南开大学

Nankai University

南　　开　　大　　学

计 算 机 学 院

计算机系统设计作业报告

PA **实验一报告**

朱浩泽 1911530

年级：2019 级

专业：计算机科学与技术

指导教师：卢冶

2022 年 3 月 17 日

目录

一、 概述

(一) 实验目的

熟悉基础设施的各种工具和手段；熟悉寄存器间的存储关系；学习实现简易调试器补充指令；学习表达式求值，监视点的实现方法

(二) 实验内容

- **阶段一**
  - 实现正确的寄存器结构体
  - 实现解析命令
  - 实现单步执行
  - 实现打印寄存器
  - 实现内存扫描
- **阶段二**
  - 实现词法分析
  - 实现递归求值
  - 实现调试中的表达式求值
- **阶段三**
  - 实现断点
  - 实现监视点

二、 阶段一

(一) 实现正确的寄存器结构体

- 为了实现 32 位、16 位、8 位寄存器各 8 个和一个程序计数器 eip，我们利用匿名 Union 各个变量互斥并共享同一内存首地址这一特点，来实现这一结构体
- 修改 nemu/include/cpu/reg.h 中的代码如下

```
typedef struct {
  union{
    /* data */
    union {
      uint32_t _32;
      uint16_t _16;
      uint8_t _8[2];
    } gpr[8];
    struct
    {
      rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
    };
  };
  vaddr_t eip;
} CPU_state;
```

- 在上面的代码中，grp 中的的 _32 代表的是 eax 中的 32 位，_16 代表的是 ax 中的 16 位，_8[2] 分别代表的是 ah 中的 8 到 16 位和 al 中的 0 到 8 位

(二) 实现解析命令

- 在 nemu/src/monitor/debug/ui.c 中实现命令表，其代码如下

```
static struct {
  char *name;
  char *description;
  int (*handler) (char *);
} cmd_table [] = {
  { "help", "Display informations about all supported commands", cmd_help },
  { "c", "Continue the execution of the program", cmd_c },
  { "q", "Exit NEMU", cmd_q },

  /* TODO: Add more commands */

  { "si", "args:[N]; exectue [N] instructions step by step", cmd_si}, //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
  { "info", "args:r/w;print information about register or watch point ", cmd_info }, //打印寄存器状态
  { "x", "x [N] [EXPR];sacn the memory", cmd_x }, //内存扫描
  { "p", "expr", cmd_p}, //表达式
  { "w", "set the watchpoint", cmd_w}, //添加监视点
  { "d", "delete the watchpoint", cmd_d} //删除监视点
};
```

- 其执行效果如下图所示

```
(nemu) help
help - Display informations about all supported commands
c - Continue the execution of the program
q - Exit NEMU
si - args:[N]; exectue [N] instructions step by step
info - args:r/w;print information about register or watch point
x - x [N] [EXPR];sacn the memory
p - expr
w - set the watchpoint
d - delete the watchpoint
```

(三) 实现单步执行

- 在 nemu/src/monitor/debug/ui.c 添加 cmd_si 函数，其代码如下

```
static int cmd_si(char *args) {
  uint64_t N = 0;
  if(args == NULL) {
    N = 1;
  }
  else {
    int temp = sscanf(args, "%lu", &N);
    if(temp <= 0) {
      printf("args error in cmd_si\n");
      return 0;
    }
  }
```

```
  cpu_exec(N);
  return 0;
}
```

- 执行效果如下图所示，输入 si 和要执行的步骤 N，进行打印

```
lighthouse@VM-24-4-ubuntu:~/main/PA/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: 10:56:22, Mar 16 2022
For help, type "help"
(nemu) si 5
  100000:   b8 34 12 00 00                      movl $0x1234,%eax
  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)
  100012:   bb 02 00 00 00                      movl $0x2,%ebx
(nemu)
```

