PA1 Report

- 姓名：陈睿颖
- 学号：2013544
- 专业：计算机科学与技术

1. 实验目的

1. 熟悉 GNU/Linux 平台
2. 初步探究“程序在计算机上运行”的相关原理
3. 初步学习 GDB 并在 PA 上实现简易寄存器

2. 实验内容

本次实验可以分为三个阶段：

1. 模拟寄存器结构，实现调试器基本功能。
2. 实现调试功能的表达式求值，并完善阶段一中的扫描内存函数。
3. 实现调试功能中的监视点，学习断点相关知识与 i386 手册。

3. 阶段一

3.1 实现正确寄存器结构体

除了程序寄存器 eip，还有 32 位寄存器、16 位寄存器、8 位寄存器各8个，且物理结构相互关联。源代码中仅仅使用了 Struct，这不能体现寄存器物理结构关联的特性， 需要使用 Union 结构。Union 中的各个变量互斥，共享同一内存地址。 源代码中仅仅使用了 Struct，下面使用 Union 结构。因为 Union 中的各个变量互斥，共享同一内存地址。

在nemu\include\cpu\reg.h中修改的具体代码如下：

```
/*仅展示部分修改的代码*/
typedef struct {
  union{
    union{
        uint32_t _32;
        uint16_t _16;
        uint8_t _8[2];
    } gpr[8];

    /* Do NOT change the order of the GPRs' definitions. */

    /* In NEMU, rtlreg_t is exactly uint32_t. This makes RTL instructions
     * in PA2 able to directly access these registers.
     */
    struct
    {
      rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
    };

  };

  vaddr_t eip;

} CPU_state;
```

运行结果如下：

```
user@user-VirtualBox:~/ics2017/nemu$ make run
+ CC src/memory/memory.c
+ CC src/monitor/diff-test/diff-test.c
+ CC src/monitor/monitor.c
+ CC src/monitor/debug/ui.c
+ CC src/monitor/debug/expr.c
+ CC src/monitor/cpu-exec.c
+ CC src/cpu/decode/decode.c
+ CC src/cpu/decode/modrm.c
+ CC src/cpu/reg.c
+ CC src/cpu/exec/special.c
+ CC src/cpu/exec/arith.c
+ CC src/cpu/exec/cc.c
+ CC src/cpu/exec/exec.c
+ CC src/cpu/exec/prefix.c
+ CC src/cpu/exec/system.c
+ CC src/cpu/exec/logic.c
+ CC src/cpu/exec/data-mov.c
+ CC src/cpu/exec/control.c
+ CC src/cpu/intr.c
+ LD build/nemu
./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: 16:03:46, Mar 15 2023
For help, type "help"
(nemu) q
```

3.2 实现单步执行 si、打印寄存器 info、扫描内存

首先在 cmd_table 结构中加入需要实现的函数解析命令，使得根据输 入不同的命令执行相应的功能。加入“命令-描述信息-处理函数”，加入“si”、“info”、“x” 命令，分别对应了单步执行、打印寄存器、扫描内存的功能。

在nemu\src\monitor\debug\ui.c中加入这三条命令：

```
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    /* TODO: Add more commands */
10   { "si", "Let the program execute N instructions step by step", cmd_si },
11   { "info", "Print registers' status for r, checkpoint informations for w", cmd_
12   { "x", "Scan the consecutive 4N bytes from Address EXPR", cmd_x }
13
14
15
16 };
```

