

# ICS PA1 Report

211870293 集成电路 李居奇 2023.10.10

## 实验进度

已完成 PA1 的全部内容，并通过所有 OJ 测试。表达式求值功能实现了对负数的处理，并设计表达式生成器完成了 10000 次的表达式测试。整个实验进度安排大致如下：

- \* 10.1-10.5 阅读并熟悉代码框架(国庆放假为主 doge)
- \* 10.6 完成简易调试器的基础功能(make\_token, si[], info r, x[])
- \* 10.7 完成表达式求值基础功能(不包含负数, 解引用, 寄存器打印)
- \* 10.8 完成表达式生成器以及部分表达式求值高级功能
- \* 10.9 继续完善表达式求值, 完成监视点和剩余内容

## 实验必答题 & 思考题

### Question 1:

#### 从状态机视角理解程序运行

以上一小节中 `1+2+...+100` 的指令序列为例, 尝试画出这个程序的状态机。

这个程序比较简单, 需要更新的状态只包括 PC 和 r1, r2 这两个寄存器, 因此我们用一个三元组 (PC, r1, r2) 就可以表示程序的所有状态, 而无需画出内存的具体状态。初始状态是 (0, x, x), 此处的 x 表示未初始化。程序 PC=0 处的指令是 `mov r1, 0`, 执行完之后 PC 会指向下一条指令, 因此下一个状态是 (1, 0, x)。如此类推, 我们可以画出执行前3条指令的状态转移过程:

```
(0, x, x) -> (1, 0, x) -> (2, 0, 0) -> (3, 0, 1)
```

请你尝试继续画出这个状态机, 其中程序中的循环只需要画出前两次循环和最后两次循环即可。

```
(1,0,x) -> (2,0,0) -> (3,0,1) -> (4,1,1) -> (2,1,1) -> (3,1,2)... -> (2,4950,99) -> (3,4950,100) -> (5,5050,100) -> (5,5050,100) ... (一直保持该状态)
```

### Question 2:

#### 为什么全部都是函数?

阅读 `init_monitor()` 函数的代码, 你会发现里面全部都是函数调用。按道理, 把相应的函数体在 `init_monitor()` 中展开也不影响代码的正确性。相比之下, 在这里使用函数有什么好处呢?

使用函数体封装不会改变程序执行的逻辑, 但可以使代码更容易阅读和维护, 并且如果这段代码在其他的函数体中也有使用, 使用函数而不是展开的代码可以减少整体代码量。从另一种角度, 这样写 `init_monitor()` 函数在内容上更加清爽, 可以更好地体现分层抽象。

### Question 3:

#### 🔗 kconfig生成的宏与条件编译

我们已经在上文提到过, kconfig会根据配置选项的结果在

`nemu/include/generated/autoconf.h` 中定义一些形如 `CONFIG_xxx` 的宏, 我们可以在C代码中通过条件编译的功能对这些宏进行测试, 来判断是否编译某些代码. 例如, 当 `CONFIG_DEVICE` 这个宏没有定义时, 设备相关的代码就无需进行编译.

为了编写更紧凑的代码, 我们在 `nemu/include/macro.h` 中定义了一些专门用来对宏进行测试的宏. 例如 `IFDEF(CONFIG_DEVICE, init_device());` 表示, 如果定义了 `CONFIG_DEVICE`, 才会调用 `init_device()` 函数; 而 `MUXDEF(CONFIG_TRACE, "ON", "OFF")` 则表示, 如果定义了 `CONFIG_TRACE`, 则预处理结果为 "ON" ("OFF" 在预处理后会消失), 否则预处理结果为 "OFF".

这些宏的功能非常神奇, 你知道这些宏是如何工作的吗?

现以分析宏 `MUXDEF` 为例, 其定义为:

```
#define MUXDEF(macro, X, Y) MUX_MACRO_PROPERTY(__P_DEF_, macro, X, Y)
```

可见其调用了另一个宏 `MUX_MACRO_PROPERTY`, 其相关定义为:

```
#define concat(x, y) x ## y
```

```
#define CHOOSE2nd(a, b, ...) b
```

```
#define MUX_WITH_COMMA(contain_comma, a, b) CHOOSE2nd(contain_comma, a, b)
```

```
#define MUX_MACRO_PROPERTY(p, macro, a, b) MUX_WITH_COMMA(concat(p, macro), a, b)
```

宏 `concat` 带有粘贴运算符 `###`, 将 `concat` 中的形参 `x` 和 `y` 简单的拼接在一起. 分析可得, `contain_comma` 是 `__P_DEF_` 和 `macro` 拼接后的结果, 如果拼接后的结果是一个宏, 比如 `__P_DEF_0`, 那么 `CHOOSE2nd(contain_comma, a, b)` 将会返回 `a`, 如果拼接后不是一个宏, 那么传给 `CHOOSE2nd` 的参数将只有两个, 那么将返回 `b`. 程序通过在 `macro.h` 中定义各种宏, 来实现上述类似开关的操作.

