操作。

使用-Wall 和-Werror 可以找出所有存在的或者潜在的错误, 提高代码的安全性, 优化程序, 便于进行代码维护以及 debug 操作。