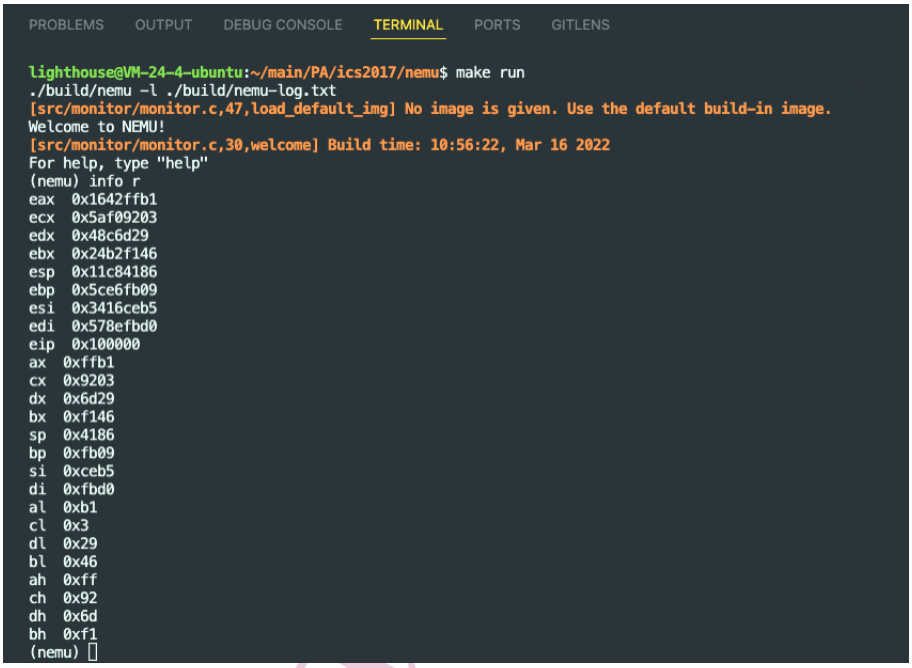
（四） 实现打印寄存器

- 在 nemu/src/monitor/debug/ui.c 添加 cmd_info 函数，其代码如下，输入参数 r 对寄存器直接进行打印，输入参数 w 对监视点进行打印

```
static int cmd_info(char *args) {
  char s;
  if(args == NULL) {
    printf("args error in cmd_info (miss args)\n");
    return 0;
  }
  int temp = sscanf(args, "%c", &s);
  if(temp <= 0) {
    //解析失败
    printf("args error in cmd_info\n");
    return 0;
  }
  if(s == 'w') {
    //打印监视点信息
    print_wp();;
    return 0;
  }
  if(s == 'r') {
    //打印寄存器
    //32bit
    for(int i = 0; i < 8; i++) {
      printf("%s  0x%x\n", regsl[i], reg_l(i));
    }
    printf("eip  0x%x\n", cpu.eip);
    //16bit
    for(int i = 0; i < 8; i++) {
      printf("%s  0x%x\n", regsw[i], reg_w(i));
    }
    //8bit
    for(int i = 0; i < 8; i++)
    {
      printf("%s  0x%x\n", regsb[i], reg_b(i));
    }
    return 0;
```

```
  }
  //如果产生错误
  printf("args error in cmd_info\n");
  return 0;
}
```

- 执行效果如下图所示

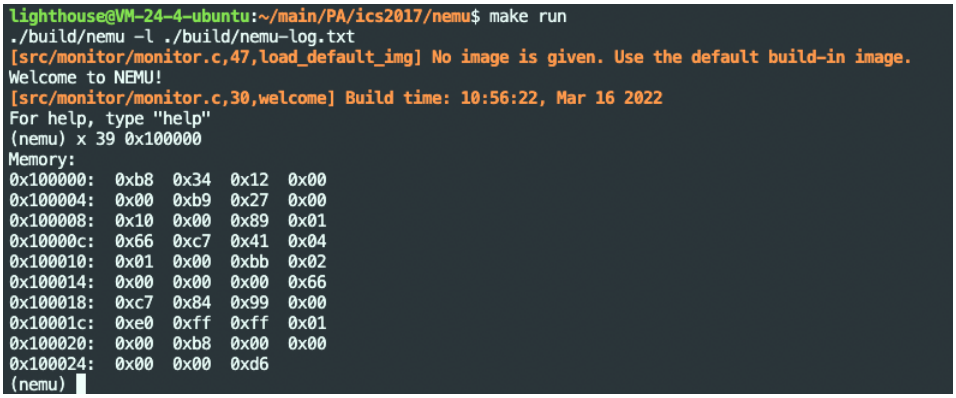


（五） 实现内存扫描

- 在 nemu/src/monitor/debug/ui.c 添加 cmd_x 函数，其代码如下

```
static int cmd_x(char *args) {
  int nLen = 0;
  vaddr_t addr;
  int temp = sscanf(args, "%d 0x%x", &nLen, &addr);
  if(temp <= 0) {
    //解析失败
    printf("args error in cmd_si\n");
    return 0;
  }
  printf("Memory:");
  for(int i = 0; i < nLen; i++) {
    if(i % 4 == 0) {
      printf("\n0x%x:  0x%02x", addr + i, vaddr_read(addr + i, 1));
    }
    else {
      printf("  0x%02x", vaddr_read(addr + i, 1));
    }
  }
  printf("\n");
  return 0;
}
```