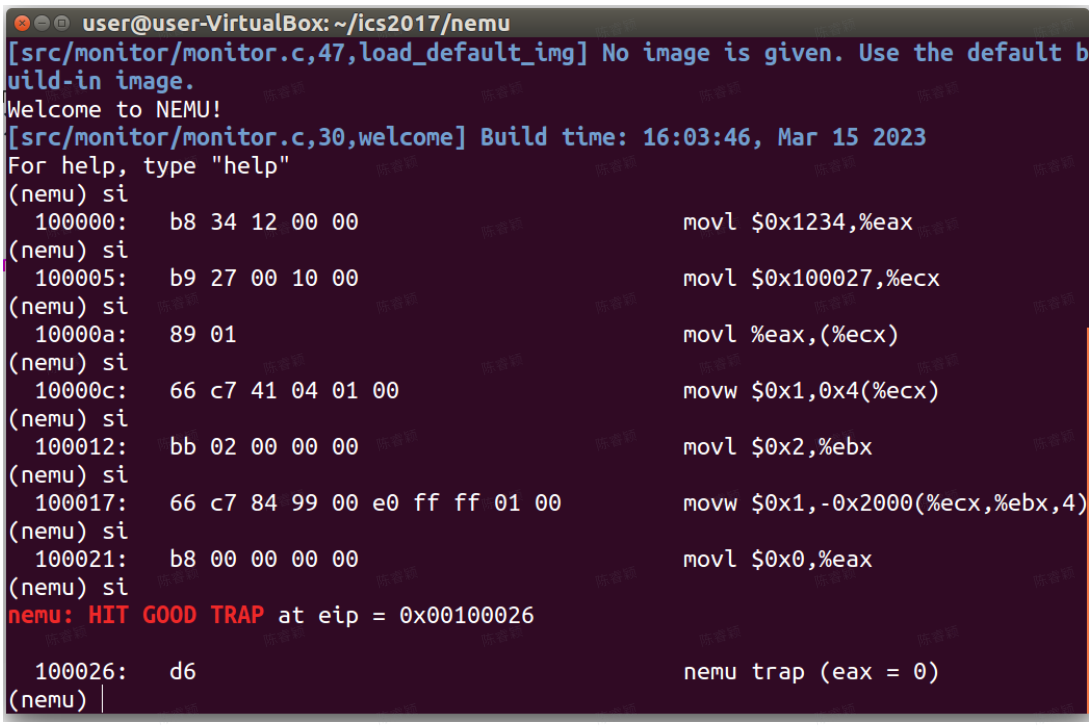
其他在c文件中添加的函数定义及声明的代码已省略

3.2.1 单步执行

调用 cpu_exec(uint64_t n) 函数即可执行 n 条指令，代码如下：

```c
static int cmd_si(char *args){
  char *arg = strtok(NULL, " ");
  if(arg!=NULL){
    cpu_exec(atoi(arg));
  }
  else{
    cpu_exec(1);
  }
  return 0;
}
```

还未实现的函数先直接 `return 0` ，直接编译运行验证此步骤的正确性。



可以看到程序输出了 `nemu: HIT GOOD TRAP at eip = 0x00100026` ，说明客户程序已经成功地结束运行.退出 `cpu_exec()` 之后,NEMU 将返回到 `ui_mainloop()` ，等待用户输入命令。

3.2.2 打印寄存器

执行 info r 之后，直接用 printf()输出所有寄存器的值即可。调用 reg_l(index 函数) 访问数组 regs[i] 所表示的寄存器，再打印即可。

```c
static int cmd_info(char *args){
```

```
  char *arg = strtok(NULL, " ");
  if(arg==NULL){
    printf("args error in cmd_info\n");
    return 0;
  }
  char s;
  int nRet = sscanf(args, "%c", &s);
  if(nRet<=0 || s=='w'){
    printf("args error in cmd_info\n");
    return 0;
  }
  if(s == 'r'){
    int i;
    for(i=0;i<8;i++){
      printf("%s        0x%x\n", regsl[i], reg_l(i));
    }
    printf("eip        0x%x\n", cpu.eip);
    for(i=0;i<8;i++){
      printf("%s        0x%x\n", regsw[i], reg_w(i));
    }
    for(i=0;i<8;i++){
      printf("%s        0x%x\n", regsb[i], reg_b(i));
    }
  }
  return 0;
}
```

运行结果如下：

```
user@user-VirtualBox:~/ics2017/nemu$ make run
+ CC src/monitor/debug/ui.c
+ LD build/nemu
./build/nemu -l ./build/nemu-log.txt
[src/monitor/monitor.c,47,load_default_img] No image is given. Use the defaul
t build-in image.
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 16:03:46, Mar 15 2023
For help, type "help"
(nemu) info r
eax        0x1965cb91
ecx        0x709abab
edx        0x9f7ee67
ebx        0x7b459932
esp        0xc7610f5
ebp        0x39802ae5
esi        0x1cb54950
edi        0x5b3bc1d8
eip        0x100000
ax        0xcb91
cx        0xabab
dx        0xee67
bx        0x9932
sp        0x10f5
bp        0x2ae5
si        0x4950
di        0xc1d8
al        0x91
cl        0xab
dl        0x67
bl        0x32
ah        0xcb
ch        0xab
dh        0xee
bh        0x99
(nemu)
```

3.2.3 扫描内存

对命令进行解析之后,先求出表达式的值。但还没有实现表达式求值的功能，现在可以先实现一个简单的版本:规定表达式 EXPR 中只能是一个十六进制数,例如 x 10 0x100000。解析出待扫描内存的起始地址之后,使用循环将指定长度的内存数据通过十六进制打印出来。调用 `vaddr_read(vaddr_t addr, int len)` 函数访问内存。

代码如下：

```
static int cmd_x(char *args){
   char *arg1 = strtok(NULL, " ");
  if(arg1==NULL){
    printf("u shall input the parameter N to specify the consecutive N..\n");
    return 0;
  }
  int i_arg1 = atoi(arg1);
  char *arg2 = strtok(NULL, " ");
  if(arg2==NULL){
    printf("u shall input the parameter EXPR must generate from keyboard input..
    return 0;
  }
  uint32_t addr_begin = strtoul(arg2,NULL,16);
  int i;
  for(i=0;i<i_arg1;i++){
    printf("0x%x ", vaddr_read(addr_begin,1));
    addr_begin+=1;
  }
  printf("\n");
  return 0;

}
```

