



南開大學
Nankai University

计算机学院
计算机系统设计实验报告

PA1

姓名：高祎珂

学号：2011743

专业：计算机科学与技术

2023 年 3 月 21 日

目录

1 实验目的	2
2 实验内容	2
3 RTFSC	2
3.1 实现正确的寄存器结构体	2
4 阶段一	4
4.1 单步执行	4
4.2 打印寄存器	4
4.3 扫描内存	6
5 阶段二	7
5.1 实现词法分析	7
5.1.1 添加 token 规则	7
5.1.2 token 匹配	8
5.2 递归求值	10
5.2.1 左右括号匹配	10
5.2.2 寻找 dominant operator	11
5.2.3 扩展表达式求值	12
6 阶段三	13
6.1 监视点链表维护	13
6.1.1 申请新监视点	13
6.1.2 释放监视点	14
6.1.3 插入新监视节点	14
6.1.4 删除对应序号的监视点	15
6.1.5 打印所有的监视点	15
6.2 监视点管理	16
7 遇到的问题和解决办法	17
8 必答题	17

1 实验目的

在计算机技术的发展历程中，先驱们为创造计算机世界所做出的努力不可忽视。这些先驱们准备好了各种工具，用以开创计算机领域的先河。为了迈出第一步，他们运用了数字电路的知识，成功地创造出了最小的计算机——图灵机。如今，计算机技术已经发展成为支撑现代社会的重要基石，而图灵机则成为计算机理论的经典代表之一。本次实验通过对 NEMU 的基础设施、表达式求值以及监视点等内容的编写，可以更深刻地领会计算机技术的奥妙所在。

2 实验内容

本次实验总共分为以下几个阶段

- 实现单步执行, 打印寄存器状态, 扫描内存
- 实现调试功能的表达式求值
- 实现调试的监视点功能

3 RTFSC

3.1 实现正确的寄存器结构体

为了兼容 X86，这里使用的寄存器结构如下所示：

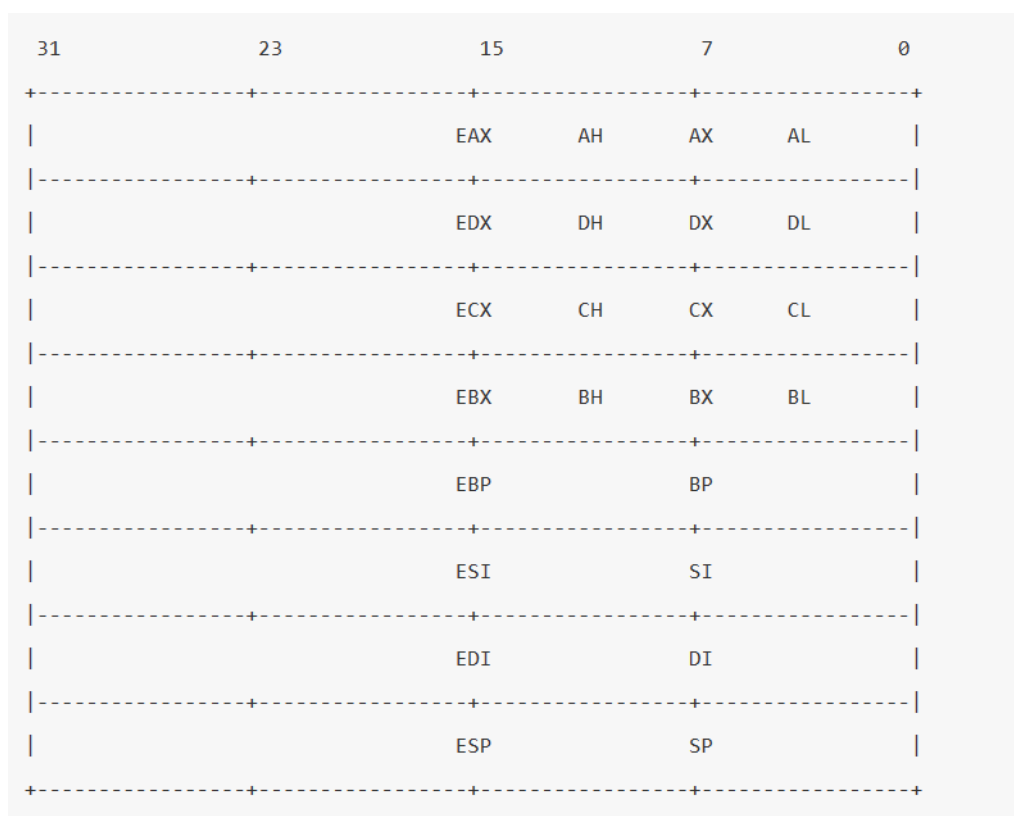


图 3.1: 寄存器结构

其中

1. EAX, EDX, ECX, EBX, EBP, ESI, EDI, ESP 是 32 位寄存器;
2. AX, DX, CX, BX, BP, SI, DI, SP 是 16 位寄存器;
3. AL, DL, CL, BL, AH, DH, CH, BH 是 8 位寄存器。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.