- 该函数主要是调用了 nemu/src/memory/memory.c 中的 vaddr_read 函数进行内存扫描
- 执行效果如下图所示



三、 阶段二

在本次实验中，要求实现包括 + - * / == != >= <= > < && || ！指针和正负运算等多种运算，以及十进制数、八进制数、十六进制数等运算。

在有了编译原理课程的基础后，我们可以知道，表达式求值的过程是先将整个句子切分为各个 token，然后对各个 token 按照赋予的优先级进行分割后求返回值，最终通过这种递归运算求出整个表达式的值。

在这个求值过程中，我们需要注意寄存器的值是可以当作操作数进行运算的，且对于括号的操作跟编译原理中直接利用 lex 工具，本次实验中需要手写函数进行动态判定，这本质上是一个脱括号的过程，需要判断脱括号后其匹配是否合法。

本节实验主要代码在 nemu/src/monitor/debug/exp.c 文件中实现

(一) 词法分析

- 首先我们定义 tokens 类型和正则表达式来识别 tokens，其代码如下

```
enum {
  TK_NOTYPE = 256,
  TK_NUMBER,
  TK_HEX,
  TK_REG,
  TK_EQ,
  TK_NEQ,
  TK_AND,
  TK_OR,
  TK_NEGATIVE,
  TK_DEREF,
};

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
  {"0x[0-9A-Fa-f][0-9A-Fa-f]*", TK_HEX},
  {"0|[1-9][0-9]*", TK_NUMBER}, //数字
  {"\\$(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},
  {"==", TK_EQ},
  {"!=", TK_NEQ},
  {"&&", TK_AND},
  {"\\|\\|", TK_OR},
  {"!", '!'},
  {"\\+", '+'},
  {"-", '-'},
  {"\\*", '*'},
  {"\\/", '/'},
  {"\\(", '('},
  {"\\)", ')'},
};
```

- 修改代码框架中的 make_token 函数，对各种类型的输入进行处理，其中八进制数和十六进制数将会被提前处理，而寄存器则将会跳过开头的美元符号，其代码如下

```
static bool make_token(char *e) {
int position = 0;
int i;
regmatch_t pmatch;

nr_token = 0;

while (e[position] != '\0') {
  /* Try all rules one by one. */
  for (i = 0; i < NR_REGEX; i ++) {
    if (regexec(&re[i], e + position, 1, &pmatch, 0) == 0 && pmatch.rm_so == 0) {
      char *substr_start = e + position;
      int substr_len = pmatch.rm_eo;

      Log("match rules[%d] = \"%s\" at position %d with len %d: %.*s",
          i, rules[i].regex, position, substr_len, substr_len, substr_start);
      position += substr_len;

      /* TODO: Now a new token is recognized with rules[i]. Add codes
        * to record the token in the array `tokens'. For certain types
        * of tokens, some extra actions should be performed.
        */
      if(substr_len > 32) {
        assert(0);
      }
      if(rules[i].token_type == TK_NOTYPE) {
        break;
      }
      else {
        tokens[nr_token].type = rules[i].token_type;
        switch (rules[i].token_type) {
        case TK_NUMBER:
```

```
        strncpy(tokens[nr_token].str, substr_start, substr_len);
        *(tokens[nr_token].str + substr_len) = '\0';
        break;
      case TK_HEX:
        strncpy(tokens[nr_token].str, substr_start + 2, substr_len - 2); //跳过开头的0x
        *(tokens[nr_token].str + substr_len - 2) = '\0';
        break;
      case TK_REG:
        strncpy(tokens[nr_token].str, substr_start + 1, substr_len - 1); //跳过开头的$
        *(tokens[nr_token].str + substr_len - 1) = '\0';
      }
      printf("Success record : nr_token = %d, dtype = %d, str = %s\n", nr_token, tokens[nr_token].type, tokens[nr_token].str);
      nr_token += 1;
      break;
    }
   }
   if (i == NR_REGEX) {
    return false;
   }
  }

  return true;
}
```