运行结果如下：

```
[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: 17:40:55, Mar 15 2023
For help, type "help"
(nemu) x 10 0x100000
0xb8 0x34 0x12 0x0 0x0 0xb9 0x27 0x0 0x10 0x0
(nemu) x 39 0x100000
0xb8 0x34 0x12 0x0 0x0 0xb9 0x27 0x0 0x10 0x0 0x89 0x1 0x66 0xc7 0x41 0x4 0x1 0x0 0xbb 0x2 0x0 0x0 0x0 0x66 0xc7 0x84 0x99 0x0 0xe0 0xff 0xff 0x1 0x0 0xb8 0x0
0x0 0x0 0x0 0xd6
```

对比默认镜像：

```
static inline int load_default_img() {
  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
  };
```

正确!

4. 阶段二

在此阶段开始之前，首先要添加表达式求值的命令，在ui.c中：

```
1  static int cmd_p(char *args){
2    char *arg = strtok(NULL," ");
3    if(arg==NULL){
4      printf("please input the expression u wanna calculate..!\n");
5      return 0;
6    }
7    bool is_finish=true;
8    uint32_t ans = expr(arg,&is_finish);
9    if(!is_finish){
10     printf("please check your expression's format..!\n");
11   }
12   else{
13     printf("%d\n", ans);
14   }
15   return 0;
16 }
17
```

在接收到命令时，调用 `expr()` 进行表达式求值。

4.1 词法分析

在nemu\src\monitor\debug\expr.c中，要完善token的添加，在枚举类型中添加：

```
enum {
  TK_NOTYPE = 256,
  TK_EQ, //equal
  TK_OR, // |
  TK_AND, //&
  TK_NEQ, //!=
  TK_LOGAND, // &&
  TK_LOGOR, // ||
  TK_NOT, // ~
  TK_DEC, // 10
  TK_HEX, // 16
  TK_REG, // register
  TK_GETVAL // get value

  /* TODO: Add more token types */

};
```

根据编译原理中的知识，切分并且识别每一个 token，为算术表达式中的各种token类型添加规则：

```
static struct rule {
  char *regex;
  int token_type;
} rules[] = {

  /* TODO: Add more rules.
   * Pay attention to the precedence level of different rules.
   */

  {" +", TK_NOTYPE},     // spaces
  {"!", TK_NOT},         // log-not
  {"\\*", '*'},          // multi/getval
  {"\\/", '/'},          // div
  {"\\+", '+'},          // plus
  {"\\-", '-'},          // minus
  {"\\|\\|", TK_LOGOR},      // log-or
  {"&&", TK_LOGAND},         // log-and
  {"\\|", TK_OR},        // calc-or
  {"&", TK_AND},         // calc-and
  {"==", TK_EQ},         // equal
  {"!=", TK_NEQ},        // not-equal
  {"\\$[eE][0-9a-zA-Z]{2}", TK_REG}, // registers
  {"0[xX][a-fA-F0-9]+", TK_HEX}, // num with radix 16
  {"[0-9]|([1-9][0-9]*)", TK_DEC}, // num with radix 10
  {"\\(", '('},          // l-parentheses
```

```
  {"\\)", ')'},          // r-patentheses
};
```

在成功识别出 token 后,将 token 的信息依次记录到 token数组中。

4.2 表达式求值

使用算数表达式的递归求值思路。在此之前需要对输入的表达式的合法性进行判断，并不是所有的表达式都可以求值，例如 $((2-6)\times 9$ 就是非法的表达式形式，因为括号并不匹配。因此需要先判断表达式是否被一对匹配的括号包围，同时检查表达式的左右括号是否匹配。使用 `check_parentheses(int p,int q)` 函数完成上述工作，实现如下：

```
bool check_parentheses(int l, int r, bool *success){
  if(!check_pre_valid(l,r)){
    *success=false;
    return *success;
  }
  if(tokens[l].type=='('&&tokens[r].type==')'){
    if(check_pre_valid(l+1,r-1)){
      return true;
    }
  }
  *success=true;
  return false;
}
```

其中使用了函数 `check_pre_valid` 其作用是检查表达式是否符合括号匹配的规则，函数接收两个参数l和r，表示需要检查的字符串的左右边界。代码首先定义一个变量tot用于记录左右括号的数量差，接着从左到右扫描字符串，若当前字符为左括号，则tot加一，若当前字符为右括号，则tot减一。若tot小于零，说明此时右括号数量多于左括号，无法匹配，直接返回false。最后如果tot等于零，则说明匹配成功，返回true：

函数 `eval()` 实现了表达式的求值，其参数 `start` 和 `end` 表示表达式的起始和结束位置，`tokens` 是一个结构体数组，表示分词后的表达式：