经过查看框架代码, 我们可以发现, 给出的寄存器结构为:

原始寄存器结构

```
1 typedef struct {
2     struct {
3         uint32_t _32;
4         uint16_t _16;
5         uint8_t _8[2];
6     } gpr[8];
7     rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
8
9     vaddr_t eip;
10
11 } CPU_state;
```

• struct 与 union 的区别

struct 和 union 都是由不同的数据类型成员组成的, 但是 struct 所有成员占用的内存空间是累加的, 而 union 中所有的成员共用一块地址空间。struct 中的内存空间等于所有成员长度之和; union 成员不能同时占用内存空间, 长度等于最长成员的长度。

而根据上面对于寄存器结构的叙述, 我们可以发现, 其实寄存器是共用内存的, 相对于 struct 而言, 使用匿名 union 这种数据结构来描述寄存器其实更合适, 因此修改代码为:

修改后寄存器结构

```
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             rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
10         };
11     };
12
13     vaddr_t eip;
14
15 } CPU_state;
```

4 阶段一

4.1 单步执行

这里仿照给出的 help 和 c 指令的代码，从 cmd_c() 和 cmd_help() 函数获得思路进行 cmd_si() 函数的编写。一下编写的每条指令均需在 cmd_table [] 里补全描述和句柄。整体编写大致思路如下：

- 仿照 help 函数编写，利用 strock 函数得到函数参数，但是得到的 arg 是一个字符串类型，所以需要强制类型转换，使用 atoi() 函数讲字符串类型转化为所需的 int 型。
- 考虑到 si 的用法，这里还有可能输入参数为空，这里我们需要给帝国一个默认参数为 1，如果未传参时，就会使用此默认参数。
- 仿照 cmd_c() 函数，可以发现实现代码执行是使用 cpu_exec() 函数，通过阅读源码我们可以发现，cpu_exec() 函数传入的参数即为程序执行的步数，所以这里直接调用 cpu_exec() 即可。

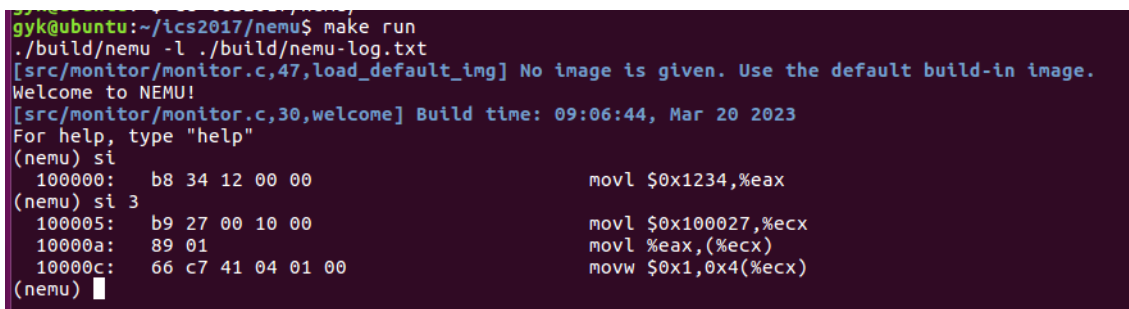
具体代码如下：

```

                                cmd_si()
1  static int cmd_si(char *args) {
2      /*get the steps number*/
3      int steps;
4      if (args == NULL){
5          steps = 1;
6      }
7      else{
8          steps = atoi(strtok(NULL, " "));
9      }
10
11     cpu_exec(steps);
12     return 0;
13 }

```

运行效果如下：



```

gyk@ubuntu:~/ics2017/nemu$ make run
./build/nemu -l ./build/nemu-log.txt
[src/monitor/monitor.c,47,load_default_img] No image is given. Use the default build-in image.
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 09:06:44, Mar 20 2023
For help, type "help"
(nemu) si
100000: b8 34 12 00 00          movl $0x1234,%eax
(nemu) si 3
100005: b9 27 00 10 00          movl $0x100027,%ecx
10000a: 89 01                  movl %eax,(%ecx)
10000c: 66 c7 41 04 01 00      movw $0x1,0x4(%ecx)
(nemu)

```

图 4.2: si 运行效果

4.2 打印寄存器

这里基本书写思路与上述一样，目前可以不用调用所给的函数，这里仿照 gdb 打印寄存器的实例：

寄存器名称	十六进制	十进制
-------	------	-----

所以这里也计划打印 cpu 里的 8 个寄存器的值，因为对于 “info” 命令可以有两个参数，r 和 w，所以这里先把整个 cmd_info() 寒素的框架搭好，但是函数内部先只实现参数 “r” 的功能，具体实现思路如下：

- 获取输入参数，如果参数为 NULL，打印错误提示信息。
- 将输入参数与 “r” / “w” 进行字符串对比，为 “r”，则执行打印寄存器值的操作，若为 “w”，这里先不执行。
- 若输入参数不是 “r” / “w”，则打印错误提示信息。