### Question 4:

#### 🕒 究竟要执行多久?

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

函数 `cpu_exec(n)` 中的参数 `n` 是一个 `uint64_t`, 代表的是 `cpu` 执行的次数, 按照负数的定义理解, 传入 `-1` 相当于传入 `0xffffffffffffffff`, 即让 `cpu` 执行  $2^{64}$  次指令. 这个数目相当大了, 一般情况就是指在程序结束(`nemu_trap`)前一直让 `cpu` 不断运行.

### Question 5:

#### 📄 优美地退出

为了测试大家是否已经理解框架代码, 我们给大家设置一个练习: 如果在运行NEMU之后直接键入 `q` 退出, 你会发现终端输出了一些错误信息. 请分析这个错误信息是什么原因造成的, 然后尝试在NEMU中修复它.

问题定位在 `cmd_q` 和 `is_exit_status_bad` 两个函数上，可知如果程序执行后状态一直为 `NEMU_END`，此时退出会使 `main` 函数 `return 1`，从而引发报错。修改如下，在 `cmd_q` 中改变状态为 `NEMU_QUIT` 即可。

```
static int cmd_q(char *args) {
// Change the flag of nemu_state
nemu_state.state = NEMU_QUIT;
return -1;
}

int is_exit_status_bad() {
int good = (nemu_state.state == NEMU_END && nemu_state.halt_ret == 0) ||
(nemu_state.state == NEMU_QUIT);
return !good;
}
```

#### Question 6:

##### 必答题

- 理解基础设施 我们通过一些简单的计算来体会简易调试器的作用。首先作以下假设：
  - 假设你需要编译500次NEMU才能完成PA。
  - 假设这500次编译当中，有90%的次数是用于调试。
  - 假设你没有实现简易调试器，只能通过GDB对运行在NEMU上的客户程序进行调试。在每一次调试中，由于GDB不能直接观测客户程序，你需要花费30秒的时间来从GDB中获取并分析一个信息。
  - 假设你需要获取并分析20个信息才能排除一个bug。

那么这个学期下来，你将会在调试上花费多少时间？

由于简易调试器可以直接观测客户程序，假设通过简易调试器只需要花费10秒的时间从中获取并分析相同的信息。那么这个学期下来，简易调试器可以帮助你节省多少调试的时间？

事实上，这些数字也许还是有点乐观，例如就算使用GDB来直接调试客户程序，这些数字假设你能通过10分钟的时间排除一个bug。如果实际上你需要在调试过程中获取并分析更多的信息，简易调试器这一基础设施能带来的好处就更大。

无简易调试器：75h

有调试器：25h

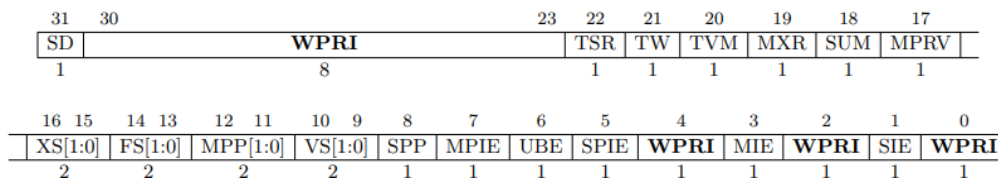
一学期总共节约 50 h

#### Question 7:

## 必答题

- **RTFM** 理解了科学查阅手册的方法之后, 请你尝试在你选择的ISA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:
  - x86
    - EFLAGS寄存器中的CF位是什么意思?
    - ModR/M字节是什么?
    - mov指令的具体格式是怎么样的?
  - mips32
    - mips32有哪几种指令格式?
    - CP0寄存器是什么?
    - 若除法指令的除数为0, 结果会怎样?
  - riscv32
    - riscv32有哪几种指令格式?
    - LUI指令的行为是什么?
    - mstatus寄存器的结构是怎么样的?