1. 如果 `start>end` ，则表示表达式非法，抛出异常。
2. 如果 `start==end` ，则表示该位置是一个数字、寄存器或其他运算符，进行相应的处理并返回结果。
3. 如果表达式被括号包裹，去掉括号后递归求解表达式的值。

4. 如果表达式中存在多个运算符，则根据运算符优先级找到最优运算符，将表达式分成两部分递归求解并计算结果。

函数 `eval()` 的具体实现则不在此展示。

运行结果如下：

```
[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: 17:00:21, Mar 17 2023
For help, type "help"
(nemu) p 1+3*6-2
[src/monitor/debug/expr.c,97,make_token] match rules[2] = "[0-9]+" at position 0 with len 1: 1
[src/monitor/debug/expr.c,97,make_token] match rules[9] = "\+" at position 1 with len 1: +
[src/monitor/debug/expr.c,97,make_token] match rules[2] = "[0-9]+" at position 2 with len 1: 3
[src/monitor/debug/expr.c,97,make_token] match rules[6] = "\*" at position 3 with len 1: *
[src/monitor/debug/expr.c,97,make_token] match rules[2] = "[0-9]+" at position 4 with len 1: 6
[src/monitor/debug/expr.c,97,make_token] match rules[7] = "-" at position 5 with len 1: -
[src/monitor/debug/expr.c,97,make_token] match rules[2] = "[0-9]+" at position 6 with len 1: 2
17
(nemu) p $eax+1
[src/monitor/debug/expr.c,97,make_token] match rules[1] = "[$][a-z,A-Z]+" at position 0 with len 4: $eax
[src/monitor/debug/expr.c,97,make_token] match rules[9] = "\+" at position 4 with len 1: +
[src/monitor/debug/expr.c,97,make_token] match rules[2] = "[0-9]+" at position 5 with len 1: 1
1571985179
(nemu)
```

5. 阶段三

在此之前，要完善 `w` 和 `d` 的命令函数，实现思路为：

函数 `cmd_w`：首先检查是否输入了表达式，如果没有则提示重新输入，否则申请一个空闲的监视点并将表达式复制到该监视点中。然后，使用表达式求值函数 `expr()` 计算表达式的值，并将该值记录在监视点中，同时打印监视点的编号和已设置的消息。

函数 `cmd_d`：首先检查是否输入了要删除的监视点的编号，如果没有则提示重新输入，否则根据输入的编号找到相应的监视点并释放该监视点，同时打印已删除的监视点的编号。

此外，在 `info` 命令中，需要另写一个打印监视点的函数，在识别到命令 `w` 时直接调用即可。

代码具体实现在此不再赘述。

5.1 监视点

监视点的作用是监视一个表达式的值何时发生变化。实际上需要实现一个链表，在cpu执行每一步操作之后，需要对当前表达式进行求值检查。

`new_wp()` 函数实现的是创建一个新的监视点的功能。首先，函数会检查当前是否有空闲的监视点，如果没有则会终止程序。接着，从空闲监视点池中获取一个空闲的监视点 `temp`，并将其从空闲监视点池中移除。然后，将该监视点 `temp` 添加到监视点链表 `head` 的末尾。最后将该监视点的下一个节点指向空，表示该监视点为链表的最后一个节点，返回该监视点的指针 `temp`。

```
1 WP* new_wp()
2 {
3     if(free_ == NULL) {
```

```
        assert(0);
    }

    WP *temp = free_;
    free_ = free_->next;

    if(head == NULL) {
        head = temp;
    } else {
        WP *search_head = head;
        while(search_head->next != NULL) {
            search_head = search_head->next;
        }
        search_head->next = temp;
    }

    temp->next = NULL;
    return temp;
}
```

`free_wp()` 函数实现的是释放一个监视点的功能。给定要释放的监视点 `wp` 的 ID，首先获取该监视点的指针 `wp`。如果该监视点是链表的第一个节点，则将链表头指向该节点的下一个节点；否则，在链表中搜索该监视点的前一个节点，并将其指向该监视点的下一个节点。接着，将该监视点的下一个节点指向空，并将该监视点添加到空闲监视点池的末尾。

```
void free_wp(int wpid)
{
    WP* wp = &wp_pool[wpid];
    WP *temp = head;

    if(temp == wp) {
        head = wp->next;
    } else {
        while(temp->next != wp) {
            temp = temp->next;
        }
        temp->next = wp->next;
    }

    wp->next = free_;
    free_ = wp;
}
```

5.2 断点

断点的功能是让程序暂停下来,从而方便查看程序某一时刻的状态.事实上,我们可以很容易地用监视点来 模拟断点的功能: `w $eip==ADDR` 其中 ADDR 为设置断点的地址.这样程序执行到 ADDR 的位置时就会暂停下来。