(二) 递归求值

- 首先我们要进行的工作是检查括号匹配，其主要想法是利用一个计数器来计算匹配的数目，如果最终计数器的值不为 0 则说明括号不匹配，则表达式出错。其代码如下

```
//判断括号的匹配
bool check_parentheses(int p, int q) {
  if(p >= q) {
    //右括号少于左括号
    printf("error:p>=q in check_parntheses\n");
    return false;
  }
  if(tokens[p].type != '(' || tokens[q].type != ')'){
    //括号不匹配
    return false;
  }
  int cnt = 0; //记录当前未匹配的左括号的数目
  for(int curr = p + 1; curr < q; curr++) {
    if(tokens[curr].type == '(') {
      cnt++;
    }
    if(tokens[curr].type == ')') {
      if(cnt != 0) {
        cnt--;
      }
      else {
        //左右括号不匹配
        return false;
```

```
    }
   }
  }
  if(cnt == 0) {
   return true;
  }
  else {
   return false;
  }
}
```

- 利用一个函数来计算表达式优先级最低的运算符的位置，如果有同为最低优先级的预算符，返回最后被结合的运算符的索引位置，其代码如下

```
int findDominantOp(int p, int q) {
  int level=0;
  int pos[5]={-1, -1, -1, -1, -1};
  for(int i = p; i < q; i++){
   if(level == 0) {
    if(tokens[i].type == TK_AND || tokens[i].type == TK_OR) {
     pos[0] = i;
    }
    if(tokens[i].type == TK_EQ || tokens[i].type == TK_NEQ) {
     pos[1] = i;
    }
    if(tokens[i].type == '+' || tokens[i].type == '-') {
     pos[2] = i;
    }
    if(tokens[i].type == '*' || tokens[i].type == '/') {
     pos[3] = i;
    }
    if(tokens[i].type == TK_NEGATIVE || tokens[i].type == TK_DEREF || tokens[i].
        type == '!') {
     pos[4] = i;
    }
   }
   if(tokens[i].type=='(') {
    level++;
   }
   if(tokens[i].type==')') {
    level--;
   }
  }
  for(int i = 0; i < 5; i++) {
   if(pos[i] != -1) {
    return pos[i];
   }
  }
  printf("error in findDominantOp\n");
  printf("[p=%d,q=%d]\n",p,q);
  assert(0);
}
```

- 根据实验指导书编写 eval 函数，用于值的计算，其中的参数 p 和 q 分别代表这个子表达式的开始位置和结束为止，其代码如下

```
uint32_t eval(int p, int q) {
  if(p > q) {
    printf("error:p>q in eval, p = %d, q = %d\n", p, q);
    assert(0);
  }
  if(p == q) {
    int num;
    switch (tokens[p].type){
      case TK_NUMBER:
        sscanf(tokens[p].str, "%d", &num);
        return num;
      case TK_HEX:
        sscanf(tokens[p].str, "%x", &num);
        return num;
      case TK_REG:
        for(int i = 0; i < 8; i++) {
          if(strcmp(tokens[p].str, regsl[i]) == 0) {
            return reg_l(i);
          }
          if(strcmp(tokens[p].str, regsw[i]) == 0) {
            return reg_w(i);
          }
          if(strcmp(tokens[p].str, regsb[i]) == 0) {
            return reg_b(i);
          }
        }
        if(strcmp(tokens[p].str, "eip") == 0) {
          return cpu.eip;
        }
        else {
          printf("error in TK_REG in eval()\n");
          assert(0);
        }
    }
  }
  if(check_parentheses(p, q) == true) {
    return eval(p + 1, q - 1);
  }
  else {
    int op = findDominantOp(p, q);
    vaddr_t addr;
    int result;
    switch (tokens[op].type) {
      case TK_NEGATIVE: //负号
        return -eval(p + 1, q);
      case TK_DEREF: //指针求值
        addr = eval(p + 1, q);
        result = vaddr_read(addr, 4);
        printf("adddr=%u(0x%x)---->value=%d(0x%08x)\n", addr, addr, result, result)
            ;
        return result;
      case '!':
        result = eval(p + 1, q);
        if(result != 0) {
          return 0;
```

```
      }
      else {
        return 1;
      }
    }
    uint32_t val1 = eval(p, op - 1);
    uint32_t val2 = eval(op + 1, q);
    switch(tokens[op].type) {
      case '+':
        return val1 + val2;
      case '-':
        return val1 - val2;
      case '/':
        return val1 / val2;
      case '*':
        return val1 * val2;
      case TK_EQ:
        return val1 == val2;
      case TK_NEQ:
        return val1 != val2;
      case TK_AND:
        return val1 && val2;
      case TK_OR:
        return val1 || val2;
      default:
        assert(0);
    }
  }
  return 1;
}
```

