



南開大學  
Nankai University

南开大学  
计算机学院  
计算机系统实验报告

---

PA1

---

年级：2022 级  
专业：计算机科学与技术  
姓名：姚知言  
学号：2211290  
指导教师：卢冶

2025 年 3 月 19 日

# 目录

<b>一、 实验目的</b>	<b>1</b>
<b>二、 实验重要内容</b>	<b>1</b>
<b>三、 阶段 1 实验方法与结果</b>	<b>2</b>
(一) 编码内容及测试结果 . . . . .	2
1. 实现正确的寄存器结构体 . . . . .	2
2. 实现单步执行 . . . . .	3
3. 实现打印寄存器 . . . . .	4
4. 实现扫描内存 . . . . .	5
(二) 问答题 . . . . .	7
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? . . . . .	7
2. 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? . . . . .	8
<b>四、 阶段 2 实验方法与结果</b>	<b>8</b>
(一) 编码内容 . . . . .	8
1. 实现算术表达式的词法分析 . . . . .	8
2. 算符优先级设置 . . . . .	11
3. expr 求值入口函数的设置 . . . . .	11
4. 括号匹配验证函数 . . . . .	11
5. eval 递归求值函数 . . . . .	12
6. ui.c 中 cmd_p 的创建及 cmd_x 的完善 . . . . .	15
(二) 测试结果 . . . . .	16
<b>五、 阶段 3 实验方法与结果</b>	<b>19</b>
(一) 编码内容 . . . . .	19
1. 实现监视点池的管理 . . . . .	19
2. 实现监视点 . . . . .	20
(二) 测试结果 . . . . .	23
(三) 问答题 . . . . .	25
1. 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? . . . . .	25
2. 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? . . . . .	26
3. 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。 . . . . .	26

4. 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(NJUDebugger), 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? . . . . .	27
5. 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? . . . . .	27
<b>六、必答题</b>	<b>28</b>
(一) Question 1 . . . . .	28
1. EFLAGS 寄存器中的 CF 位是什么意思? . . . . .	28
2. ModR/M 字节是什么? . . . . .	29
3. mov 指令的具体格式是怎么样的? . . . . .	29
(二) Question 2 . . . . .	30
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? 你是使用什么命令得到这个结果的? . . . . .	30
2. 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”?) . . . . .	31
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. . . . .	31
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? . . . . .	31
(三) Question 3 . . . . .	32
1. 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? . . . . .	32
<b>七、所遇 BUG 及解决思路</b>	<b>32</b>
(一) 虚拟机无法复制文本 . . . . .	32
(二) 修改 git 远程仓库时遇到的问题 . . . . .	32

## 一、实验目的

1. 熟悉 i386 架构。
2. 熟悉 GDB 使用，并从底层模拟实现。
3. 复习编译原理词法分析和数据结构等知识。

## 二、实验重要内容

PA1 的实验共分为 3 个阶段，包括以下要点：

- 第 1 阶段：使用 union 结构实现正确的寄存器结构体。

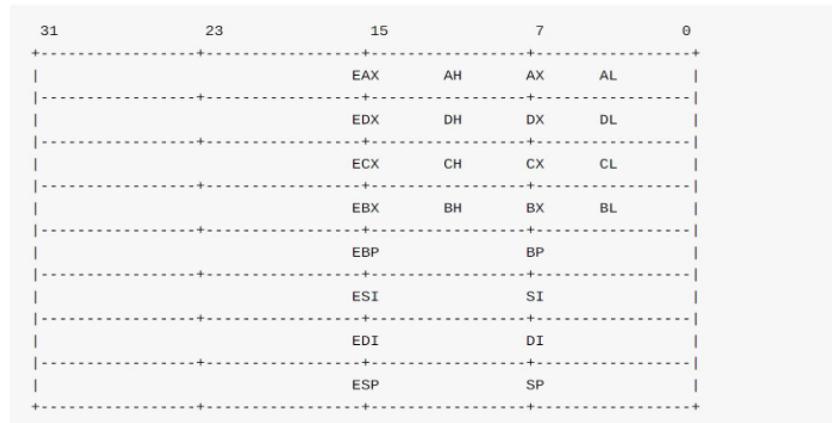


图 1: 寄存器结构体结构

- 第 2 阶段：完成词法分析，实现命令表达式求值。

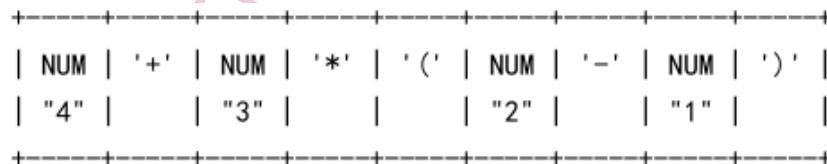


图 2: 词法分析

- 第 3 阶段：完成监视点的增加、删除、检查机制。
- 第 3 阶段：学习根据问题快速阅读手册的技能。
- 第 1, 2, 3 阶段：完成 NEMU 的调试指令构建与测试，包括 si/info/p/x/w/d。

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 3: NEMU 指令

## 三、阶段 1 实验方法与结果

### (一) 编码内容及测试结果

#### 1. 实现正确的寄存器结构体

在 PA0 中，由于没有正确实现模拟寄存器的结构体 CPU\_state，运行 NEMU 会出现 assertion fail 的错误信息。

观察 nemu/include/cpu/reg.h，看到寄存器的结构体是通过 struct 组织的，这样会导致每个 gpr 的 \_32, \_16, \_8，都分配开辟了空间，后续的 eax 等寄存器也都各自开辟了空间，这并不是我们想要实现的。通过 union 格式共用内存地址，解决这一问题。

修改后的代码如下：

```

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 { rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi; };
9     };
10    vaddr_t eip;
11 } CPU_state;

```

union 和 struct 类似，在句尾需要分号，在实验中曾经因为缺失分号出现 bug。

映射关系如下（以 eax 为例）：

31-28	27-24	23-20	19-16	15-12	11-8	7-4	3-0
eax(gpr[0]._32)							
						ax(gpr[0]._16)	
						ah(gpr[0]._8[1])	al(gpr[0]._8[0])

修改后的运行结果如图4所示：

```
latesoon@latesoon-virtual-machine:~/ics2017/nemu$ make run
+ CC src/monitor/cpu-exec.c
+ CC src/monitor/diff-test/diff-test.c
+ CC src/monitor/diff-test/gdb-host.c
+ CC src/monitor/diff-test/protocol.c
+ CC src/monitor/monitor.c
+ CC src/monitor/debug/watchpoint.c
+ CC src/monitor/debug/ui.c
+ CC src/monitor/debug/expr.c
+ CC src/main.c
+ CC src/memory/memory.c
+ CC src/cpu/exec/arith.c
+ CC src/cpu/exec/logic.c
+ CC src/cpu/exec/data-mov.c
+ CC src/cpu/exec/cc.c
+ CC src/cpu/exec/exec.c
+ CC src/cpu/exec/special.c
+ CC src/cpu/exec/control.c
+ CC src/cpu/exec/prefix.c
+ CC src/cpu/exec/system.c
+ CC src/cpu/intr.c
+ CC src/cpu/decode/modrm.c
+ CC src/cpu/decode/decode.c
+ CC src/cpu/reg.c
+ CC src/misc/logo.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: 20:22:03, Mar 2 2025
For help, type "help"
(nemu) 
```

图 4: 运行结果

可以看到，通过了寄存器的测试。

## 2. 实现单步执行

cpu\_exec 的函数是根据传入的参数执行至多 n 条指令。（该函数在本阶段的问答题部分中有详细介绍，不在此重复）

实现单步执行主要需要接收参数（可能存在），然后模仿 cmd\_c 的形式对 cpu\_exec 进行调用即可。

新增的函数声明和对 cmd\_table 结构体的修改如下。

```
1 static int cmd_si(char *args);
2
3 static struct {
4     ...
5 } cmd_table [] = {
6     ...
7     // 新增
8     { "si", "Step instruction by 1 or n", cmd_si }
9 }; 
```

实现的 cmd\_si 函数如下。首先设置 step 为 1，若传入了参数，则将参数转换为整型并更新 step，然后调用 cpu\_exec，向后执行对应数量的指令。

```

1 static int cmd_si(char *args) {
2     uint64_t step = 1;
3     char *arg = strtok(args, " ");
4     if(arg != NULL)
5         step = atoi(arg);
6     cpu_exec(step);
7     return 0;
8 }
```