具体实现如下:

```
bool check_watchpoint()
{
        WP *wp=head;
        bool* success_flag=malloc(1);
        bool ret=true;
        while(wp!=NULL)
        {
                uint32_t new_value=expr(wp->expr,success_flag);
                if(new_value!=wp->expr_record_val){
                        printf("\nHit watchpoint %d  old value = 0x%x, new value = 0x%x .\n"
                                ,wp->NO,wp->expr_record_val,new_value);
                        ret=false;
                        goto end;
                }
                wp=wp->next;
        }
end:

        return ret;
}
```

运行结果:

```
[src/monitor/monitor.c,30,welcome] Build time: 16:37:02, Mar 20 2023
For help, type "help"
(nemu) w $eip==0x100000
[src/monitor/debug/expr.c,97,make_token] match rules[1] = "[$][a-z,A-Z]+" at position 0 with len 4: $eip
[src/monitor/debug/expr.c,97,make_token] match rules[10] = "==" at position 4 with len 2: ==
[src/monitor/debug/expr.c,97,make_token] match rules[0] = "[0][x]([0-9,a-f,A-F]){1,8}" at position 6 with len 8: 0x100000
watchpoint 0 is set
(nemu) info w
WID     expr            record value
0       $eip==0x100000          0x1
(nemu) c
[src/monitor/debug/expr.c,97,make_token] match rules[1] = "[$][a-z,A-Z]+" at position 0 with len 4: $eip
[src/monitor/debug/expr.c,97,make_token] match rules[10] = "==" at position 4 with len 2: ==
[src/monitor/debug/expr.c,97,make_token] match rules[0] = "[0][x]([0-9,a-f,A-F]){1,8}" at position 6 with len 8: 0x100000

Hit watchpoint 0  old value = 0x1, new value = 0x0 .
[src/monitor/debug/expr.c,97,make_token] match rules[1] = "[$][a-z,A-Z]+" at position 0 with len 4: $eip
[src/monitor/debug/expr.c,97,make_token] match rules[10] = "==" at position 4 with len 2: ==
[src/monitor/debug/expr.c,97,make_token] match rules[0] = "[0][x]([0-9,a-f,A-F]){1,8}" at position 6 with len 8: 0x100000

Hit watchpoint 0  old value = 0x1, new value = 0x0 .
[src/monitor/debug/expr.c,97,make_token] match rules[1] = "[$][a-z,A-Z]+" at position 0 with len 4: $eip
[src/monitor/debug/expr.c,97,make_token] match rules[10] = "==" at position 4 with len 2: ==
[src/monitor/debug/expr.c,97,make_token] match rules[0] = "[0][x]([0-9,a-f,A-F]){1,8}" at position 6 with len 8: 0x100000

Hit watchpoint 0  old value = 0x1, new value = 0x0 .
[src/monitor/debug/expr.c,97,make_token] match rules[1] = "[$][a-z,A-Z]+" at position 0 with len 4: $eip
[src/monitor/debug/expr.c,97,make_token] match rules[10] = "==" at position 4 with len 2: ==
[src/monitor/debug/expr.c,97,make_token] match rules[0] = "[0][x]([0-9,a-f,A-F]){1,8}" at position 6 with len 8: 0x100000

Hit watchpoint 0  old value = 0x1, new value = 0x0 .
[src/monitor/debug/expr.c,97,make_token] match rules[1] = "[$][a-z,A-Z]+" at position 0 with len 4: $eip
[src/monitor/debug/expr.c,97,make_token] match rules[10] = "==" at position 4 with len 2: ==
[src/monitor/debug/expr.c,97,make_token] match rules[0] = "[0][x]([0-9,a-f,A-F]){1,8}" at position 6 with len 8: 0x100000

Hit watchpoint 0  old value = 0x1, new value = 0x0 .
[src/monitor/debug/expr.c,97,make_token] match rules[1] = "[$][a-z,A-Z]+" at position 0 with len 4: $eip
[src/monitor/debug/expr.c,97,make_token] match rules[10] = "==" at position 4 with len 2: ==
[src/monitor/debug/expr.c,97,make_token] match rules[0] = "[0][x]([0-9,a-f,A-F]){1,8}" at position 6 with len 8: 0x100000

Hit watchpoint 0  old value = 0x1, new value = 0x0 .
[src/monitor/debug/expr.c,97,make_token] match rules[1] = "[$][a-z,A-Z]+" at position 0 with len 4: $eip
[src/monitor/debug/expr.c,97,make_token] match rules[10] = "==" at position 4 with len 2: ==
```

```
Hit watchpoint 0  old value = 0x1, new value = 0x0 .
[src/monitor/debug/expr.c,97,make_token] match rules[1] = "[$][a-z,A-Z]+" at position 0 with len 4: $eip
[src/monitor/debug/expr.c,97,make_token] match rules[10] = "==" at position 4 with len 2: ==
[src/monitor/debug/expr.c,97,make_token] match rules[0] = "[0][x]([0-9,a-f,A-F]){1,8}" at position 6 with len 8: 0x100000

Hit watchpoint 0  old value = 0x1, new value = 0x0 .
[src/monitor/debug/expr.c,97,make_token] match rules[1] = "[$][a-z,A-Z]+" at position 0 with len 4: $eip
[src/monitor/debug/expr.c,97,make_token] match rules[10] = "==" at position 4 with len 2: ==
[src/monitor/debug/expr.c,97,make_token] match rules[0] = "[0][x]([0-9,a-f,A-F]){1,8}" at position 6 with len 8: 0x100000

Hit watchpoint 0  old value = 0x1, new value = 0x0 .
nemu: HIT GOOD TRAP at eip = 0x00100026

[src/monitor/debug/expr.c,97,make_token] match rules[1] = "[$][a-z,A-Z]+" at position 0 with len 4: $eip
[src/monitor/debug/expr.c,97,make_token] match rules[10] = "==" at position 4 with len 2: ==
[src/monitor/debug/expr.c,97,make_token] match rules[0] = "[0][x]([0-9,a-f,A-F]){1,8}" at position 6 with len 8: 0x100000

Hit watchpoint 0  old value = 0x1, new value = 0x0 .
(nemu) info w
WID     expr            record value
0       $eip==0x100000          0x1
(nemu) d 0
watchpoint 0 is deleted
(nemu)
```