具体代码如下：

打印寄存器代码

```

1 static int cmd_info(char *args) {
2     if (args == NULL) {
3         printf("Please input the info r or info w\n");
4     }
5     else {
6         if (strcmp(args, "r") == 0) {
7             printf("eax: 0x%-10x    %-10d\n", cpu.eax, cpu.eax);
8             printf("edx: 0x%-10x    %-10d\n", cpu.edx, cpu.edx);
9             printf("ecx: 0x%-10x    %-10d\n", cpu.ecx, cpu.ecx);
10            printf("ebx: 0x%-10x    %-10d\n", cpu.ebx, cpu.ebx);
11            printf("ebp: 0x%-10x    %-10d\n", cpu.ebp, cpu.ebp);
12            printf("esi: 0x%-10x    %-10d\n", cpu.esi, cpu.esi);
13            printf("esp: 0x%-10x    %-10d\n", cpu.esp, cpu.esp);
14            printf("eip: 0x%-10x    %-10d\n", cpu.eip, cpu.eip);
15        }
16        else if (strcmp(args, "w") == 0) {
17
18        }
19        else {
20            printf("The info command need a parameter 'r' or 'w'\n");
21        }
22    }
23    return 0;
24 }

```

实现结果如下图：

```

(nemu) info r
eax: 0x1234          4660
edx: 0x6f3bdac5     1866193605
ecx: 0x100027        1048615
ebx: 0x5b4d5b3a     1531796282
ebp: 0x20fabade     553302750
esi: 0x3a88df1d     982048541
esp: 0x3b75c96a     997575018
eip: 0x100012        1048594
(nemu) q
gyk@ubuntu:~/ics2017/nemu$

```

图 4.3: 打印寄存器

4.3 扫描内存

此指令与上面两个不太一样的时，这个需要承接两个参数，这里我使用的函数是 `strtok()` 函数得到分割的参数，要使用强制类型转换，而第二个参数表达式那里，此时是十六进制数，所以需要书写函数进行值的转换。具体思路如下：

- 如果输入参数不合法，打印错误提示信息
- 读取参数，行数和起始地址，并进行相应的转化，调用所给函数 `vaddr_read()`，进行内存的读取。

实现代码如下：

简单表达式内存扫描

```

1  static int cmd_x(char *args) {
2  if (args == NULL) {
3      printf("Input invalid command!\n");
4  }
5  else {
6      int num, addr, i;
7      char *exp;
8      num = atoi(strtok(NULL, " "));
9      exp = strtok(NULL, " ");
10     //trans函数这里为将字符串"0x12"转化为int型"18"的函数
11     addr = trans(exp);
12     for (i = 0; i < num; i++) {
13         printf("0x%x\n", vaddr_read(addr, 4));
14         addr += 4;
15     }
16 }
17 }
18 return 0;
19 }

```

运行效果如下：

```

const uint8_t img [] = {
    0xb8, 0x34, 0x12, 0x00, 0x00,          // 100000: movl $0x1234,%eax
    0xb9, 0x27, 0x00, 0x10, 0x00,          // 100005: movl $0x100027,%ecx
    0x89, 0x01,                             // 10000a: movl %eax,(%ecx)
    0x66, 0xc7, 0x41, 0x04, 0x01, 0x00,    // 10000c: movw $0x1,0x4(%ecx)
    0xbb, 0x02, 0x00, 0x00, 0x00,          // 100012: movl $0x2,%ebx
    0x66, 0xc7, 0x84, 0x99, 0x00, 0xe0,    // 100017: movw $0x1,-0x2000(%ecx,%ebx,4)
    0xff, 0xff, 0x01, 0x00,
    0xb8, 0x00, 0x00, 0x00, 0x00,          // 100021: movl $0x0,%eax
    0xd6,                                  // 100026: nemu_trap
};

```

图 4.5: 默认镜像内容

```

gyk@ubuntu:~/ics2017/nemu$ make run
./build/nemu -l ./build/nemu-log.txt
[src/monitor/monitor.c,47,load_default_img] No image is given. Use the default build-
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 09:06:44, Mar 20 2023
For help, type "help"
(nemu) x 10 0x100000
0x1234b8
0x27b900
0x1890010
0x441c766
0x2bb0001
0x66000000
0x9984c7
0x1ffffe0
0xb800
0xd60000
(nemu)

```

图 4.4: 打印内存效果

查看默认镜像，这里的内存内容为下图，对别人这可以发现，已成功实现预想功能。

5 阶段二

5.1 实现词法分析

5.1.1 添加 token 规则

这里需要现在枚举类型中写出所有类型，需要添加的与基本运算符，十进制、十六进制数据，逻辑运算符，大小括号，接着在 rules 数组中补全规则，这里还需要给出一些运算符的优先级，所以我这里又重新写了一个判断优先级的函数。具体实现如下：

rules 数组

```

1 rules [] = {
2     {" +", TK_NOTYPE},          // spaces
3     {" \\+", ADD},             // plus
4     {" ==", TK_EQ},            // equal
5     {" 0[xX][0-9a-fA-F]+", HEX}, // hex number