测试结果如图5所示，在传入参数和不传入参数的情况下都能够正确运行，直到出现 HIT GOOD TRAP，测试成功。

```

latesoon@latesoon-virtual-machine:~/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: 20:22:03, Mar 2 2025
For help, type "help"
(nemu) si
100000: b8 34 12 00 00          movl $0x1234,%eax
(nemu) si 2
100005: b9 27 00 10 00          movl $0x100027,%ecx
10000a: 89 01                  movl %eax,(%ecx)
(nemu) si 3
10000c: 66 c7 41 04 01 00      movw $0x1,0x4(%ecx)
100012: bb 02 00 00 00          movl $0x2,%ebx
100017: 66 c7 84 99 00 e0 ff ff 01 00    movw $0x1,-0x2000(%ecx,%ebx,4)
(nemu) si
100021: b8 00 00 00 00          movl $0x0,%eax
(nemu) si
nemu: HIT GOOD TRAP at eip = 0x00100026
100026: d6                      nemu trap (eax = 0)
(nemu) q
```

图 5: 单步执行测试

### 3. 实现打印寄存器

新增的函数声明和对 cmd\_table 结构体的修改如下。

```

1 static int cmd_info(char *args);
2
3 static struct {
4     ...
5 } cmd_table [] = {
6     ...
7     //新增
8     { "info", "Register info", cmd_info}
9 };
```

实现的 cmd\_info 如下。目前只实现了 info r 的功能，首先检查参数是否缺失或错误，只有在指令为 info r 的时候才会打印全部寄存器（其中 08X 表示向前补 0，保持 8 位十六进制输出），否则返回对应错误。

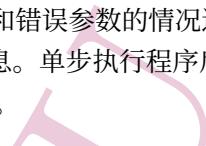
```

1 static int cmd_info(char *args) {
2     char *arg = strtok(args, " ");
```

```

3   if(arg == NULL){
4       printf("\033[1;31mMissing arg in cmd_info!\033[0m\n");
5       return 0;
6   }
7   if(strcmp(strtok(args, " "), "r")){
8       printf("\033[1;31mInvalid arg in cmd_info!\033[0m\n");
9       return 0;
10  }
11  printf("Registers info:\n");
12  printf("EAX:0x%08X ECX:0x%08X\n",cpu.eax,cpu.ecx);
13  printf("EDX:0x%08X EBX:0x%08X\n",cpu.edx,cpu.ebx);
14  printf("ESP:0x%08X EBP:0x%08X\n",cpu.esp,cpuebp);
15  printf("ESI:0x%08X EDI:0x%08X\n",cpu.esi,cpu.edi);
16  printf("EIP:0x%08X\n",cpu.eip);
17  return 0;
18 }
```

测试结果如图6所示，首先我对缺省参数和错误参数的情况进行了测试，正确触发了报错。然后正确的 info r 指令也正确打印出了对应信息。单步执行程序后，对应指令修改的寄存器 EAX 及指令寄存器 EIP 也得到了更新，测试成功。



```

latesoon@latesoon-virtual-machine:~/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: 20:22:03, Mar 2 2025
For help, type "help"
(nemu) info
Missing arg in cmd_info!
(nemu) info 123
Invalid arg in cmd_info!
(nemu) info r
Registers info:
EAX:0x3CE8C416 ECX:0x788647BC
EDX:0x12B0DFB7 EBX:0x5D0EAE67
ESP:0x48150A13 EBP:0x4D27D68C
ESI:0x639967A6 EDI:0x79B6ED1C
EIP:0x00100000
(nemu) si
    100000: b8 34 12 00 00          movl $0x1234,%eax
(nemu) info r
Registers info:
EAX:0x00001234 ECX:0x788647BC
EDX:0x12B0DFB7 EBX:0x5D0EAE67
ESP:0x48150A13 EBP:0x4D27D68C
ESI:0x639967A6 EDI:0x79B6ED1C
EIP:0x00100005
(nemu) q

```

图 6: 打印寄存器测试

#### 4. 实现扫描内存

新增的函数声明和对 cmd\_table 结构体的修改如下。

```

1 static int cmd_x(char *args);
2
3 static struct {
4     ...
5 } cmd_table [] = {
6     ...
}
```

```

7 //新增
8 { "x", "Get continuous data from address", cmd_x }
9 };

```

函数实现如下，首先还是进行参数的缺失和错误检查，然后将长度和地址进行字符串到数值的类型转换。然后就是遍历整个内存范围了，使用了实现的 vaddr\_read 函数，每行打印 8 个字节。

```

1 static int cmd_x(char *args){
2     char *arg1 = strtok(args, " ");
3     char *arg2 = strtok(NULL, " ");
4     if(arg1 == NULL || arg2 == NULL){
5         printf("\033[1;31mMissing arg in cmd x!\033[0m\n");
6         return 0;
7     }
8     if(atoi(arg1)<=0){
9         printf("\033[1;31mInvalid arg in cmd x!\033[0m\n");
10    return 0;
11 }
12 uint64_t len = atoi(arg1);
13 uint32_t addr = strtoul(arg2,NULL,16);
14 printf("Address      Data\n");
15 while(len){
16     int cnt = 8;
17     printf("0x%08X: ",addr);
18     while(cnt && len){
19         printf("%02X ",vaddr_read(addr,1));
20         addr++,cnt--,len--;
21     }
22     printf("\n");
23 }
24 return 0;
25 }

```

vaddr\_read 的函数调用如下，是一个内存访问操作，最终调用 pmem\_rw，一次返回 4 字节，根据我们传入的 len 进行分割。

```

1 uint32_t paddr_read(paddr_t addr, int len) {
2     return pmem_rw(addr, uint32_t) & (~0u >> ((4 - len) << 3));
3 }
4
5 uint32_t vaddr_read(vaddr_t addr, int len) {
6     return paddr_read(addr, len);
7 }

```

功能验证如图7所示，可以看到，对于错误和缺省参数都正确的返回了报错。对于正确的扫描，以 EIP 寄存器位置开始的 25 个字节为例进行验证，可以看到扫描内存指令与 si 指令返回的信息是相同的，验证成功。

```
latesoon@latesoon-virtual-machine:~/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: 20:22:03, Mar 2 2025
For help, type "help"
(nemu) X
Missing arg in cmd x!
(nemu) x -1 0x100
Invalid arg in cmd x!
(nemu) x 25 0x00100000
Address Data
0x00100000: B8 34 12 00 00 B9 27 00
0x00100008: 10 00 89 01 66 C7 41 04
0x00100010: 01 00 BB 02 00 00 00 66
0x00100018: C7
(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) q
```

图 7: 扫描内存验证

## (二) 问答题

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?

cmd\_c 的函数如下:

```
1 static int cmd_c(char *args) {
2     cpu_exec(-1);
3     return 0;
4 }
```

cpu\_exec 的函数如下:

```
1 void cpu_exec(uint64_t n) {
2     if (nemu_state == NEMU_END) {
3         printf("Program execution has ended. To restart the program, exit NEMU
4             and run again.\n");
5     }
6     nemu_state = NEMU_RUNNING;
7
8     bool print_flag = n < MAX_INSTR_TO_PRINT;
9
10    for (; n > 0; n--) {
11        /* Execute one instruction , including instruction fetch ,
12           * instruction decode, and the actual execution. */
13        exec_wrapper(print_flag);
14
15        ... //一些DEBUG和HAS_IOE的宏定义代码
16
17        if (nemu_state != NEMU_RUNNING) { return; }
18    }
19
20    if (nemu_state == NEMU_RUNNING) { nemu_state = NEMU_STOP; }
```

21 }

从这里可以看到，传入的参数 n 是执行指令的数量。cpu\_exec 在状态不是 NEMU\_END 的情况下，会将状态转换为 NEMU\_RUNNING，然后开始执行指令，若执行到某条指令后状态不再是 NEMU\_RUNNING，则直接返回，否则在 n 条指令结束完成后，将状态改为 NEMU\_STOP 并返回。

在 cmd\_table 中，cmd\_c 的定义如下，即继续程序的执行。

```
1 { "c", "Continue the execution of the program", cmd_c }
```

由于传入 n 的类型是 uint\_64，在 cmd\_c 中传入的 -1 会隐式转换为 0xFFFFFFFFFFFFFFF，这样会使程序可以继续执行无数条指令，直到 nemu 状态在执行时被修改，符合我们希望实现的功能。

2. 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？

这一种说法是一种简化的说法，但实际上，程序真正的结束是通过运行时环境调用的 exit，在操作系统层级进行释放，毕竟程序没有办法释放自己。

## 四、阶段 2 实验方法与结果

### (一) 编码内容

#### 1. 实现算术表达式的词法分析

首先需要在 expr.c 中添加语言规则，在枚举类型中添加更多 token，此处从 256 开始是为了避免与 ASCII 码重复。要添加的包括 AND, OR, NUM（此处表示十进制数），HEX（十六进制数），REG（寄存器），以及无符号加法，无符号减法以及解引用的乘号。

说明：

- 对于用一个符号表示的运算符直接用 ASCII 码作为其 token。
- 这里直接用加法减法乘法的 ASCII 码表示他们是双目运算符的情况。但在 rule 阶段识别时先不做区分，在存入 token 数组的时候进行处理。