6. 实验手册思考题

> 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.

当一个C或C++程序启动时，操作系统会创建一个进程，并为程序的代码、数据和堆栈分配内存。然后调用 `main()` 函数，程序从那里开始执行。当 `main()` 函数返回时，程序的退出状态被设置，任何打开的资源，如文件或网络连接被关闭。但是，进程本身继续存在于内存中，直到操作系统终止它。因此，可以在 `main()` 函数返回后继续执行代码，要么通过调用其他函数，要么通过设置信号处理程序来捕获和处理事件，但是，一般不会这么做。

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?

```c
void cpu_exec(uint64_t n) {
  if (nemu_state == NEMU_END) {
    printf("Program execution has ended. To restart the program, exit NEMU and
    return;
  }
  nemu_state = NEMU_RUNNING;

  bool print_flag = n < MAX_INSTR_TO_PRINT;

  for (; n > 0; n --) {
    /* Execute one instruction, including instruction fetch,
     * instruction decode, and the actual execution. */
    exec_wrapper(print_flag);

#ifdef DEBUG
    /* TODO: check watchpoints here. */
    if (check_watchpointsvalue()){
      return;
    }

#endif

#ifdef HAS_IOE
    extern void device_update();
    device_update();
#endif

    if (nemu_state != NEMU_RUNNING) { return; }
  }

  if (nemu_state == NEMU_RUNNING) { nemu_state = NEMU_STOP; }
}

```

`cmd_c` 函数用于实现继续运行被暂停的程序的命令。

函数 `cpu_exec` 接受一个参数 `n`，表示要执行的指令数。如果 `nemu_state` 不等于 `NEMU_END`，那么函数会将其设置为 `NEMU_RUNNING`，并且循环执行 `n` 次指令。当 `n` 小于一个常量 `MAX_INSTR_TO_PRINT` 时，函数会打印每条指令的信息。

当调用 `cpu_exec(-1)` 时，参数 `-1` 会被传递给函数，意味着要执行无限多条指令，直到程序结束或者手动停止。因为 `n` 是无限的，所以 `print_flag` 永远为 false，不会打印每条指令的信息。

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?

`static` 关键字的作用是限定变量的作用域。具体来说，它将变量 `wp_pool`、`head`、`free_` 和 `used_next` 的作用域限制在当前源文件中，这意味着这些变量只能在当前文件中被访问和修改，而不能被其他源文件访问和修改。

这种做法的目的是为了避免命名冲突和符号重定义。由于这些变量是用于实现监视点功能的，如果它们的作用域不被限制，可能会与其他源文件中的变量或函数产生命名冲突。因此，使用 `static` 关键字可以确保这些变量只在当前文件中可见，避免了命名冲突的问题。同时，这也可以防止这些变量被其他源文件修改，确保了程序的正确性和可维护性。

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?

int3指令的长度为一个字节并不是严格必要的，但它是x86架构中长期遵循的约定。如果x86架构的变体，例如my-x86，将int3指令的长度更改为两个字节，它仍然可以使用文章中提到的断点机制，只要调试器做出相应调整即可。然而，改变int3指令的长度可能会有其他影响，例如可能需要更新依赖旧指令长度的现有软件或工具。此外，改变指令长度也可能影响某些应用程序或系统的性能，这取决于它们是如何设计的。 总体而言，虽然断点机制仍然可以使用my-x86中的两字节int3指令运行，但在实施之前还是仔细考虑这种更改所产生影响。

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.