```



```

6      {"[0-9]+", NUM},          // numbers
7      {"\\-", SUB},            // minus
8      {"\\*", MULTIPLY},       // multiply
9      {"\\/", DIVIDE},         // divide
10     {"\\(", LBRACKET},        // left bracket
11     {"\\)", RBRACKET},        // right bracket
12     {"\\$e[abc]x", REG},      // register
13     {"\\$e[bs]p", REG},
14     {"\\$e[sd]i", REG},
15     {"\\$eip", REG},
16     {"&&", AND},              // and
17     {"\\|\\|", OR},           // or
18     {"!=", NEQ},              // not equal
19 }

```

这里为了后续计算方便，还需要给出各个运算符的优先级，所以这里写了一个优先级函数，具体如下：

运算符优先级函数

```

1 int priority(int i) {
2     if (tokens[i].type == ADD || tokens[i].type == SUB) return 4;
3     else if (tokens[i].type == MULTIPLY || tokens[i].type == DIVIDE) return 3;
4     else if (tokens[i].type == OR) return 12;
5     else if (tokens[i].type == AND) return 11;
6     else if (tokens[i].type == NEQ || tokens[i].type == TK_EQ) return 7;
7     return 0;
8 }

```

5.1.2 token 匹配

实现匹配，主要是补全 `make_token()` 函数的，识别出 token，并写到 `tokens` 数组中。Token 结构体中有两个成员变量，一个是 `int` 型的 `type`，一个是 `char` 类型的 `str`，其中 `type` 成员用于记录 token 的类型。大部分 token 只要记录类型就可以了，例如 `+`，`-`，`*`，`/`，但对于有些 token 类型是不够的：如果我们只记录了一个十进制整数 token 的类型，在进行求值的时候我们还是不知道这个十进制整数是多少。这时我们应该将 token 相应的子串也记录下来，`str` 成员就是用来做这件事情的。`tokens` 数组用于按顺序存放已经被识别出的 token 信息。具体实现思路如下：

- 利用 `position` 变量来指示当前处理到的位置，遍历 `rules` 数组中的每一条规则
- 如果匹配成功成功，则更新 `position`，`nr_token`，且把匹配到信息按序填入 `tokens` 数组中，判断 token 类型，若有需要，补充 `str` 字段。如果未匹配成功，打印错误信息，跳出函数。
- 如果未到字符串结尾，则重复上述两步，否则识别终止，跳出函数。

具体代码实现如下：

token 匹配

```

1 static bool make_token(char *e) {
2     int position = 0;

```

```

3  int i;
4  regmatch_t pmatch;
5  nr_token = 0;
6  while (e[position] != '\0') {
7      /* Try all rules one by one. */
8      for (i = 0; i < NR_REGEX; i++) {
9          if (regexec(&re[i], e + position, 1, &pmatch, 0) == 0 && pmatch.rm_so == 0) {
10             char *substr_start = e + position;
11             int substr_len = pmatch.rm_eo;
12             position += substr_len;
13             tokens[nr_token].type = rules[i].token_type;
14             switch (rules[i].token_type) {
15                 case TK_NOTYPE:
16                     break;
17                 case NUM:
18                 case REG:
19                 case HEX:
20                     for (i = 0; i < substr_len; i++)
21                         tokens[nr_token].str[i] = substr_start[i];
22                     tokens[nr_token].str[i] = '\0';
23                     nr_token++;
24                     break;
25                 case ADD:
26                 case SUB:
27                 case MULTIPLY:
28                 case DIVIDE:
29                 case LBRACKET:
30                 case RBRACKET:
31                     tokens[nr_token].str[0] = substr_start[0];
32                     tokens[nr_token++].str[1] = '\0';
33                     break;
34                 case AND:
35                 case OR:
36                 case TK_EQ:
37                 case NEQ:
38                     tokens[nr_token].str[0] = substr_start[0];
39                     tokens[nr_token].str[1] = substr_start[1];
40                     tokens[nr_token++].str[2] = '\0';
41                     break;
42                 default: TODO();
43             }
44             break;
45         }
46     }
47
48     if (i == NR_REGEX) {
49         printf("no match at position %d\n%s\n%.s^n", position, e, position, "");
50         return false;
51     }

```

```

52     }
53     return true;
54 }

```

5.2 递归求值

5.2.1 左右括号匹配

check_parentheses() 函数用于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, 如果不匹配, 这个表达式肯定是不符合语法的, 也就不需要继续进行求值了. 而这里为了后续识别不匹配是因为不合法, 还是没有被括号包围, 所以这里我又引入了一个关于表达式括号合法的识别函数 judge_exp(),y 具体实现思路如下:

- 从输入的其实位置 p 开始, 遍历表达式
- 设置一个变量 bra, 记录左右括号的情况, 如果在表达式内部, bra 为 0, 那么说明这不是一个被括号包围的情况
- 如果能够遍历结束, 说明是被括号包围的, 返回 true