然后添加新的 rule，注意单目的叹号一定要在！= 之后，因为规则是从上到下依次匹配，直到匹配成功为止。

```
1 enum {
2     TK_NOTYPE = 256, TK_EQ,
3
4     TK_NEQ, TK_AND, TK_OR, TK_NUM, TK_HEX, TK_REG, TK_UNARYPLUS, TK_UNARYSUB, TK_DEREF
5 };
6
7 static struct rule {
8     char *regex;
9     int token_type;
```

```

11 } rules [] = {
12
13     {"+", TK_NOTYPE}, // spaces
14     {"\\+", '+'}, // plus
15     {"==", TK_EQ}, // equal
16
17     {"\\-", '-'},
18     {"\\*", '*'},
19     {"/", '/'},
20     {"\\(", '('},
21     {"\\)", ')'},
22     {"!=", TK_NEQ},
23     {"&&", TK_AND},
24     {"\\|\\|", TK_OR},
25     {"\\$[a-zA-Z0-9]+", TK_REG},
26     {"0[xX][0-9a-fA-F]+", TK_HEX},
27     {"0[1-9][0-9]*", TK_NUM},
28     {"!", '!'}
29 };

```

然后，在 make\_token 函数中，对于每一个 rule 数组匹配的结果，将其保存在 token 列表中。

- 对于空格，不需要保存。
- 对于 +/\*，需要判断他们是不是单目运算符的情况，判断方法是，如果他们是第一个字符，或者他们的前一个字符不是右括号或者数字或者寄存器，那他们就是单目运算符，否则是双目运算符，将其类型保存至 token 数组中。
- 对于立即数/寄存器，需要把他们的类型和数值均保存在 token 中。
- 其他情况，将类型保存至 token 数组中。

```

1 static bool make_token(char *e) {
2
3     nr_token = 0;
4     while (e[position] != '\0') {
5         /* Try all rules one by one. */
6         for (i = 0; i < NR_REGEX; i++) {
7             if (regexec(&re[i], e + position, 1, &pmatch, 0) == 0 && pmatch.rm_so
8                 == 0) {
9                 ...
10                switch (rules[i].token_type) {
11                    case TK_NOTYPE:
12                        break;
13                    case '+':
14                        if (!nr_token || (tokens[nr_token - 1].type != ')' && tokens[nr_token
15                            - 1].type != TK_NUM
16                            && tokens[nr_token - 1].type != TK_HEX && tokens[nr_token - 1].type
17                            != TK_REG)) {

```

```
15         tokens[ nr_token ].type = TK_UNARYPLUS;
16     }
17     else{
18         tokens[ nr_token ].type = rules[ i ].token_type;
19     }
20     nr_token++;
21     break;
22 case '-':
23     if( !nr_token || (tokens[ nr_token -1].type != ')' && tokens[ nr_token -1].type != TK_NUM
24     && tokens[ nr_token -1].type != TK_HEX && tokens[ nr_token -1].type
25     != TK_REG) ){
26         tokens[ nr_token ].type = TK_UNARYSUB;
27     }
28     else{
29         tokens[ nr_token ].type = rules[ i ].token_type;
30     }
31     nr_token++;
32     break;
33 case '*':
34     if( !nr_token || (tokens[ nr_token -1].type != ')' && tokens[ nr_token -1].type != TK_NUM
35     && tokens[ nr_token -1].type != TK_HEX && tokens[ nr_token -1].type
36     != TK_REG) ){
37         tokens[ nr_token ].type = TK_DEREF;
38     }
39     else{
40         tokens[ nr_token ].type = rules[ i ].token_type;
41     }
42     nr_token++;
43     break;
44 case TK_HEX:
45     case TK_NUM:
46     case TK_REG:
47         strncpy(tokens[ nr_token ].str , substr_start , substr_len);
48         tokens[ nr_token ].str [substr_len]='\0';
49         default: //TODO();
50         tokens[ nr_token ].type = rules[ i ].token_type;
51         nr_token++;
52     }
53     break;
54     ...
55 }
56 return true;
57 }
```

## 2. 算符优先级设置

为节省查询时间开销,且由于 TK 编号不超过 300,新建一个全局数组 priority,在 make\_token 开始时进行初始化。各算符的优先级如下。

```

1 int priority[300];
2 static bool make_token(char *e) {
3     priority[TK_NUM] = priority[TK_HEX] = priority[TK_REG] = 1;
4     priority['('] = priority[')'] = 2;
5     priority[TK_UNARYPLUS] = priority[TK_UNARYSUB] = priority['!'] = priority[
6         TK_DEREF] = 3;
7     priority['*'] = priority['/'] = 4;
8     priority['+'] = priority['-'] = 5;
9     priority[TK_EQ] = priority[TK_NEQ] = 6;
10    priority[TK_AND] = 7;
11    priority[TK_OR] = 8;
12    ...
13 }
```

## 3. expr 求值入口函数的设置

建立一个新的递归求值函数 eval, 首先通过 make\_token 进行词法分析, 然后通过该函数进行递归求值。

```

1 uint32_t expr(char *e, bool *success) {
2     if (!make_token(e)) {
3         *success = false;
4         return 0;
5     }
6     return eval(0, nr_token - 1, success);
7 }
```

## 4. 括号匹配验证函数

这个函数判定的整个子表达式是否被同一组括号包围。逻辑较为简单, 首先需要最左边是左括号, 最右边是右括号, 然后用一个计数器实现, 左括号 +1, 右括号-1, 如果计数器在过程中清零, 则左右括号不是同一组括号, 如果全部遍历完之后计数器还没清零, 说明括号不匹配。在都满足的情况下返回 true, 否则 false。

```

1 bool check_par(uint32_t s, uint32_t e) {
2     if (tokens[s].type != '(' || tokens[e].type != ')')
3         return false;
4     int cnt = 0;
5     for (int now = s; now <= e ; now++){
6         if (tokens[now].type == '(')
7             cnt++;
8         else if (tokens[now].type == ')'){
9             cnt--;
10            if (cnt == 0 && now != e)
```

```

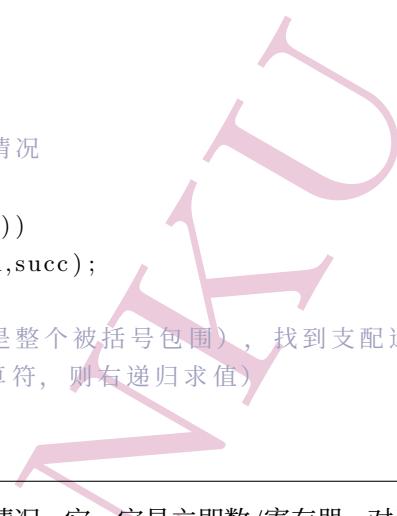
11     return false;
12 }
13 }
14 if (cnt) return false;
15 return true;
16 }
```

## 5. eval 递归求值函数

该函数的主要框架如下，分为不合法的情况（开始大于结束），处理单个运算符的情况，整个被括号包围（去掉括号进行递归），和处理子表达式且不是整个被括号包围的情况。

```

1 uint32_t eval(uint32_t s, uint32_t e, bool* succ){
2     if (!(*succ))
3         return 0;
4     if (s > e){
5         *succ = false;
6         return 0;
7     }
8     else if (s == e){
9         //处理单个运算符的情况
10    }
11    else if (check_par(s, e))
12        return eval(s+1, e-1, succ);
13    else{
14        //处理子表达式（不是整个被括号包围），找到支配运算符，然后对左右递归求值
15        //（如果是单目运算符，则右递归求值）
16    }
}
```



对于处理单个运算符的情况，它一定是立即数/寄存器。对于立即数，直接调用 strtoul 求值（需要 include 系统库），对于寄存器，首先我先将输入的寄存器名转为小写，然后依次遍历所有寄存器，若有名字匹配则成功求值，否则返回错误。

```

1 else if (s == e){
2     char reg[4];
3     switch(tokens[s].type){
4         case TK_HEX:
5             return strtoul(tokens[s].str, NULL, 16);
6         case TK_NUM:
7             return strtoul(tokens[s].str, NULL, 10);
8         case TK_REG:
9             for (int i=1; i<=4; i++){
10                 if (tokens[s].str[i] >= 'A' && tokens[s].str[i] <= 'Z')
11                     reg[i-1] = tokens[s].str[i] + 32;
12                 else if (tokens[s].str[i] >= 'a' && tokens[s].str[i] <= 'z')
13                     reg[i-1] = tokens[s].str[i];
14                 else if (tokens[s].str[i] == '\0'){
15                     reg[i-1] = tokens[s].str[i];
16                 }
17             }
18             return reg[3];
19         default:
20             return 0;
21     }
22 }
```

```

16         break;
17     }
18     else{
19         *succ = false;
20         return 0;
21     }
22 }
23 for( int i=0;i<8;i++){
24     if (!strcmp( reg , regsl[ i ] ))
25         return cpu.gpr[ i ]._32;
26     if (!strcmp( reg , regsw[ i ] ))
27         return cpu.gpr[ i ]._16;
28     if (!strcmp( reg , regsb[ i ] ))
29         return cpu.gpr[ i %4 ]._8[ i /4 ];
30 }
31 if (!strcmp( reg , "eip"))
32     return cpu.eip;
33 //all not match, also into default
34 default:
35     *succ = false;
36     return 0;
37 }
38 }
```