可能会发生以下几种情况：

- 程序会停在设置断点的那条指令上，而不是在断点所在的完整指令上。
- 程序可能会出现错误或异常行为，因为将断点设置在指令的非首字节可能会破坏指令的完整性。例如，如果一个指令被拆分成两部分，那么这两部分可能会被执行在不同的时刻，导致程序的行为不可预测。
- 程序可能会跳过断点并继续执行，因为断点所在的字节不是指令的起始位置，GDB 可能会认为指令并没有被中断，而是继续执行到下一条指令。

> 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?

模拟器是一种软件，用于模拟硬件或其他软件的行为。模拟器通常用于测试、开发和调试软件。例如，计算机模拟器可以模拟一个完整的计算机系统，允许软件工程师在模拟环境中测试和调试软件。

调试器是一种软件，用于帮助程序员在程序运行时查找和解决程序中的错误。调试器通常提供一组工具，如单步执行、断点设置、内存查看等，帮助程序员跟踪程序运行状态，并找出程序中的错误。

NEMU是一个基于QEMU的RISC-V ISA模拟器，用于模拟RISC-V指令集架构的处理器。而GDB是一种调试器，用于在程序运行时查找和解决错误。GDB可以与NEMU结合使用，允许程序员在NEMU运行时进行调试。程序员可以使用GDB设置断点、单步执行、查看内存等工具，以帮助他们找到和解决程序中的错误。

> 假设你现在需要了解一个叫 selector 的概念,请通过 i386 手册的目录确定你需要阅读手册中的哪些地方

5.1.3 Selectors

> 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:
>
> 1. EFLAGS 寄存器中的 CF 位是什么意思?
> 2. ModR/M 字节是什么?
> 3. mov 指令的具体格式是怎么样的?

1. 需要阅读的范围为：2.3.4 Flags Register APPENDIX C STATUS FLAG SUMMARY

   阅读可知，EFLAGS 寄存器的 Cf 位全称为 Carry Flag，是一个状态 标志位。若算术操作的结果在高阶位产生借位或进位，则 CF 位置为 1，否则为0。
2. 需要阅读的范围为：17.2 INSTRUCTION FORMAT

   17.2.1 ModR/M and SIB Bytes 用于变长指令，用于指定待操作的寄存器，是变长指令中的一部分。
3. 需要阅读的范围为：Chapter 17 -- 80386 Instruction Set

   指令的格式如下：

MOV -- Move Data

Opcode	Instruction	Clocks	Description
88 /r	MOV r/m8, r8	2/2	把一个字节大小的寄存器的值移动到一个字节大小的内存地址
89 /r	MOV r/m16, r16	2/2	把一个字大小的寄存器的值移动到一个字大小的内存地址
89 /r	MOV r/m32, r32	2/2	把一个双字大小的寄存器的值移动到一个双字大小的内存地址
8A /r	MOV r8, r/m8	2/4	把一个字节大小的内存地址的值移动到一个字节大小的寄存器
8B /r	MOV r16, r/m16	2/4	把一个字大小的内存地址的值移动到一个字大小的寄存器
8B /r	MOV r32, r/m32	2/4	把一个双字大小的内存地址的值移动到一个双字大小的寄存器
8C /r	MOV r/m16, Sreg	2/2	把一个段寄存器的值移动到一个字大小的内存地址
8E /r	MOV Sreg, r/m16	2/5, pm=18/19	把一个字大小的内存地址的值移动到一个段寄存器
A0	MOV AL, moffs8	4	把一个字节大小的内存地址的值移动到AL寄存器
A1	MOV AX, moffs16	4	把一个字大小的内存地址的值移动到AX寄存器
A1	MOV EAX, moffs32	4	把一个双字大小的内存地址的值移动到EAX寄存器
A2	MOV moffs8, AL	2	把AL寄存器的值移动到一个字节大小的内存地址
A3	MOV moffs16, AX	2	把AX寄存器的值移动到一个字大小的内存地址
A3	MOV moffs32, EAX	2	把EAX寄存器的值移动到一个双字大小的内存地址
B0+rb	MOV reg8, imm8	2	把一个字节大小的立即数移动到一个字节大小的寄存器
B8+rw	MOV reg16, imm16	2	把一个字大小的立即数移动到一个字大小的寄存器
B8+rd	MOV reg32, imm32	2	把一个双字大小的立即数移动到一个双字大小的寄存器
C6 ib	MOV r/m8,imm8	2/2	将立即字节移动到r/m字节
C7 iw	MOV r/m16,imm16	2/2	将立即字移动到r/m字
C7 id	MOV r/m32,imm32	2/2	将立即双字移动到r/m双字

MOV -- Move to/from Special Registers

Opcode	Instruction	Clocks	Description
0F 20/r	MOV r32,CR0/CR2/CR3	6	将控制寄存器移动到通用寄存器
0F 22/r	MOV CR0/CR2/CR3,r32	10/4/5	将通用寄存器移动到控制寄存器
0F 21/r	MOV r32,DR0--3	22	将调试寄存器移动到通用寄存器
0F 21/r	MOV r32,DR6/DR7	14	将调试寄存器移动到通用寄存器
0F 23/r	MOV DR0--3,r32	22	将通用寄存器移动到调试寄存器
0F 23/r	MOV DR6/DR7,r32	16	将通用寄存器移动到调试寄存器
0F 24/r	MOV r32,TR6/TR7	12	将测试寄存器移动到通用寄存器
0F 26/r	MOV TR6/TR7,r32	12	将通用寄存器移动到测试寄存器

1. 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?
2. 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"?)
3. 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.
4. 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?