1. riscv32 有六种基本的指令格式, 分别是寄存器-寄存器操作的 R 型, 用于短立即数访存的 I 型指令, 用于访存操作的 S 型指令, 用于条件跳转操作的 B 型指令, 用于长立即数的 U 型指令, 和用于无条件跳转的 J 型指令。
2. ui 指令的全称是 Load Upper Immediate, 它的功能是把一个 20 位的立即数加载到寄存器的高 20 位, 执行完下面的指令后, 寄存器 x 的值为 0x12345000.  
lui x3, 0x12345
3. 32 位寄存器如下图所示:



Mstatus 寄存器即 Machine Status Registers, 根据 RV 手册内容其含有 Global interrupt-enable bits, MIE and SIE, 以及 SXL and UXL fields which are WARL fields that control the value of XLEN for S-mode and U-mode, MPRV (Modify PRiVilege) bit modifies the effective privilege mode 等. 可见该寄存器的结构主要是一些全局终端, 特权等相关的标志位.

## Question 8:

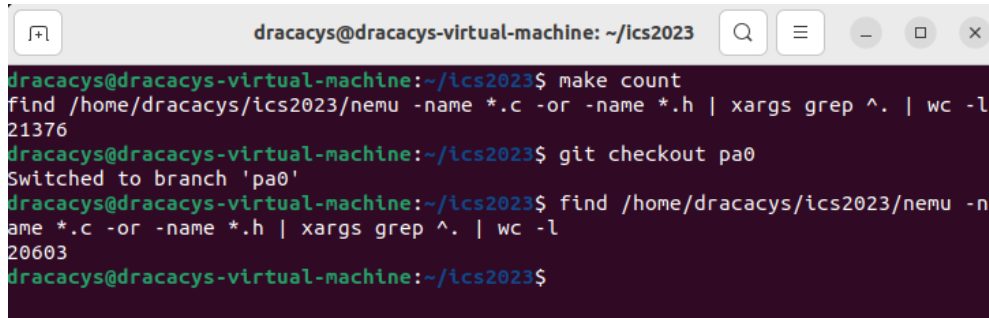
## 必答题

- **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 `pa0` 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?

count:

```
find $(NEMU_HOME) -name *.c -or -name *.h | xargs grep ^. | wc -l
```

通过 `make count` 即可打印在 `nemu` 目录下 `c` 文件和 `h` 文件包含的行数,且该行数不包括空格。



```
dracacys@dracacys-virtual-machine: ~/ics2023
dracacys@dracacys-virtual-machine:~/ics2023$ make count
find /home/dracacys/ics2023/nemu -name *.c -or -name *.h | xargs grep ^. | wc -l
21376
dracacys@dracacys-virtual-machine:~/ics2023$ git checkout pa0
Switched to branch 'pa0'
dracacys@dracacys-virtual-machine:~/ics2023$ find /home/dracacys/ics2023/nemu -n
ame *.c -or -name *.h | xargs grep ^. | wc -l
20603
dracacys@dracacys-virtual-machine:~/ics2023$
```

### Question 9:

#### 必答题

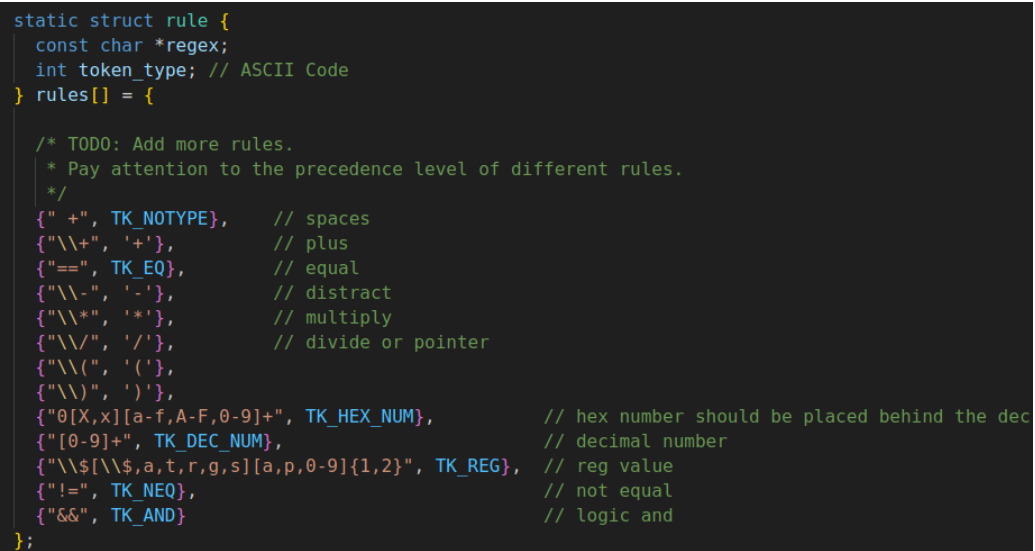
- **RTFM** 打开 `nemu/scripters/build.mk` 文件, 你会在 `CFLAGS` 变量中看到gcc的一些编译选项. 请解释gcc中的 `-Wall` 和 `-Werror` 有什么作用? 为什么要使用 `-Wall` 和 `-Werror` ?