处理子表达式且不是整个被括号包围的情况如下（即 else 分支）。

- 找到支配运算符，首先在括号里的运算符一定不是支配运算符，然后比较优先级，对于双目运算符，同等优先级下最右边的是最后计算的，所以最右边的是支配运算符，对于单目运算符来说则是最左边的。（同时，因为在这个阶段维护了括号计数器，如果遍历结束之后括号不匹配，则直接返回 false）

```

1 int par = 0;
2 int mon = -1;
3 for( int now=s ; now<=e ; now++ ){
4     switch( tokens[ now ].type ){
5         case '(':
6             par++;
7             break;
8         case ')':
9             par--;
10            break;
11        default:
12            if( par )
13                break;
14            if( mon == -1 || ( priority[ tokens[ now ].type ] >= priority[ tokens[ mon ].type ] && priority[ tokens[ now ].type ] >3 )
15                || priority[ tokens[ now ].type ] > priority[ tokens[ mon ].type ] )
16                mon = now ;
17 }
```

```

18 }
19 if(par){
20     *succ = false;
21     return 0;
22 }
```

- 对于支配运算符是单目运算符的计算，先递归求取右边子表达式的结果，然后求取整个表达式的结果。

```

1 if(priority[tokens[mon].type] == 3){
2     uint32_t val = eval(mon+1,e,succ);
3     switch(tokens[mon].type){
4         case TK_UNARYPLUS:
5             return val;
6         case TK_UNARYSUB:
7             return -val;
8         case '!':
9             return !val;
10        case TK_DEREF:
11            return vaddr_read(val,4);
12        default:
13            *succ = false;
14            return 0;
15    }
16 }
```

- 对于支配运算符是双目运算符的情况，先递归求取左右两边子表达式的结果，然后求取整个表达式的结果。

```

1 else{
2     uint32_t val1 = eval(s,mon-1,succ);
3     uint32_t val2 = eval(mon+1,e,succ);
4     switch(tokens[mon].type){
5         case TK_AND:
6             return val1 && val2;
7         case TK_OR:
8             return val1 || val2;
9         case TK_EQ:
10            return val1 == val2;
11        case TK_NEQ:
12            return val1 != val2;
13        case '+':
14            return val1 + val2;
15        case '-':
16            return val1 - val2;
17        case '*':
18            return val1 * val2;
19        case '/':
20            return val1 / val2;
```

```

21     default:
22         *succ = false;
23         return 0;
24     }

```

## 6. ui.c 中 cmd\_p 的创建及 cmd\_x 的完善

cmd\_p 的建立如下，主要就是读取参数，然后直接传入 expr 计算，如果成功的话输出结果的 10 进制和 16 进制。

```

1 static int cmd_p(char *args);
2 static struct {
3     ...
4 } cmd_table [] = {
5     ...
6     { "p", "Calculate expr", cmd_p }
7 };
8 static int cmd_p(char *args){
9     if(args == NULL){
10         printf("\033[1;31mMissing arg in cmd p!\033[0m\n");
11         return 0;
12     }
13     bool succ = true;
14     uint32_t val = expr(args, &succ);
15     if(!succ){
16         printf("\033[1;31mInvalid arg in cmd p!\033[0m\n");
17         return 0;
18     }
19     printf("The answer is:\n");
20     printf("Base10:%d\n", val);
21     printf("Base16:0x%08X\n", val);
22     return 0;
23 }

```

cmd\_x 的完善如下，在阶段 1 中仅能读取普通的 16 进制立即数，现在可以读取 expr 进行计算了。

```

1 static int cmd_x(char *args){
2     char *arg1 = strtok(args, " ");
3     //char *arg2 = strtok(NULL, " ");      //删除
4     char *arg2 = args + strlen(arg1) + 1; //新增
5     ...
6     //uint32_t addr = strtoul(arg2, NULL, 16); //删除
7     bool succ = true;                      //新增
8     uint32_t addr = expr(arg2, &succ);      //新增
9     if(!succ){                            //新增
10         printf("\033[1;31mInvalid arg2 in cmd x!\033[0m\n");
11         return 0;
12     }

```

```

13 ...
14 }

```

## (二) 测试结果

对缺失和错误表达式的测试如图8所示。

```

latesoon@latesoon-virtual-machine:~/ics2017/nemu$ make run
+ CC src/monitor/debug/expr.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: 20:22:03, Mar 2 2025
For help, type "help"
(nemu) p (2+3)-(4+(5+6)
[src/monitor/debug/expr.c,103,make_token] match rules[6] = "\(" at position 0 with len 1: (
[src/monitor/debug/expr.c,103,make_token] match rules[13] = "0|[1-9][0-9]*" at position 1 with len 1: 2
[src/monitor/debug/expr.c,103,make_token] match rules[1] = "\+" at position 2 with len 1: +
[src/monitor/debug/expr.c,103,make_token] match rules[13] = "0|[1-9][0-9]*" at position 3 with len 1: 3
[src/monitor/debug/expr.c,103,make_token] match rules[7] = "\)" at position 4 with len 1: )
[src/monitor/debug/expr.c,103,make_token] match rules[3] = "\-" at position 5 with len 1: -
[src/monitor/debug/expr.c,103,make_token] match rules[6] = "\(" at position 6 with len 1: (
[src/monitor/debug/expr.c,103,make_token] match rules[13] = "0|[1-9][0-9]*" at position 7 with len 1: 4
[src/monitor/debug/expr.c,103,make_token] match rules[1] = "\+" at position 8 with len 1: +
[src/monitor/debug/expr.c,103,make_token] match rules[6] = "\(" at position 9 with len 1: (
[src/monitor/debug/expr.c,103,make_token] match rules[13] = "0|[1-9][0-9]*" at position 10 with len 1: 5
[src/monitor/debug/expr.c,103,make_token] match rules[1] = "\+" at position 11 with len 1: +
[src/monitor/debug/expr.c,103,make_token] match rules[13] = "0|[1-9][0-9]*" at position 12 with len 1: 6
[src/monitor/debug/expr.c,103,make_token] match rules[7] = "\)" at position 13 with len 1: )
Invalid arg in cmd p!
(nemu) p
Missing arg in cmd p!
(nemu) p $ABC-SDEF
[src/monitor/debug/expr.c,103,make_token] match rules[11] = "\$\{a-zA-Z0-9\}+" at position 0 with len 4: $ABC
[src/monitor/debug/expr.c,103,make_token] match rules[3] = "\-" at position 4 with len 1: -
[src/monitor/debug/expr.c,103,make_token] match rules[11] = "\$\{a-zA-Z0-9\}+" at position 5 with len 4: $DEF
Invalid arg in cmd p!
(nemu) p $$$
no match at position 0
$$$^
Invalid arg in cmd p!
(nemu) q