(三) 实现调试中的表达式求值

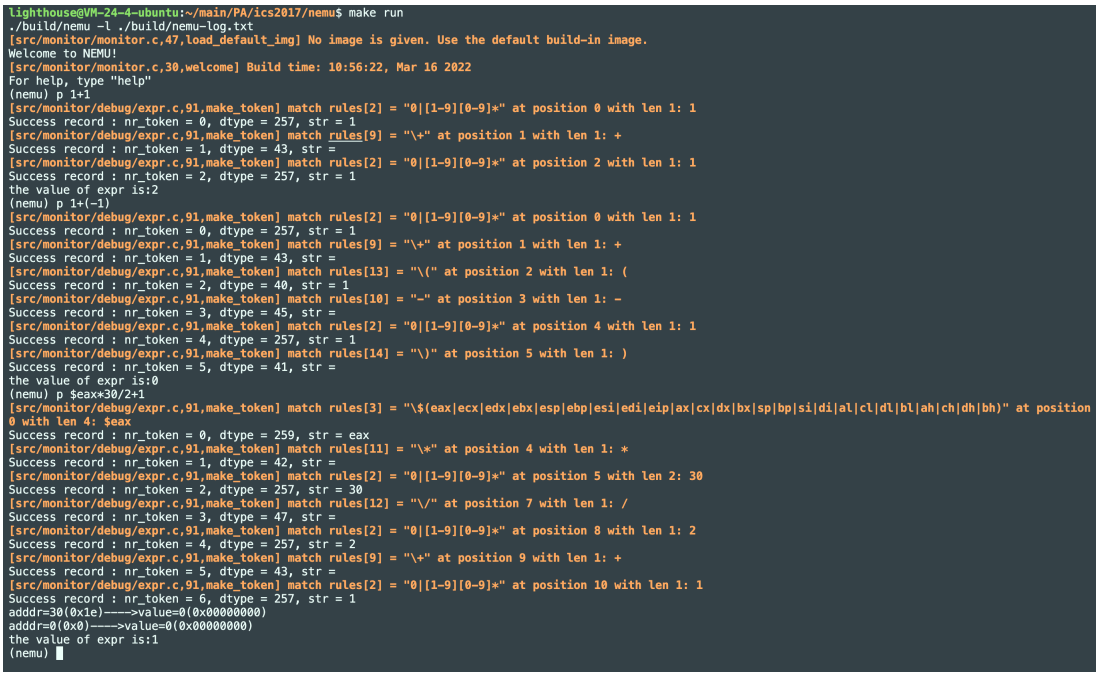
- 修改框架代码中的 expr 函数进行表达式求值的工作,其中主要分为两步:先调用 make_token 进行词法分析，再调用 eval 函数进行求值操作。为了实现这一功能，我们要对 token 进行有选择保存信息：对于寄存器和数字，我们保存其类型，并利用字符串保存寄存器的名字和数值；对于一般运算符保存 token 类型；对于加减法需要区分是单目运算符还是双目运算符，其代码如下

```
uint32_t expr(char *e, bool *success) {
  if (!make_token(e)) {
    *success = false;
    return 0;
  }
  /* TODO: Insert codes to evaluate the expression. */
  // TODO();

  // return 0;
  if(tokens[0].type == '-') {
    tokens[0].type = TK_NEGATIVE;
  }
  if(tokens[0].type == '*') {
    tokens[0].type = TK_DEREF;
  }
```

```
  for(int i = 1; i < nr_token; i++) {
    if(tokens[i].type == '-') {
      if(tokens[i - 1].type != TK_NUMBER && tokens[i - 1].type != ')') {
        tokens[i].type = TK_NEGATIVE;
      }
    }
    if(tokens[i].type == '*') {
      if(tokens[i - 1].type != TK_NUMBER && tokens[i - 1].type != ')') {
        tokens[i].type = TK_DEREF;
      }
    }
  }
  *success = true;
  return eval(0, nr_token - 1);
}
```

- 实验效果如下图所示



四、 阶段三

监视点和断点

这一部分的主要工作是通过实现两个链表（free 和 head）进行维护的，其代表空闲的和占用的监视点。主要工作和代码在 nemu/include/ monitor/watchpoint.h 和 nemu/src/monitor/debug/watchpoint.c 文件中实现。