具体代码实现如下:

check_parentheses(int p,int q) 函数

```

1  bool check_parentheses(int p, int q) {
2  int i, bra = 0;
3  for (i = p; i <= q; i++) {
4      if (tokens[i].type == LBRACKET) {
5          bra++;
6      }
7      if (tokens[i].type == RBRACKET) {
8          bra--;
9      }
10     if(bra == 0 && i < q) {
11         return false;
12     }
13 }
14 return true;
15 }

```

判断表达式括号是否合法

```

1  bool judge_exp() {
2  int i, cnt;
3  cnt = 0;
4  for (i = 0; i <= nr_token; i++) {
5      if (tokens[i].type == LBRACKET)
6          cnt++;
7      else if (tokens[i].type == RBRACKET)
8          cnt--;

```

```

9
10     if (cnt < 0)
11         return false;
12     }
13     return true;
14 }

```

至此，括号匹配已经结束。

5.2.2 寻找 dominant operator

dominant operator 为表达式人工求值时，最后一步进行运行的运算符，它指示了表达式的类型（例如当最后一步是减法运算时，表达式本质上是一个减法表达式）。要正确地对一个长表达式进行分裂，就是要找到它的 dominant operator。其实这算是简单的语法分析阶段，构建一个表达式的语法树，主要思想还是采用分而治之的思想计算表达式的值，这里主要是进行表达式分割，返回最上层运算符的位置 position。主要思路如下所示：

- 遍历给出的表达式，如果该位置 token 非运算符，则继续遍历，否则执行下述规则。
- 如果运算符为左括号，则进入新的循环，该循环是为了把这个括号包含的表达式都吞掉，因为这里不会出现 dominant operator
- 若为普通运算符，则记录运算符，并不断与新运算符比较，得出优先级最小的运算符，最终得到的即为 dominant operator, 返回其位置 position

具体代码如下所示：

寻找 dominant operator

```

1  int find_dominant_operator(int p, int q) {
2      int i = 0, j, cnt;
3      int op = 0, opp, pos = -1;
4      for (i = p; i <= q; i++){
5          if (tokens[i].type == NUM || tokens[i].type == REG || tokens[i].type == HEX)
6              continue;
7          else if (tokens[i].type == LBRACKET) {
8              cnt = 0;
9              int bra=1;
10             for (j = i + 1; j <= q; j++) {
11                 if (tokens[j].type == RBRACKET) {
12                     cnt++;
13
14                     bra--;
15                     if(bra==0)
16                     {
17                         i += cnt;
18                         break;
19                     }
20                 }
21             else if(tokens[j].type == LBRACKET){

```

```

22         cnt++;
23         bra++;
24     }
25     else
26         cnt++;
27     }
28 }
29 else {
30     opp = priority(i);
31     if (opp >= op) {
32         pos = i;
33         op = opp;
34     }
35 }
36 }
37 return pos;
38 }

```

至此寻找最后一个运算符的函数已经书写完毕。

5.2.3 扩展表达式求值

一些较为复杂的符号例如逻辑运算符 &&, !=, 我在上文中其实已经写明了, 已经实现, 这里不再赘述, 主要说一下增加一个负数和指针的用法, 也就是用 “-” 和 “*”, 如果代表不同的含义, 这里就不能只用 SUB 和 MUL 来表示两个符号, 因此我们还需要做以下的修改。

- 符号的枚举中, 需要加入 MINUS, 和 POINTER 标识
- 在运算符优先级的判断中, 需要加入两者, 这俩都是一元运算符, 因此优先级会更高
- 在 eval() 函数中, 由于之前并没有进行这两者的区分, 都统一识别为了 SUB 和 MUL, 进行运算前, 需要再遍历一下表达式, 改变 token type。
- 新加入运算符, 也需要修改主符号函数的查找, 需要对一元运算符进行处理
- 指针进引用问题, 将识别到是指计解引用的 “重新定义为 POINTER, 在 eval 函数中, 若碰到 tokens[i].type= Deref 时, 将后面的十六进制数读取出来, 传入之前扫描内存的函数中, 并清空 tokens[i+1].str, 将地址放入。修改 tokens 的类型, 继续下面的计算。

具体代码实现如下:

加入负号和指针

```

1 for (i = 0; i < nr_token; i++) {
2     //printf("token%d = %s\n", i, tokens[i].str);
3     if (tokens[i].type == '*' && (i == 0 || (tokens[i - 1].type != NUM && tokens[i - 1].type != HEX && tokens[i - 1].type != REG && tokens[i - 1].type != ')')) {
4         tokens[i].type = POINTER;
5     }
6     if (tokens[i].type == '-' && (i == 0 || (tokens[i - 1].type != DEX && tokens[i - 1].type != HEX && tokens[i - 1].type != REG && tokens[i - 1].type != ')')) {

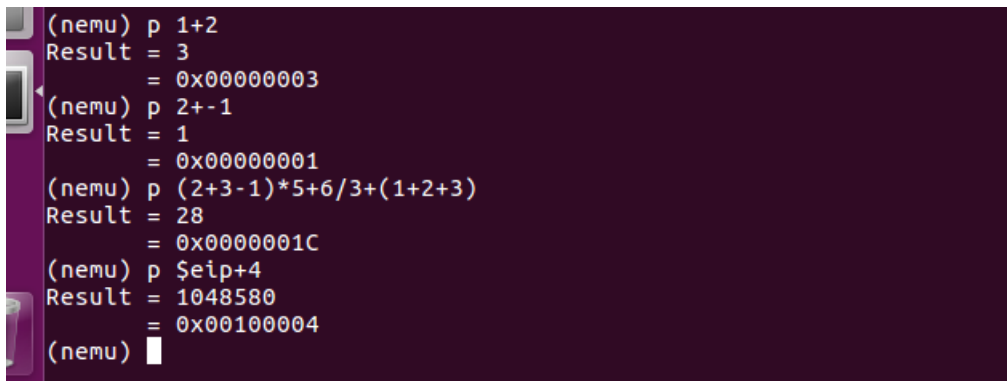
```

```

7     tokens[i].type = MINUS;
8 }
9
10 }
11 int priority(int i) {
12 //新加入代码
13     if (tokens[i].type == MINUS || tokens[i].type == POINTER) return 2;
14 }

```

整体 eval() 函数的结构跟实验指导书中提出的一样，运行结果如下所示：



```

(nemu) p 1+2
Result = 3
        = 0x00000003
(nemu) p 2+-1
Result = 1
        = 0x00000001
(nemu) p (2+3-1)*5+6/3+(1+2+3)
Result = 28
        = 0x0000001C
(nemu) p %eip+4
Result = 1048580
        = 0x00100004
(nemu) 

```

图 5.6: p 指令运算结果

6 阶段三

6.1 监视点链表维护

6.1.1 申请新监视点

这里主要是需要完善 wp_new() 函数，从 free_ 链表中返回一个空闲的监视点结构。如果 free_ 链表为空，使用 assert(0) 报错，若不为空，则从 free_ 链表中取出一个空闲监视点，加入 wp 链表中，具体代码如下：

申请新监视点

```

1 WP* new_wp() {
2     if (free_ == NULL) {
3         assert(0);
4     }
5
6     WP *wp = free_;
7     free_ = free_->next;
8     wp->next = NULL;
9
10    return wp;
11 }

```

6.1.2 释放监视点

这里需要执行的操作时，清空 wp 里的内容，并且把 wp 链表重新加入 free_ 链表中，程序较为简单，这里不再进行代码展示

6.1.3 插入新监视节点

这里需要先从 free_ 链表申请一个新的监视节点，并且把该节点插入到 wp 链表中，这里需要又对 wp 链表的序列号的操作，具体思路如下：

- 申请一个新的监视节点，计算 expr 值，初始化该监视节点的 value 字段
- 如果现在 wp 链表为空，那么直接复制 wp 的序号为 1 即可，并且将该节点赋值给 head
- 如果 wp 链表不为空，那么一直遍历 wp 链表到链表尾部，进行序号的赋值，并把新申请的 wp 节点加入链表。

具体代码如下：

插入新监视点

```
1  void insert_wp(char *args) {
2  bool flag = true;
3  uint32_t val = expr(args, &flag);
4
5  if (!flag) {
6      printf("You input an invalid expression, failed to create watchpoint!");
7      return ;
8  }
9
10 WP *wp = new_wp();
11 wp->value = val;
12 strcpy(wp->exp, args);
13
14 if (head == NULL) {
15     wp->NO = 1;
16     head = wp;
17 }
18 else {
19     WP *wwp;
20     wwp = head;
21     while (wwp->next != NULL) {
22         wwp = wwp->next;
23     }
24     wp->NO = wwp->NO + 1;
25     wwp->next = wp;
26 }
27
28 return ;
29 }
```

6.1.4 删除对应序号的监视点

这个操作很简单，找到要删除的序号对一个的监视点，把它从 wp 链表中删除，并把该链表又重新加入 free_ 链表中，但是这里需要注意到是，如果要删除的是 head 头节点，需要重新对头结点进行处理，要修改头节点为头节点的下一个节点。具体代码如下

删除对应序号的监视点

```
1 void delete_wp(int no) {
2
3     if (head == NULL) {
4         printf("There is no watchpoint to delete!");
5         return ;
6     }
7
8     WP *wp;
9     if (head->NO == no) {
10        wp = head;
11        head = head->next;
12        free_wp(wp);
13    }
14    else {
15        wp = head;
16        while (wp->next != NULL && wp->next->NO != no) {
17            wp = wp->next;
18        }
19        if (wp == NULL) {
20            printf("Failed to find the NO.%d watchpoint!", no);
21        }
22        else {
23            WP *del_wp;
24            del_wp = wp->next;
25            wp->next = del_wp->next;
26            free_wp(del_wp);
27            printf("NO.%d watchpoint has been deleted!\n", no);
28        }
29    }
30
31    return ;
32 }
```