1. 4468行:
2. 使用命令: `find ./ \( -name '*.c' -o -name '*.h' \) -print0 | xargs -0 wc -l`

```
user@user-VirtualBox:~/ics2017/nemu$ find ./ \( -name '*.c' -o -name '*.h' \) -print0 | xargs -0 wc -l
    28 ./src/memory/memory.c
    31 ./src/misc/logo.c
   106 ./src/monitor/diff-test/gdb-host.c
   157 ./src/monitor/diff-test/diff-test.c
    48 ./src/monitor/diff-test/protocol.h
   314 ./src/monitor/diff-test/protocol.c
   143 ./src/monitor/monitor.c
   239 ./src/monitor/debug/ui.c
   429 ./src/monitor/debug/expr-bk.c
    99 ./src/monitor/debug/watchpoint.c
   449 ./src/monitor/debug/expr.c
    44 ./src/monitor/cpu-exec.c
   311 ./src/cpu/decode/decode.c
   113 ./src/cpu/decode/modrm.c
    43 ./src/cpu/reg.c
    46 ./src/cpu/exec/special.c
   223 ./src/cpu/exec/arith.c
    32 ./src/cpu/exec/cc.c
   255 ./src/cpu/exec/exec.c
     9 ./src/cpu/exec/prefix.c
    65 ./src/cpu/exec/system.c
    60 ./src/cpu/exec/logic.c
    77 ./src/cpu/exec/data-mov.c
    43 ./src/cpu/exec/control.c
     8 ./src/cpu/exec/all-instr.h
    13 ./src/cpu/intr.c
    12 ./src/main.c
    70 ./src/device/keyboard.c
    70 ./src/device/io/mmio.c
    56 ./src/device/io/port-io.c
    29 ./src/device/serial.c
    98 ./src/device/device.c
    38 ./src/device/vga.c
    28 ./src/device/timer.c
    18 ./include/memory/memory.h
    82 ./include/memory/mmu.h
    13 ./include/macro.h
     8 ./include/nemu.h
     7 ./include/monitor/monitor.h
    21 ./include/monitor/watchpoint.h
     8 ./include/monitor/expr.h
    53 ./include/cpu/exec.h
    67 ./include/cpu/reg.h
   189 ./include/cpu/rtl.h
   115 ./include/cpu/decode.h
    29 ./include/common.h
    45 ./include/debug.h
    14 ./include/device/mmio.h
    13 ./include/device/port-io.h
  4468 总用量
user@user-VirtualBox:~/ics2017/nemu$
```

使用两次上述命令，在两次之间切换分支，两个结果相减即可：

```
user@user-VirtualBox:~/ics2017/nemu$ find ./ -name "*.c" -o -name "*.h" | xargs grep -v '^$' | wc -l
3740
user@user-VirtualBox:~/ics2017/nemu$ git checkout master
M       Makefile
M       include/monitor/watchpoint.h
M       src/monitor/debug/expr.c
M       src/monitor/debug/ui.c
M       src/monitor/debug/watchpoint.c
切换到分支 'master'
user@user-VirtualBox:~/ics2017/nemu$ find ./ -name "*.c" -o -name "*.h" | xargs grep -v '^$' | wc -l
3321
user@user-VirtualBox:~/ics2017/nemu$
```

master是编写前的分支，相减得419行。

3. 在makefile中添加：

```
count:
	@find ./ -type f \( -name "*.c" -o -name "*.h" \) -print0 | xargs -0 wc -l | tail -1
```

```
user@user-VirtualBox:~/ics2017/nemu$ make count
  4468 总用量
user@user-VirtualBox:~/ics2017/nemu$
```

4. 使用命令：`find ./ -name "*.c" -o -name "*.h" | xargs grep -v '^$' | wc -l`，

```
user@user-VirtualBox:~/ics2017/nemu$ find ./ -name "*.c" -o -name "*.h" | xargs grep -v '^$' | wc -l
3740
```

其中，`./`是需要统计的目录路径。`-name`选项用于指定文件名的匹配模式，使用通配符`*`匹配所有以.c和.h结尾的文件。`xargs`命令用于将多个文件名作为参数传递给`grep`命令，`-v`选项用于过滤掉空行，`wc -l`命令用于计算行数；

> ❓ 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`生成所有警告信息，`-Werror`在发生警告时停止编译操作，即要求 gcc 将所有警告当成错误进行处理。