首先是关于实现方式的，如果删除某个监视点，其余监视点不会递补该监视点的编号（这操作实际上减少了部分工作量），新增加的监视点则会在最大的监视点编号上继续增加。监视点结构及其初始化的代码如下：

```
typedef struct watchpoint {
  int NO;
```

```
  struct watchpoint *next;

  /* TODO: Add more members if necessary */
  int old; //旧的值
  char e[32]; //表达式
  int hitNum; //记录触发次数

} WP;

//结构初始化
void init_wp_pool() {
  int i;
  for (i = 0; i < NR_WP; i ++) {
    wp_pool[i].NO = i;
    wp_pool[i].next = &wp_pool[i + 1];
    wp_pool[i].old = 0;
    wp_pool[i].hitNum = 0;
  }
  wp_pool[NR_WP - 1].next = NULL;

  head = NULL;
  free_ = wp_pool;
  used_next = 0;
}
```

接下来，我们创建 new_up 函数和 free_up 函数来创建和删除监视点。主要思路是在添加监视点时，首先查找是否有足够的位置，如果有的话从链表中返回一个空闲的监视点结构，并更新 head 链表，将新添加的监视点添加到其尾部；在删除监视点的时候，先遍历 head 链表找到要被删除的监视点进行释放，然后更新 free 链表对空闲的监视点进行更新。其主要代码如下：

```
bool new_wp(char *args) {
  //从free链表中返回一个空闲监视点结构
  if(free_ == NULL) {
    assert(0);
  }
  WP* result = free_;
  free_ = free_ -> next;

  result -> NO = used_next;
  used_next++;
  result -> next = NULL;
  strcpy(result -> e, args);
  result -> hitNum = 0;
  bool is_success;
  result -> old = expr(result -> e, &is_success); //计算旧的值
  if(is_success == false) {
    printf("error in new_wp; expression fault!\n");
    return false;
  }

  wptemp = head;
  if(wptemp == NULL) {
    head = result;
  }
  else {
```

```
    while (wptemp -> next != NULL)
    {
      wptemp = wptemp -> next;
    }
    wptemp -> next = result;
  }
  printf("Success: set watchpoint %d, oldvalue = %d\n", result -> NO, result -> old);
  return true;
}

//删除监视点
bool free_wp(int num) {
  WP *chosen = NULL;
  if(head == NULL) {
    printf("no watch point now\n");
    return false;
  }
  if(head -> NO == num) {
    chosen = head;
    head = head -> next;
  }
  else {
    wptemp = head;
    while (wptemp != NULL && wptemp -> next != NULL)
    {
      /* code */
      if(wptemp -> next -> NO == num) {
        chosen = wptemp -> next;
        wptemp -> next = wptemp -> next -> next;
        break;
      }
      wptemp = wptemp -> next;
    }
  }
  if(chosen != NULL) {
    chosen -> next = free_;
    free_ = chosen;
    return true;
  }
  return false;
}
```

同时,我们还对监视点进行实时监测,其主要调用了 nemu/include/cpu/exe.c 中的 cpu_exec() 函数。通过在 cpu 每执行一次命令是都检查一下所有的监视点的表达式的值是否发生了变化，如果发生了则打印变化信息来实现。其主要代码如下

```
void print_wp() {
  if(head == NULL) {
    printf("no watchpoint now\n");
    return;
  }
  printf("watchpoint:\n");
  printf("NO.  expr    hitTimes\n");
  wptemp = head;
  while (wptemp != NULL)
  {
```

```
    printf("%d  %s    %d\n", wptemp -> NO, wptemp -> e, wptemp -> hitNum);
    wptemp = wptemp ->next;
  }
}

bool watch_wp() {
  bool is_success;
  int result;
  if(head == NULL) {
    return true;
  }
  wptemp = head;
  while (wptemp != NULL)
  {
    /* code */
    result = expr(wptemp -> e, &is_success);
    if(result != wptemp -> old)
    {
      wptemp -> hitNum += 1;
      printf("Hardware watchpoint %d:%s\n", wptemp -> NO, wptemp -> e);
      printf("Old value:%d\nNew valus:%d\n\n", wptemp -> old, result);
      wptemp -> old = result;
      return false;
    }
    wptemp = wptemp -> next;
  }
  return true;
}
```