```

图 8: 缺失错误表达式测试

讨论了词法分析错误，求值过程中的寄存器无法识别及括号不匹配等错误。可以看到，均能安全的返回错误，不会导致系统崩溃。

对运算符递归求值，单目运算符的测试如图9和图10所示。

```

latesoon@latesoon-virtual-machine:~/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: 20:22:03, Mar 2 2025
For help, type "help"
(nemu) p (3==4 && (5!=6))+7+-(+---++-18)
[src/monitor/debug/expr.c,103,make_token] match rules[6] = "\(" at position 0 with len 1: (
[src/monitor/debug/expr.c,103,make_token] match rules[13] = "0|[1-9][0-9]*" at position 1 with len 1: 3
[src/monitor/debug/expr.c,103,make_token] match rules[2] = "==" at position 2 with len 2: ==
[src/monitor/debug/expr.c,103,make_token] match rules[13] = "0|[1-9][0-9]*" at position 4 with len 1: 4
[src/monitor/debug/expr.c,103,make_token] match rules[0] = "+" at position 5 with len 1:
[src/monitor/debug/expr.c,103,make_token] match rules[9] = "&&" at position 6 with len 2: &&
[src/monitor/debug/expr.c,103,make_token] match rules[0] = "+" at position 8 with len 1:
[src/monitor/debug/expr.c,103,make_token] match rules[6] = "\(" at position 9 with len 1: (
[src/monitor/debug/expr.c,103,make_token] match rules[13] = "0|[1-9][0-9]*" at position 10 with len 1: 5
[src/monitor/debug/expr.c,103,make_token] match rules[8] = "!=" at position 11 with len 2: !=
[src/monitor/debug/expr.c,103,make_token] match rules[13] = "0|[1-9][0-9]*" at position 13 with len 1: 6
[src/monitor/debug/expr.c,103,make_token] match rules[14] = "\)" at position 14 with len 1: )
[src/monitor/debug/expr.c,103,make_token] match rules[7] = "\)" at position 15 with len 1: )
[src/monitor/debug/expr.c,103,make_token] match rules[1] = "\+" at position 16 with len 1: +
[src/monitor/debug/expr.c,103,make_token] match rules[13] = "0|[1-9][0-9]*" at position 17 with len 1: 7
[src/monitor/debug/expr.c,103,make_token] match rules[1] = "\+" at position 18 with len 1: +
[src/monitor/debug/expr.c,103,make_token] match rules[3] = "\-" at position 19 with len 1: -
[src/monitor/debug/expr.c,103,make_token] match rules[6] = "\(" at position 20 with len 1: (
[src/monitor/debug/expr.c,103,make_token] match rules[1] = "\+" at position 21 with len 1: +
[src/monitor/debug/expr.c,103,make_token] match rules[1] = "\+" at position 22 with len 1: +
[src/monitor/debug/expr.c,103,make_token] match rules[3] = "\-" at position 23 with len 1: -
[src/monitor/debug/expr.c,103,make_token] match rules[3] = "\-" at position 24 with len 1: -
[src/monitor/debug/expr.c,103,make_token] match rules[1] = "\+" at position 25 with len 1: +
[src/monitor/debug/expr.c,103,make_token] match rules[1] = "\+" at position 26 with len 1: +
[src/monitor/debug/expr.c,103,make_token] match rules[3] = "\-" at position 27 with len 1: -
[src/monitor/debug/expr.c,103,make_token] match rules[13] = "0|[1-9][0-9]*" at position 28 with len 2: 18
[src/monitor/debug/expr.c,103,make_token] match rules[7] = "\)" at position 30 with len 1: )
The answer is:
Base10:25
Base16:0x00000019
(nemu) q

```

图 9: 复杂运算符/单目运算符测试

```

(nemu) p !!!(-3)
[src/monitor/debug/expr.c,103,make_token] match rules[14] = "!" at position 0 with len 1: !
[src/monitor/debug/expr.c,103,make_token] match rules[14] = "!" at position 1 with len 1: !
[src/monitor/debug/expr.c,103,make_token] match rules[14] = "!" at position 2 with len 1: !
[src/monitor/debug/expr.c,103,make_token] match rules[6] = "\(" at position 3 with len 1: (
[src/monitor/debug/expr.c,103,make_token] match rules[3] = "\-" at position 4 with len 1: -
[src/monitor/debug/expr.c,103,make_token] match rules[13] = "0|[1-9][0-9]*" at position 5 with len 1: 3
[src/monitor/debug/expr.c,103,make_token] match rules[7] = "\)" at position 6 with len 1: )
The answer is:
Base10:0
Base16:0x00000000
(nemu) q

```

图 10: 单目运算符测试

成功通过测试，运算结果正确。

对寄存器求值，解引用，16 进制数，负数运算测试如图11所示。

```

latesoon@latesoon-virtual-machine:~/lcs2017/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: 20:22:03, Mar 2 2025
For help, type "help"
(nemu) p *$eip+$bh -0x10+5
[src/monitor/debug/expr.c,103,make_token] match rules[4] = "\*" at position 0 with len 1: *
[src/monitor/debug/expr.c,103,make_token] match rules[11] = "\[$[a-zA-Z0-9]+"
[src/monitor/debug/expr.c,103,make_token] match rules[1] = "\+" at position 5 with len 1: +
[src/monitor/debug/expr.c,103,make_token] match rules[11] = "\[$[a-zA-Z0-9]+"
[src/monitor/debug/expr.c,103,make_token] match rules[0] = "+" at position 6 with len 3: $bh
[src/monitor/debug/expr.c,103,make_token] match rules[3] = "\-" at position 9 with len 1:
[src/monitor/debug/expr.c,103,make_token] match rules[13] = "\0|[1-9][0-9]*"
[src/monitor/debug/expr.c,103,make_token] match rules[12] = "\0|[xX][0-9a-fA-F]*"
[src/monitor/debug/expr.c,103,make_token] match rules[1] = "\+" at position 15 with len 1: +
[src/monitor/debug/expr.c,103,make_token] match rules[13] = "\0|[1-9][0-9]*"
The answer is:
Base10:1193198
Base16:0x001234EE
(nemu) info r
Registers info:
EAX:0x475F8F9C ECX:0x7035DDC3
EDX:0x338CAF65 EBX:0xE594161
ESP:0x7A9F21B2 EBP:0x7BA0344E
ESI:0x28BC767A EDI:0x14C58617
EIP:0x00100000
(nemu) si
    100000: b8 34 12 00 00          movl $0x1234,%eax
(nemu) p (-3)*(-5)*(-7)
[src/monitor/debug/expr.c,103,make_token] match rules[6] = "\(" at position 0 with len 1: (
[src/monitor/debug/expr.c,103,make_token] match rules[3] = "\-" at position 1 with len 1: -
[src/monitor/debug/expr.c,103,make_token] match rules[13] = "\0|[1-9][0-9]*"
[src/monitor/debug/expr.c,103,make_token] match rules[7] = "\)" at position 3 with len 1: )
[src/monitor/debug/expr.c,103,make_token] match rules[4] = "\*" at position 4 with len 1: *
[src/monitor/debug/expr.c,103,make_token] match rules[6] = "\(" at position 5 with len 1: (
[src/monitor/debug/expr.c,103,make_token] match rules[3] = "\-" at position 6 with len 1: -
[src/monitor/debug/expr.c,103,make_token] match rules[15] = "\0|[1-9][0-9]*"
[src/monitor/debug/expr.c,103,make_token] match rules[7] = "\)" at position 7 with len 1: )
[src/monitor/debug/expr.c,103,make_token] match rules[4] = "\*" at position 8 with len 1: *
[src/monitor/debug/expr.c,103,make_token] match rules[6] = "\(" at position 9 with len 1: (
[src/monitor/debug/expr.c,103,make_token] match rules[3] = "\-" at position 10 with len 1: -
[src/monitor/debug/expr.c,103,make_token] match rules[13] = "\0|[1-9][0-9]*"
[src/monitor/debug/expr.c,103,make_token] match rules[7] = "\)" at position 11 with len 1: )
[src/monitor/debug/expr.c,103,make_token] match rules[13] = "\0|[1-9][0-9]*"
The answer is:
Base10:-105
Base16:0xFFFFFFF97
(nemu) q

```

图 11: 寄存器/解引用/16 进制数/负数运算测试

第二个样例显然正确，第一个样例是将 eip 的解引用结果 0x001234B8（通过下面 si 指令观察到），与 bh 结果 0x41 相加，再减 16，再加 5，结果是 0x001234EE，正确。

对 cmd\_x 完善的追加测试如图12所示。

```

latesoon@latesoon-virtual-machine:~/lcs2017/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: 20:22:03, Mar 2 2025
For help, type "help"
(nemu) x 25 $eip +4 +$eIP-$EIP
[src/monitor/debug/expr.c,103,make_token] match rules[11] = "\[$[a-zA-Z0-9]+"
[src/monitor/debug/expr.c,103,make_token] match rules[0] = "+" at position 4 with len 1:
[src/monitor/debug/expr.c,103,make_token] match rules[1] = "\+" at position 5 with len 1: +
[src/monitor/debug/expr.c,103,make_token] match rules[13] = "\0|[1-9][0-9]*"
[src/monitor/debug/expr.c,103,make_token] match rules[0] = "+" at position 6 with len 1: 4
[src/monitor/debug/expr.c,103,make_token] match rules[1] = "\+" at position 7 with len 1:
[src/monitor/debug/expr.c,103,make_token] match rules[1] = "\+" at position 8 with len 1: +
[src/monitor/debug/expr.c,103,make_token] match rules[11] = "\[$[a-zA-Z0-9]+"
[src/monitor/debug/expr.c,103,make_token] match rules[3] = "\-" at position 9 with len 4: $eIP
[src/monitor/debug/expr.c,103,make_token] match rules[11] = "\[$[a-zA-Z0-9]+"
Address      Data
0x00100004: 00 B9 27 00 10 00 89 01
0x0010000C: 66 C7 41 04 01 00 BB 02
0x00100014: 00 00 00 66 C7 84 99 00
0x0010001C: E0
(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) q