6.1.5 打印所有的监视点

这里只需要遍历整个 wp 链表，若不为空，则打印信息，如果头节点也为空，打印监视点为空信息杰克，这里也不再进行代码展示。

6.2 监视点管理

上述已经实现了基本的链表操作函数，这里如果要实现 `cmd_w()`, `cmd_d()`, `info w` 的操作，其实只需要调用函数即可，`cmd_w()` 通过对表达式也即是传进来的参数的处理，调用 `insert_wp()` 函数完成，`cmd_d()` 直接调用 `delete_wp()` 函数即可，`info w` 调用监视点打印函数完成操作，这里需要注意的是，我们对于监视点完成的功能是当监视点表达式值发生变化时，我们需要给出变化信息，因此我们还需要在 `cpu-exec.c` 文件中的 `cpu_exec()` 函数给出相应的监视点检测函数，具体代码如下：

监视点检测

```

1  #ifdef DEBUG
2  /* TODO: check watchpoints here. */
3  int *no = haschanged();
4  if (*no != -1) {
5      int i;
6      for (i = 0; *(no + i) != -1; i++) {
7          printf("NO.%d ", *(no + i));
8      }
9      printf("watchpoint has been changed\n");
10     nemu_state = NEMU_STOP;
11 }
12 #endif

```

上述代码的思路是：

- 通过 `haschanged()` 函数遍历监视点链表，判断该监视点的值是否发生了变化，如果发生了变化，则将该监视点的序号加如 `int` 型的 `no` 数组，最终返回此数组。
- `no[i] = -1` 是数组结束的标志，这里就是遍历上有给出的数组，打印监视点信息。
- 监视点信息一旦发生变化，也需要阻止程序继续执行，所以这里还需要更改 `nemu_state`

具体运行结果如下图所示：

```

(nemu) w $eip
(nemu) info w
NO      expression      value
1      $eip             1048576
(nemu) w $ebp
(nemu) info w
NO      expression      value
1      $eip             1048576
2      $ebp             697804928
(nemu) d 1
(nemu) info w
NO      expression      value
2      $ebp             697804928
(nemu)

```

图 6.7: 监视点打印结果

至此监视点的全部操作已经完成

7 遇到的问题和解决办法

内存打印函数编写

在编写内存打印函数也即 `cmd_x()` 函数过程中, 因为这个函数与 `help c` 函数不太一样, 他的参数较多, 初始读参数过程中就遇到一些问题, 后来通过查找资料, 更加熟悉了 `strtok()` 函数的用法, 这个问题也就解决了。另外就是在读内存过程中, 其中并没有意识到第二个参数是十六进制字符串, 并不能直接转化为 `int` 型, 经过内存对比, 发现结果有误, 返回查找, 是这里强制类型转换的问题。

表达式求值阶段

表达式求值过程中, 遇到的最大问题就是很多情况没有考虑清楚。且最开始题目中已经给好了架构, 对于类似 $(1+2)*(3+4)$ 的类型判断进行括号匹配时判断为不合法, 这里实验指导书中也给出了说明, 通过网上查找其他人的解决办法, 就有新增了一个括号合法判断问题, 只在初始判断一次即可, 就解决了后面的不合法问题。除此之外, 对于指针的操作, 由于之前并未实现指针操作, 这里基本是按照负数的操作实现的, 特殊的是, 指针这里与内存相关, 除此之外, 我还需要加入内存的相关操作, 这里在编写代码过程中也遇到了一些问题, 通过借鉴一些人的思路实现了指针的操作。

监视点打印

这个阶段基本函数编写的时候就有点无从下手, 主要是不熟悉 `gdb` 的监视点打印功能, 并不知道该如何使用, 具体参数都有什么, 输出什么, 后来通过 `gdb` 使用手册的学习, 解决了基本函数编写问题。后来函数编写过程中, 监视点变化, 但是并没有打印信息, 起初并没有意识到哪里的问题, 感觉基本函数并没有错, 发现是关于监视点函数的判断有误, 导致一直不能正确打印信息。

8 必答题

1. 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?

`cpu_exec()` 函数接收的参数是 `uint32_t`, 接收的是无符号数, 整数使用补码表示, -1 的补码为所有位全 1, 因此无符号情况下 -1 为最大值, 实际意思也就是继续运行程序, 执行所有指令。

2. 谁来指示程序的结束? 为什么程序执行到 `main()` 函数的返回处就结束了?

`exit()` 函数用于结束正在运行的整个程序, 它将参数返回给 OS, 把控制权交给操作系统, 在 `main()` 函数执行 `return` 语句时, 会隐式地调用 `exit()` 函数, 所以一般程序执行到 `main()` 结尾时, 则结束主进程。

3. 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?


`wp_pool` 等变量是实现监视点的关键变量, 存储监视点信息, 只有特定操作才能进行更改。在定义变量时使用 `static` 关键字, 会将变量的作用域限定在当前文件中, 即使另一个文件中使用 `extern` 关键字来声明其他文件中存在的静态全局变量也不能使用。保证监视点池的直接操作只会在 `watchpoint.c` 中完成, 其他文件只能通过函数调用方式间接操作监视点池。同时, `static` 关键字还会改变变量的存储方式, 使得它在程序运行期间始终占用同一块内存空间, 确保变量在整个程序运行期间都存在且不会被意外修改。

4. 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?