实验效果如下图所示

```
lighthouse@VM-24-4-ubuntu:~/main/PA/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: 10:56:22, Mar 16 2022
For help, type "help"
(nemu) w $ecx
[src/monitor/debug/expr.c,91,make_token] match rules[3] = "\$(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)" at position 0 with len 4: $ecx
Success record : nr_token = 0, dtype = 259, str = ecx
Success: set watchpoint 0, oldvalue = 1944013018
(nemu) si 4
  100000:   b8 34 12 00 00                  movl $0x1234,%eax
[src/monitor/debug/expr.c,91,make_token] match rules[3] = "\$(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)" at position 0 with len 4: $ecx
Success record : nr_token = 0, dtype = 259, str = ecx
  100005:   b9 27 00 10 00                  movl $0x100027,%ecx
[src/monitor/debug/expr.c,91,make_token] match rules[3] = "\$(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)" at position 0 with len 4: $ecx
Success record : nr_token = 0, dtype = 259, str = ecx
Hardware watchpoint 0:$ecx
Old value:1944013018
New valus:1048615

(nemu) info w
watchpoint:
NO.  expr    hitTimes
0  $ecx    1
(nemu) delete wacthpoint 0
Unknown command 'delete'
(nemu) d 0
Success delete watchpoint 0
(nemu) info w
no watchpoint now
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100026

(nemu)
```

五、 必答题

(一) 查阅 i386 手册回答问题

1. EFLAGS 寄存器中的 CF 位是什么意思?

CF 是 Carry Flag 的缩写，位于 EFLAGS 的第 0 位，是标志进位的意思。用来允许一条指令的结果影响后面的指令，即一些指令用于设置高位进位或借位，否则清 0。该结果是位于 i386-2.3Register(P33) 和 i386-附录 C(P419)

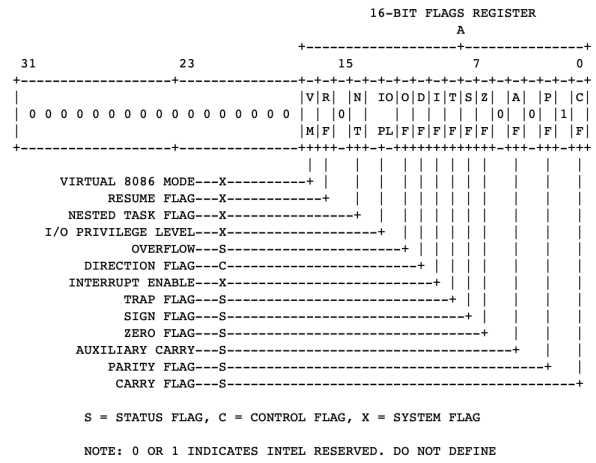
2.3.4 Flags Register

The flags register is a 32-bit register named EFLAGS. Figure 2-8 defines the bits within this register. The flags control certain operations and indicate the status of the 80386.

The low-order 16 bits of EFLAGS is named FLAGS and can be treated as a unit. This feature is useful when executing 8086 and 80286 code, because this part of EFLAGS is identical to the FLAGS register of the 8086 and the 80286.

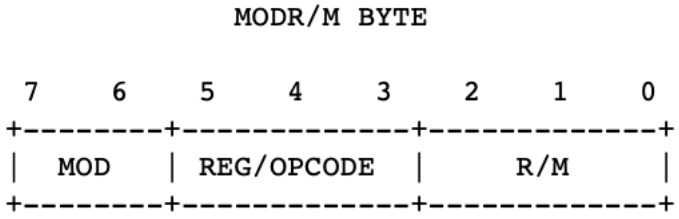
The flags may be considered in three groups: the status flags, the control flags, and the systems flags. Discussion of the systems flags is delayed until Part II.

Figure 2-8. EFLAGS Register



2. ModR/M 字节是什么?

ModR/M 是 opcode(操作码) 后面的一个字节，是变长指令的一部分，用于指定带操作的寄存器或是操作数在内存中。参考 i386-17.2(P38-40)



3. mov 指令的具体格式是怎么样的?

参考 i386-3.1(P45)

MOV (Move) transfers a byte, word, or doubleword from the source operand to the destination operand. The MOV instruction is useful for transferring data along any of these paths

There are also variants of MOV that operate on segment registers. These are covered in a later section of this chapter.:

- To a register from memory
- To memory from a register
- Between general registers
- Immediate data to a register
- Immediate data to a memory

参考 i386-17.3(P345)

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
C6	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.

(二) 题目二

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**?