```

图 12: cmd\_x 完善测试

可以看到，当前 x 指令的第二个参数已经可以正确计算算数表达式（运算结果为 \$eip+4），

与下面 si 5 的结果对比，可以看到结果正确。同时，系统能够正确识别大小写的寄存器，也能够正确处理空格。

## 五、阶段 3 实验方法与结果

### (一) 编码内容

#### 1. 实现监视点池的管理

首先需要构造 new\_wp 函数，该函数用于找到空闲监视点链表的编号最小的结点（由于两个链表都是按照编号从小到大管理的，所以直接取出 free 链表的头结点。）然后遍历 head 链表，将结点插到对应位置以保证链表结点按编号升序连接，最后将结点返回。

在其中也添加了 assert(0) 等错误判定机制，以防程序崩溃。

```

1 WP* new_wp() {
2     if (free_ == NULL)
3         return NULL;
4     WP* now = free_;
5     int no = now->NO;
6     free_ = free_->next;
7     if (head == NULL){
8         now->next = NULL;
9         head = now;
10        return now;
11    }
12    if (no < head->NO){
13        now->next = head;
14        head = now;
15        return now;
16    }
17    WP* temp = head;
18    while (temp != NULL){
19        if (temp->next == NULL || temp->next->NO > no){
20            now->next = temp->next;
21            temp->next = now;
22            return now;
23        }
24        temp = temp->next;
25    }
26    assert(0);
27    return NULL;
28 }
```



在 free\_wp 中，原理类似。根据传入的 wp 结点在链表中找到，并将其删除，再将其插入对应的空闲链表中。这里我认为传入的 wp 结点一定正在被使用，对于传入参数不合法的情况在外层函数中进行判定，因此这里的错误直接进行 assert(0) 处理。

```

1 void free_wp(WP* wp){
2     //when in this funtion , wp must be valid
```

```

3   if(head == NULL)
4       assert(0);
5   if(wp == head)
6       head = head->next;
7   else{
8       WP* tmp = head;
9       while(1){
10          if(tmp->next == NULL)
11              assert(0);
12          else if (tmp->next == wp){
13              tmp->next = tmp->next->next;
14              break;
15          }
16          else
17              tmp = tmp->next;
18      }
19  }
20  if(free_ == NULL){
21      wp->next = NULL;
22      free_ = wp;
23      return;
24  }
25  if(wp->NO < free_->NO){
26      wp->next = free_;
27      free_ = wp;
28      return;
29  }
30  WP* temp = free_;
31  while(temp != NULL){
32      if(temp->next == NULL || temp->next->NO > wp->NO){
33          wp->next = temp->next;
34          temp->next = wp;
35          break;
36      }
37      temp = temp->next;
38  }
39 }
```



## 2. 实现监视点

要完成监视点的求值，首先需要修改监视点结构体。新增两个成员变量，val 表示表达式的旧值，exp 存储表达式，以便后续重新求值比较。

```

1  typedef struct watchpoint {
2      int NO;
3      struct watchpoint *next;
4      uint32_t val;    //new
5      char exp[1000]; //new
```

```
6 } WP;
```

在 watchpoint.c 中实现几个对外的接口函数。包括 wp\_check (用于 cpu\_exec.c 中，执行完一条语句就检查所有现有监视点是否变化)， wp\_print (与 info w 对接，打印全部监视点)， wp\_add (与 w 指令对接，求值并增加监视点)， wp\_del (与 d 指令对接，删除监视点)。将这些函数添加到 watchpoint.h 头文件中，并在对应文件中 include 该头文件。

在 wp\_check 中，遍历全部监视点，打印并更新所有结果有变化的监视点（如有）。返回值表示了是否有监视点更新。

```
1 bool wp_check() {
2     bool update = false;
3     WP* temp = head;
4     while(temp != NULL){
5         bool success = true;
6         uint32_t num = expr(temp->exp,&success);
7         if(num != temp->val){
8             if(!update){
9                 update = true;
10                printf("These watchpoint has been changed:\n");
11                printf("No Stored_value           Now_value           Expr\n");
12            }
13            printf("%2d %010d(0x%08X) %010d(0x%08X) %s\n",temp->NO,temp->val,temp->
14               val,num,num,temp->exp);
15            temp->val = num;
16        }
17        temp = temp -> next;
18    }
19    return !update;
}
```

如果发现检查点更新，在 cpu\_exec.c 中立即将 NEMU 状态切换为 STOP，停止运行。

```
1 if (!wp_check())
2     nemu_state = NEMU_STOP;
```

打印函数比较简单，遍历整个监视点链表并打印就可以。

```
1 void wp_print(){
2     if(head == NULL){
3         printf("No watchpoint exists.\n");
4         return;
5     }
6     printf("No Stored_value           Expr\n");
7     WP* temp = head;
8     while(temp != NULL){
9         printf("%2d %010d(0x%08X) %s\n",temp->NO,temp->val,temp->val,temp->exp);
10        temp = temp -> next;
11    }
12 }
```

相应的，修改了 cmd\_info 函数，对整体逻辑有一些调整，增加了 info w 分支。

```

1 ...
2 if (!strcmp (strtok (args, " "), "r")){
3     ...
4     return 0;
5 }
6 if (!strcmp (strtok (args, " "), "w")){
7     printf ("Watchpoints info:\n");
8     wp_print ();
9     return 0;
10}
11 printf ("\033[1;31mInvalid arg in cmd info!\033[0m\n");
12 return 0;

```

wp\_add 函数实现表达式求值及保存，通过 new\_wp 调用获得一个新监视点，然后存储对应信息并打印。

```

1 void wp_add (char* arg){
2     bool success = true;
3     uint32_t num = expr (arg, &success);
4     if (!success){
5         printf ("\033[1;31mInvalid arg in watchpoint addition!\033[0m\n");
6         return;
7     }
8     WP* new = new_wp ();
9     if (new == NULL){
10        printf ("\033[1;31mNo enough watchpoint to be added!\033[0m\n");
11        return;
12    }
13    new->val = num;
14    strcpy (new->exp, arg);
15    printf ("Success! New watchpoint info:\n");
16    printf ("No Stored_value Expr\n");
17    printf ("%2d %010d(0x%08X) %s\n", new->NO, new->val, new->val, new->exp);
18 }

```

由于把工作都交给 wp\_add 完成，cmd\_w 中只需要接收参数，然后传递就可以。

```

1 static int cmd_w (char *args){
2     if (args == NULL){
3         printf ("\033[1;31mMissing arg in cmd p!\033[0m\n");
4         return 0;
5     }
6     wp_add (args);
7     return 0;
8 }

```

在 wp\_del 中，根据编号删除对应的监视点。在开头进行一次初步筛选，也就是当 no 传入的内容明显不合法或者当前没有监视点的时候直接返回错误，然后遍历整个链表查找，若找到，调用 free\_wp 删除，若完成遍历还没找到，报错（通过 success 变量管理）。

```

1 void wp_del(uint32_t no){
2     if(no >= NR_WP || head == NULL){
3         printf("\033[1;31mInvalid arg in watchpoint deletion!\033[0m\n");
4         return;
5     }
6     bool success = false;
7     WP* temp = head;
8     while(temp != NULL){
9         if(temp->NO == no){
10             free_wp(temp);
11             success = true;
12             printf("Successful watchpoint deletion!\n");
13             break;
14         }
15         temp = temp -> next;
16     }
17     if(!success)
18         printf("\033[1;31mInvalid arg in watchpoint deletion!\033[0m\n");
19 }
```

cmd\_d 也是一样，完成参数类型转换后直接交给 wp\_del。

```

1 static int cmd_d(char *args){
2     char *arg = strtok(args, " ");
3     if(arg != NULL)
4         wp_del(atoi(arg));
5     else
6         printf("\033[1;31mMissing arg in cmd p!\033[0m\n");
7     return 0;
8 }
```

## (二) 测试结果

注：在这部分测试后，由于词法分析功能已经完备，关闭词法分析部分的 log 使得界面更清晰。

首先进行监视点运行的测试，我添加一个 \$eip==0x10000a 的监视点，以期待它在执行两条语句后停下。我们通过不断运行命令 c 继续程序。根据运行结果可以发现，该监视点的结果被刷新的两次，依次是 eip 到达该位置，从 0 到 1，另一次是执行完该条语句，从 1 到 0。然后就直接运行到了 GOOD TRAP 位置，实现正确。

```

latesoon@latesoon-virtual-machine:~/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: 20:22:03, Mar 2 2025
For help, type "help"
(nemu) w $eip == 0x10000a
Success! New watchpoint info:
No Stored_value Expr
0 0000000000(0x00000000) $eip == 0x10000a
(nemu) info w
Watchpoints info:
No Stored_value Expr
0 0000000000(0x00000000) $eip == 0x10000a
(nemu) c
These watchpoint has been changed:
No Stored_value Now_value Expr
0 0000000000(0x00000000) 0000000001(0x00000001) $eip == 0x10000a
(nemu) c
These watchpoint has been changed:
No Stored_value Now_value Expr
0 0000000001(0x00000001) 0000000000(0x00000000) $eip == 0x10000a
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100026
(nemu) q