不能，因为 `int3` 的工作机制为用断点替换任何指令的第一个字节，包括其他单字节指令，而不会覆盖其他代码。如果 `int3` 指令的长度变为 2 个字节，且被替换的指令长度仅有一个字节的话，`int3` 指令就会因为空间不够被迫覆盖下一条指令的一部分，导致程序执行出现问题。

5. 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?

当 GDB 尝试在错误的地址上设置断点时，它实际上是在那个地址上插入了一个断点指令。如果该地址不是指令的开头，那么断点指令可能会被插入到指令的中间，覆盖一部分原始指令。这可能会导致程序的行为不可预测，因为指令的剩余部分可能已经被破坏了。



```
(gdb) info break
Num    Type             Disp Enb Address      What
1      breakpoint      keep y   0x080485ec in main() at transform.cpp:4
2      breakpoint      keep y   0x080485ed in main() at transform.cpp:4
(gdb) r
Starting program: /home/gyk/Desktop/tranfform

Breakpoint 1, main () at transform.cpp:4
4      int a=0;
(gdb) si
Terminal: b=1;
(gdb) si
6      int c=a+b;
(gdb) b *0x0804860a
Breakpoint 3 at 0x0804860a: file transform.cpp, line 7.
(gdb) si
0x080485fd      6      int c=a+b;
(gdb) si
0x08048600      6      int c=a+b;
(gdb) si
0x08048602      6      int c=a+b;
(gdb) si
7      cout<<c;
(gdb)
```

图 8.8: gdb 运行结果

如图所示，第二个断点打在指令中间，并没有被检测到，第三个断点成功引起了中断，程序的行为不可预测。

6. 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?

模拟器 (Emulator) 和调试器 (Debugger) 是两种不同的工具。模拟器通常用于模拟硬件或软件环境，允许在模拟环境中运行程序，并且可以使用特定于模拟器的工具进行调试。调试器则是一种专门用于调试程序的工具，通常允许用户暂停程序的执行并检查程序状态，以帮助找出错误。

NEMU 是一个模拟器，它模拟了一个计算机的硬件环境，包括 CPU、内存、寄存器等。它可以在模拟环境中运行程序，并允许用户使用调试命令来检查程序状态，例如设置断点、单步执行、查看寄存器状态等。与传统的调试器不同，NEMU 还可以模拟整个系统，包括操作系统和外部设备，这使得它可以用于操作系统和嵌入式系统的开发和调试。

GDB 是一个常见的调试器，它可以用于调试 C、C++、汇编等程序。与 NEMU 不同，GDB 不是一个模拟器，它不能模拟整个系统。相反，它在运行程序时使用操作系统提供的调试接口，以便可以访问程序的内存和寄存器状态。用户可以使用 GDB 命令来控制程序的执行，例如设置断点、单步执行、查看内存和寄存器状态等。GDB 也可以与其他工具集成，例如 GCC 编译器，以方便用户在编译和调试之间切换。

GDB 是一个用户程序，通过系统调用来获得运行指定程序的进程的指令空间、寄存器值等。而 NEMU 为运行在其上的用户程序提供虚拟硬件平台，因此 nemu 可以直接访问用户程序的寄存器和内存等相关信息。

7. 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？

位置 “2.3.4.1 Status Flags” 下 p34，意思为进位标志。

- ModR/M 字节是什么？

位置 “17.2.1 ModR/M and SIB Bytes” 下 p241, 242

ModR/M 字节由 mod 字段、reg 字段、r/m 字段组成，用于表示要在指令中使用的索引类型或寄存器号，要使用的寄存器等信息，其中 mod 字段为最高两位，提供寄存器和索引模式信息；reg 字段为 mod 字段后三位，指定寄存器号或补充操作码信息（reg 字段的含义由指令的第一个操作码字节决定）；r/m 字段为最低三位，与 mod 字段组合指示寄存器或索引模式。

- mov 指令的具体格式是怎么样的？

位置 “17.2.2.11 Instruction Set Detail” 下 p345-p347

格式：DEST ← SRC。

8. shell 命令完成 PA1 的内容之后，nemu/目录下的所有.c 和.h 和文件总共有多少行代码？你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？

命令：find . -name " *.[h|.c]" | xargs wc -l

总代码：4023 编写代码：497 行

除去空行后总共有 3292 行。（命令为 find . -name " *.[h|.c]" | xargs cat | grep -Ev "\$" | wc -l）

9. 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

-Wall 启用所有警告。它会输出所有警告信息，包括一些常规的警告，如未定义的函数、不兼容的指针类型等。-Werror 将所有警告视为错误。当使用这个选项时，编译器会将所有警告视为错误，如果有任何一个警告，编译器就会停止编译并返回错误信息。是用这两个选项可以帮助我们在编译时检测到代码中的潜在问题，防止在之后的编写中因前置代码的潜在漏洞导致问题发生。