- 用于计算.c .h 文件有多少行

```
find ./ -name "*.[ch]" |xargs cat|wc -l
```

- 用于计算.c .h 文件除去空格有多少行

```
find . -name "*[.cpp| .h]" | xargs grep "^."| wc -l
```

- 如果需要统计 PA1 做的代码量，只需将 git branch 切换到 PA0 做一次统计，将其结果相减便可以得到 PA1 的代码量

- 试验结果如下

```
lighthouse@VM-24-4-ubuntu:~/main/PA/ics2017/nemu$ find ./ -name "*.[ch]" |xargs cat|wc -l
3956
lighthouse@VM-24-4-ubuntu:~/main/PA/ics2017/nemu$ find . -name "*[.cpp| .h]" | xargs grep "^."| wc -l
grep: .: 是一个目录
grep: ./src: 是一个目录
grep: ./src/cpu/exec: 是一个目录
grep: ./src/misc: 是一个目录
grep: ./build/obj/cpu/exec: 是一个目录
grep: ./build/obj/misc: 是一个目录
3308
lighthouse@VM-24-4-ubuntu:~/main/PA/ics2017/nemu$ git checkout pa0
切换到分支 'pa0'
您的分支与上游分支 'origin/pa0' 一致。
lighthouse@VM-24-4-ubuntu:~/main/PA/ics2017/nemu$ find . -name "*[.cpp| .h]" | xargs grep "^."| wc -l
grep: .: 是一个目录
grep: ./src: 是一个目录
grep: ./src/cpu/exec: 是一个目录
grep: ./src/misc: 是一个目录
grep: ./build/obj/cpu/exec: 是一个目录
grep: ./build/obj/misc: 是一个目录
2850
lighthouse@VM-24-4-ubuntu:~/main/PA/ics2017/nemu$
```

可以看出，PA1 中.c 和.h 文件中的代码总共有 3956 行，除去空行总共有 3308 行，PA1 总工作量为 3308-2850 = 458 行代码。

（三） 题目三

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，打开 gcc 的所有警告。

- -Werror，它要求 gcc 将所有的警告当成错误进行处理

我觉得使用这两者是为了在编译阶段尽早发现错误，提高代码的安全性，便于 bug 的发现与修复，提高程序的鲁棒性。

六、 感想与遇到的问题

最初我是在 Mac M1 芯片体系结构下利用 VMWare 安装 arm 架构的 Ubuntu 20.04 系统下进行试验的。可是很遗憾，只要一 make 就会报错，如下图

```
+ CC src/cpu/decode/decode.c
+ CC src/cpu/exec/cc.c
+ CC src/cpu/exec/arith.c
In file included from ./include/cpu/decode.h:6,
                 from ./include/cpu/exec.h:9,
                 from src/cpu/exec/arith.c:1:
src/cpu/exec/arith.c: In function ‘exec_mul’:
./include/cpu/rtl.h:47:3: error: impossible constraint in ‘asm’
   47 |   asm volatile("mul %3" : "=d"(*dest_hi), "=a"(*dest_lo) : "a"(*src1), "r"(*src2));
      |   ^~~
make: *** [Makefile:26: build/obj/cpu/exec/arith.o] 错误 1
```

这是因为 arm linux 和 x86linux 在交叉编译上所出现的问题，通过查阅资料得知，通过在此时只要在./config 命令中添加：–without-v4l ，然后 make clean, 重新 make 即可修复该问题。但考虑

到体系结构不同可能在后续实验中带来一系列的不必要的麻烦等问题，我选择了放弃 arm linux，尝试在电脑上通过 UTM 虚拟 x86 版 Linux，但由于差异巨大，导致性能极差，故再次放弃。最终，我选择租用腾讯云服务器，在腾讯云上安装了 Ubuntu Server 20.04 LTS 64bit 进行，配合 Vs Code 的远程 ssh 控制进行实验（我实在用不惯 vim）。但在 PA1 实验开始进行 make 操作时，仍遇到了编译错误，如下图

这是一个很明显的编译警告问题由于 gcc 的参数设置成了-Wall 导致编译错误，于是我毫不犹豫的更改了 MakeFile，顺利的进行了实验。但我很不理解作者为什么要留这样一个坑，直到前几天老师在微信群里说了这个问题，大概是编译器版本没有选择指导书建议的吧，不然高版本编译优化级别可能会带来带来暗坑，顿时又有些害怕。希望今后的三个 PA 实验不要因为环境导致错误导致整个 remake。