`-Wall` 的作用是生成所有警告信息, `-Werror` 是将所有警告信息转换为编译错误, 使用这两个严格的编译选项可以确保代码的质量和可靠性, 以及强制执行一些编码规范. 这在大型项目中尤其有用, 可以防止潜在的问题进入到最终的可执行文件中。

## 实验的关键步骤

### Make Token:

(1) 正则表达式的配置



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

    /* TODO: Add more rules.
     * Pay attention to the precedence level of different rules.
     */
    {" +", TK_NOTYPE}, // spaces
    {"\\+", '+'}, // plus
    {"==", TK_EQ}, // equal
    {"\\-", '-'}, // distract
    {"\\*", '*'}, // multiply
    {"\\/", '/'}, // divide or pointer
    {"\\(", '('},
    {"\\)", ')'},
    {"0[X,x][a-f,A-F,0-9]+", TK_HEX_NUM}, // hex number should be placed behind the dec
    {"[0-9]+", TK_DEC_NUM}, // decimal number
    {"\\$[\\$,a,t,r,g,s]{a,p,0-9}{1,2}", TK_REG}, // reg value
    {"!=", TK_NEQ}, // not equal
    {"&&", TK_AND} // logic and
};
```

(2) 函数实现的基本逻辑: 第一类为运算符(包含负号, 解指针, 寄存器), 第二类为数值, 对于第一类, 仅需记录其类型, 对于第二类需要完整的将其内容 `copy` 到 `str` 中, 方便后续的 `parsing`。

```

case TK_AND: {
    if(nr_token == 4096) {
        printf("Token exceed.\n");
        return false;
    }
    tokens[nr_token].type = TK_AND;
    strcpy(tokens[nr_token].str, "&&");
    *(tokens[nr_token].str + substr_len) = '\0';
    nr_token++;
    break;
}

case TK_HEX_NUM: {
    if (nr_token == 4096)
    {
        printf("Token exceed.");
        return false;
    }
    tokens[nr_token].type = TK_HEX_NUM;
    if(substr_len < 4096) {
        strncpy(tokens[nr_token].str, e + position - substr_len, substr_len);
        *(tokens[nr_token].str + substr_len) = '\0';
    }
    else {
        printf("Token str exceed.");
        return false;
    }
    nr_token++;
    break;
}
}

```

调试器的基本功能:

### 1. 单步执行

```

// TASK1: The function that can step through the program
static int cmd_si(char *args) {

    uint64_t exe_times = 1;
    if (args == NULL) {
        /* no argument given */
        printf("Step through the program for %lu steps.\n", exe_times);
        cpu_exec(exe_times); // Excute once
    }
    else {
        sscanf(args, "%lu", &exe_times);
        printf("Step through the program for %lu steps.\n", exe_times);
        cpu_exec(exe_times);
    }
    return 0;
}

```

### 2. 打印寄存器和监视点

```

// TASK2: Print the information of reg or watching point
static int cmd_info(char *args) {
    if (args == NULL) {
        /* no argument given */
        printf("Error: The info needs 1 args!\n");
        return 0;
    }
    else if (strcmp(args, (const char*)"r") == 0){
        printf("The rigister value in nemu:\n");
        isa_reg_display();
    }
    else if (strcmp(args, (const char*)"w") == 0){
        // Print the value of watching point
        print_wp();
    }
    return 0;
}

```

### 3. 内存扫描