```

图 13: 监视点运行测试

然后进行监视点反复增加/删除的测试。

第一组测试展示了删除中间监视点，删除头监视点以及删除后重新分配监视点的情况，可以看到，均正确完成。

```

latesoon@latesoon-virtual-machine:~/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: 20:22:03, Mar 2 2025
For help, type "help"
(nemu) w 1==1
Success! New watchpoint info:
No Stored_value Expr
0 0000000001(0x00000001) 1==1
(nemu) w 2==2
Success! New watchpoint info:
No Stored_value Expr
1 0000000001(0x00000001) 2==2
(nemu) w 3!=3
Success! New watchpoint info:
No Stored_value Expr
2 0000000000(0x00000000) 3!=3
(nemu) info w
Watchpoints info:
No Stored_value Expr
0 0000000001(0x00000001) 1==1
1 0000000001(0x00000001) 2==2
2 0000000000(0x00000000) 3!=3
(nemu) d 1
Successful watchpoint deletion!
(nemu) d 0
Successful watchpoint deletion!
(nemu) w 4!=4
Success! New watchpoint info:
No Stored_value Expr
0 0000000000(0x00000000) 4!=4
(nemu) info w
Watchpoints info:
No Stored_value Expr
0 0000000000(0x00000000) 4!=4
2 0000000000(0x00000000) 3!=3
(nemu) q

```

图 14: 监视点添加/删除测试 1

这部分进行了尾监视点删除，更进一步的分配/删除交替操作，以及当 d 和 w 指令传入错误

```

latesoon@latesoon-virtual-machine:~/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: 20:22:03, Mar 2 2025
For help, type "help"
(nemu) w (-3)*(+5)
Success! New watchpoint info:
No Stored_value Expr
0 -000000015(0xFFFFFFFF1) (-3)*(+5)
(nemu) w 12345
Success! New watchpoint info:
No Stored_value Expr
1 00000012345(0x000003039) 12345
(nemu) w 333
Success! New watchpoint info:
No Stored_value Expr
2 0000000333(0x0000014D) 333
(nemu) d 0
Successful watchpoint deletion!
(nemu) d 0
Invalid arg in watchpoint deletion!
(nemu) d 11
Invalid arg in watchpoint deletion!
(nemu) d
Missing arg in cmd d!
(nemu) w
Missing arg in cmd w!
(nemu) d 2
Successful watchpoint deletion!
(nemu) w 111
Success! New watchpoint info:
No Stored_value Expr
0 0000000111(0x00000006F) 111
(nemu) w 111
Success! New watchpoint info:
No Stored_value Expr
2 0000000111(0x00000006F) 111
(nemu) w 111
Success! New watchpoint info:
No Stored_value Expr
3 0000000111(0x00000006F) 111
(nemu) info w
Watchpoints info:
No Stored_value Expr
0 0000000111(0x00000006F) 111
1 00000012345(0x000003039) 12345
2 0000000111(0x00000006F) 111
3 0000000111(0x00000006F) 111
(nemu) q

```

图 15: 监视点添加/删除测试 2

参数的处理（不再测试 expr 求值的正确性，主要确认删除结点不存在/不合法的情况以及参数缺失的情况）。对于所有测试要求，程序均正确完成。

### (三) 问答题

1. 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 会将变量的作用域限制在本文件中，将 wp\_pool, head 和 free\_ 指针使用关键字 static，最重要的作用就是保证他们不会被其他文件修改。因为监视点这个模块一定会被外部所引用，通过这样的形式保证这些变量只被本文件修改，可以便于后续的维护和 debug。此外，head 和 free\_ 这种变量名在很多情况下都比较容易用到，加入 static 也能避免与其他文件的命名冲突。

2. 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？

这并不是必须的，如果将 int3 指令的长度变成 2 个字节，仍然可以正常工作。但是，如果仅仅更改断点指令而不进行相关其他调整的话，可能会引发一些问题而导致无法正常工作。

在 x86 架构中，int3 指令的长度为 1 个字节。然而，x86 是允许可变指令长度的架构，即使把指令长度变成 2 个字节，理论上也是可以正常执行的。由于断点的触发机制是将原指令进行覆盖，如果原指令只有 1 个字节，这样的做法可能会覆盖下一条指令的第 1 个字节。必须要优化指令继续执行的位置，可能需要把断点后的指令进行跳转，而不能再和之前一样进行简单覆盖，否则会导致污染下一条指令，或者无法为该单字节指令和下一条指令同时添加断点的问题。

3. 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。

在我们的 nemu 中测试该功能，可以看到前五条指令如下。

```
latesoon@latesoon-virtual-machine:~/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: 20:22:03, Mar 2 2025
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) q
```

图 16: 前五条指令

将用监视点模拟的断点机制在 0x100008 处增加一个断点，可以看到程序并没有停止。

```
latesoon@latesoon-virtual-machine:~/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: 20:22:03, Mar 2 2025
For help, type "help"
(nemu) w 0x100008
Success! New watchpoint info:
No Stored_value           Expr
_0 0001048584(0x00100008) 0x100008
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100026

(nemu) q
```

图 17: 断点非首字节模拟

程序正常执行完毕，没有触发断点。但这是由于我们的监视点机制并不会对程序造成修改。根据断点的原本机制，会将对应位置的内容直接覆写成断点。这种情况与正常使用断点机制不同，系统可能不会将其处理为操作码，而是将其处理为操作数，使得断点并不能生效，并因此引发意外之外的程序错误或者引发意外的结果。

4. 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(NJUDebugger), 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?

模拟器: 模拟目标硬件的完整环境 (如 CPU、内存、外设等), 需要实现目标硬件的完整行为, 可以用于测试、开发和调试等。

调试器: 直接运行在宿主机的硬件和操作系统上, 用于控制和分析程序的执行, 通常依赖于宿主机的硬件和操作系统, 帮助程序员更好的测试程序以及查找程序中的错误。

如图18所示, 与 NEMU 直接在模拟的环境中调试不同, GDB 不会模拟计算机环境, 而是直接运行在宿主机的操作系统上, 利用操作系统的调试接口来控制目标程序的进程, 通过直接修改机器码增加断点, 通过控制通过控制 CPU 的指令指针实行单步执行等操作。

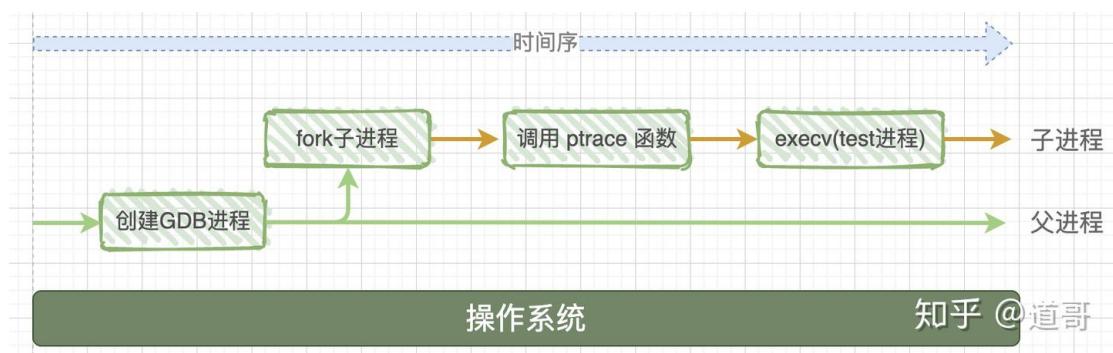


图 18: GDB 调试程序

5. 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?

图19展示了查找 selector 的结果。

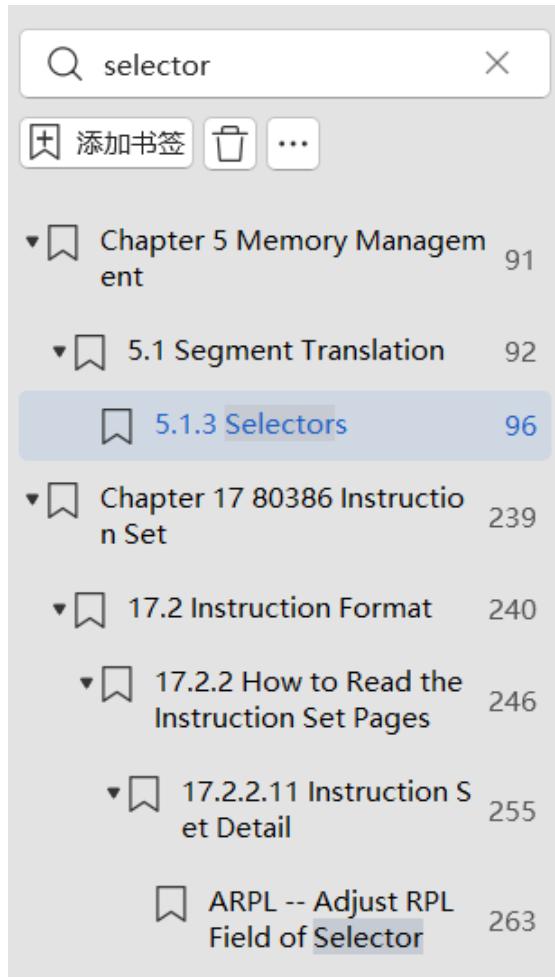


图 19: select 查找结果

根据该结果，可以认为想了解该概念主要需要阅读的是 5.1.3 节，如果需要了解调整 Selector 的字段等信息，可以补充阅读 17.2.2.11。

## 六、必答题

### (一) Question 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。

#### 1. EFLAGS 寄存器中的 CF 位是什么意思？

在手册中查找 EFLAGS，虽然没能在目录直接找到，看到有两个 Figure 包括该关键词，即：

- |              |   |
|--------------|---|
| <sup>1</sup> | 2-8      EFLAGS Register                  |
| <sup>2</sup> | 4-1      Systems Flags of EFLAGS Register |

Figure 2-8. EFLAGS Register

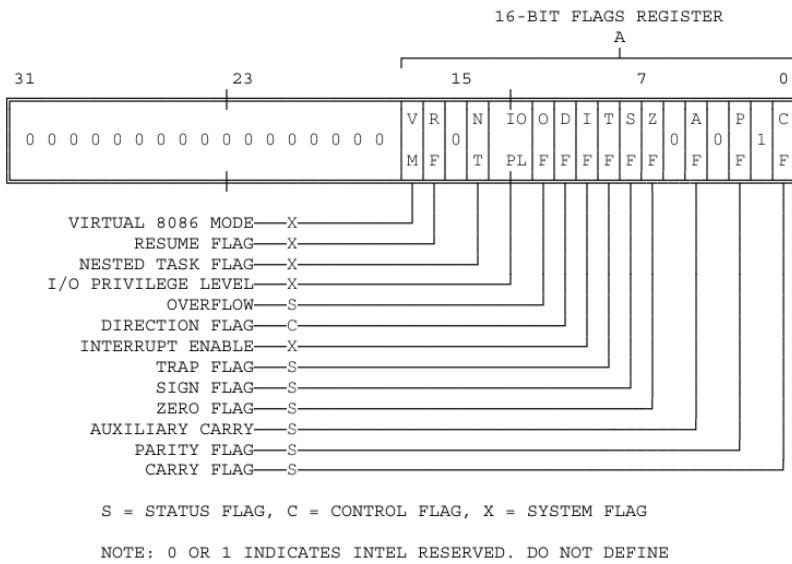


图 20: EFLAGS

从该图中可以看出，CF 是 STATUS FLAG 中的 Carry Flag。因为确定了不是 SYSTEM FLAG 所以不再需要阅读 4-1。

以下是 2.3.4.1 中对该类别 Flag 的介绍。

<sup>1</sup> 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.

这里看到对他们的详细说明指向了附录 C。

实际需要阅读的范围是：

<sup>1</sup> 2.3.4 Flags Register  
<sup>2</sup> Appendix C

## 2. ModR/M 字节是什么？

从目录中查找 ModR/M，可以看到：

<sup>1</sup> 17.2.1 ModR/M and SIB Bytes

阅读该范围即可，了解到其主要包括指令中要使用的索引类型或寄存器编号，要使用的寄存器，或用于选择指令的更多信息，基址、索引和比例信息。包括 MOD，REG，R/M 字段等。

## 3. mov 指令的具体格式是怎么样的？

我们寻找指令细节，看到目录中包括 17.2.2.11 这一部分，然后其中包括了 MOV 的细节展示。

MOV — Move Data			
Opcde	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
C1411114	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

图 21: MOV(1)

Opcde	Instruction	Clocks	Description
OF 20 /r	MOV r32,CR0/CR2/CR3	6	Move (control register) to (register)
OF 22 /r	MOV CR0/CR2/CR3,r32	10/4/5	Move (register) to (control register)
OF 21 /r	MOV r32,DR0 -- 3	22	Move (debug register) to (register)
OF 21 /r	MOV r32,DR6/DR7	14	Move (debug register) to (register)
OF 23 /r	MOV DR0 -- 3,r32	22	Move (register) to (debug register)
OF 23 /r	MOV DR6/DR7,r32	16	Move (register) to (debug register)
OF 24 /r	MOV r32,TR6/TR7	12	Move (test register) to (register)
OF 26 /r	MOV TR6/TR7,r32	12	Move (register) to (test register)

图 22: MOV(2)

1 17.2 INSTRUCTION FORMAT  
 2 17.2.2.11 Instruction Set Detail  
 3 MOV Move Data.  
 4 MOV Move to/from Special Registers

具体内容如图21和图22所示。

## (二) Question 2

- 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?

总共有 3972 行代码。

使用命令:

```
1 find . -name "*.c" -o -name "*.h" | xargs cat | wc -l
```

find 指令统计所有名字以.c 和.h 结尾的文件, xargs cat 把文件内容输出到标准输出, wc -l 表示只统计行数。

```
latesoon@latesoon-virtual-machine:~/ics2017/nemu$ find . -name "*.c" -o -name "*.
.h" | xargs cat | wc -l
3972
```

图 23: 统计代码行数

2. 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”?)

可以切换回 master 分支查看。

```
latesoon@latesoon-virtual-machine:~/ics2017/nemu$ git branch
  master
  pa0
* pa1
latesoon@latesoon-virtual-machine:~/ics2017/nemu$ git checkout master
切换到分支 'master'
您的分支领先 'origin/2017' 共 8 个提交。
(使用 "git push" 来发布您的本地提交)
latesoon@latesoon-virtual-machine:~/ics2017/nemu$ find . -name "*.c" -o -name "*.h" | xargs cat | wc -l
3487
```

图 24: PA1 之前的行数

当时是 3487 行, 共增加了 485 行。

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.

在 Makefile 中添加:

```
count:
    find . -name "*.c" -o -name "*.h" | xargs cat | wc -l
```

图 25: Makefile

输入 make count 测试, 正确。

```
latesoon@latesoon-virtual-machine:~/ics2017/nemu$ make count
find . -name "*.c" -o -name "*.h" | xargs cat | wc -l
3972
latesoon@latesoon-virtual-machine:~/ics2017/nemu$
```

图 26: make count

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?

共有 3283 行。

使用命令:

```
1 find . -name "*.c" -o -name "*.h" | xargs grep -v '^$' | wc -l
```

xargs 部分, -v 表示反向匹配, 这一操作的实际含义是删除匹配的行 (即空行), 其余不变。

```
latesoon@latesoon-virtual-machine:~/ics2017/nemu$ find . -name "*.c" -o -name "*.h" | xargs grep -v '^$' | wc -l
3283
```

图 27: 除去空行的统计

### (三) Question 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?

-Wall 的作用是生成所有警告信息。

-Werror 在发生警告时停止编译操作, 也就是说将所有警告当成错误进行处理。

使用的原因是提升代码的安全性, 方便在编写程序的过程中 debug。

## 七、所遇 BUG 及解决思路

### (一) 虚拟机无法复制文本

在配置环境阶段, 遇到了宿主机和虚拟机之间无法复制文本的问题。根据群里提供的解决方案, 通过先 Ctrl+Alt, 再 Ctrl+V 可以完成从宿主机到虚拟机的文本复制, 但无法完成从虚拟机到宿主机的复制。

最终, 我完成了共享文件夹的配置, 通过这种方式解决了不能文本复制的问题, 也为后续文件传输提前做好了准备。

我在 VMware 中完成了共享文件夹的配置, 并且在虚拟机中设置了开机自动挂载, 并为共享文件夹在主文件夹下创建了快捷方式, 以提高后续开发效率。

创建快捷方式的方法如下:

```
1 ln -s /mnt/hgfs/shared ~/shared
```

### (二) 修改 git 远程仓库时遇到的问题

考虑到虚拟机稳定性不足, 我在 GitHub 中创建了一个新仓库(在课程结课之前保持 Private), 并将 ics 项目的远程仓库进行修改。在修改过程中, 我遇到了由于 GitHub 更新, 强行要求我**绑定 SSH key** 并使用**仓库 SSH 地址**才能完成认证等问题, 通过一步一步查找解决方案解决了这一问题。

在虚拟机生成 SSH 密钥:

```
1 ssh-keygen -t ed25519 -C "your_email@example.com"
```

修改远程仓库:

```
1 git remote set-url origin git@github.com:username/repo-name.git
```