```
// TASK3: Print the information of memory
static int cmd_x(char *args) {
    if (args == NULL) {
        /* no argument given */
        printf("Error: The x needs 2 args!\n");
        return 0;
    }
    else {
        char *arg1 = strtok(args, " ");
        char *arg2 = strtok(NULL, " ");
        if (arg2 == NULL) {
            printf("Error: The x needs 2 args!\n");
            return 0;
        }
        // Parsing the value
        uint32_t index, addr;
        bool success;
        sscanf(arg1, "%u", &index);
        addr = expr(arg2, &success);
        if(success == false) assert(0);
        puts("The information of memory is listed below:\n");
        for(uint16_t i = 0; i < index; i++) {
            printf("Address: 0x%08x   Value: 0x%02x\n", addr + i, vaddr_read(addr + i, 1));
        }
    }
    return 0;
}
```

#### 表达式求值:

(1) 表达式双目计算: 即只考虑常规的加减乘除, 寄存器, 十六进制

该过程和思路十分简单, 我基本参照实验手册所提供的框架, 即采用递归的方法, 利用主运算符不断将 token 分组计算, 分到每组只剩一个 token 时, 便可以直接 return value 即可。可以看出, 我的代码基本与所给框架一致。

```
eval(p, q) {
    if (p > q) {
        /* Bad expression */
    }
    else if (p == q) {
        /* Single token.
        * For now this token should be a number.
        * Return the value of the number.
        */
    }
    else if (check_parentheses(p, q) == true) {
        /* The expression is surrounded by a matched pair of parentheses.
        * If that is the case, just throw away the parentheses.
        */
        return eval(p + 1, q - 1);
    }
    else {
        /* We should do more things here. */
    }
}
```

#### My function:

1. word\_t eval(int p, int q) {
- 2.

```

3.  if (p > q) {
4.      /* Bad expression */
5.      printf("Error: eval() occurs bad expression.\n");
6.      assert(0);
7.  }
8.
9.  else if (p == q) {
10.     /* Single token.
11.      * For now this token might be a decimal/hex/reg
12.      * Return the value of the number.
13.      */
14.     word_t result;
15.
16.     switch (tokens[p].type) {
17.         case TK_DEC_NUM: {
18.             sscanf(tokens[p].str, "%u", &result);
19.             return result;
20.         }
21.         case TK_HEX_NUM: {
22.             sscanf(tokens[p].str, "%x", &result);
23.             return result;
24.         }
25.         case TK_REG: {
26.             bool success;
27.             char reg_name [5] = {0};
28.             sscanf((tokens[p].str + 1), "%s", reg_name);
29.             result = isa_reg_str2val(reg_name, &success);
30.             if (success == 1)
31.                 return result;
32.             else {
33.                 printf("Reg value fault");
34.                 assert(0);
35.             }
36.         }
37.         default: assert(0);
38.     }
39. }
40.
41. else if (check_parentheses(p, q) == true) {
42.     /* The expression is surrounded by a matched pair of parentheses.
43.      * If that is the case, just throw away the parentheses.
44.      */
45.     return eval(p + 1, q - 1);
46. }

```



```

47.
48.     else {
49.         int op_index = find_main_op(p, q);
50.         if (op_index > 0) {
51.             switch (tokens[op_index].type) {
52.                 case '+': return (eval(p, op_index - 1) + eval(op_index + 1, q));
53.                 case '-': return (eval(p, op_index - 1) - eval(op_index + 1, q));
54.                 case '*': return (eval(p, op_index - 1) * eval(op_index + 1, q));
55.                 case '/': return (eval(p, op_index - 1) / eval(op_index + 1, q));
56.                 case TK_EQ: return (eval(p, op_index - 1) == eval(op_index + 1, q));
57.                 case TK_NEQ: return (eval(p, op_index - 1) != eval(op_index + 1, q));
58.                 case TK_AND: return (eval(p, op_index - 1) && eval(op_index + 1, q));
59.                 default: assert(0);
60.             }
61.         }
62.
63.         else if(tokens[p].type == TK_NEG) {
64.             /* For now this token is a negtive
65.              * Return the value of the number.
66.              * 正常情况下，一个负号的后面只可能为数字，括号，负号，ptr*
67.              */
68.             int i;
69.             int num = 1;
70.             // Continuing '-'
71.             for(i = p + 1; i < nr_token; i++) {
72.                 if(tokens[i].type != TK_NEG)
73.                     break;
74.                 num++;
75.             }
76.             if (tokens[i].type != '(') {
77.                 if (num%2 != 0) return ~(eval(i, i)) + 1);
78.                 else return eval(i, i);
79.             }
80.             // still have '('
81.             else {
82.                 for(int j = i + 1; j < nr_token; j++) {
83.                     if (check_parentheses(i, j) == true) {
84.                         if (num%2 != 0) return ~(eval(i + 1, j - 1)) + 1);
85.                         else return eval(i + 1, j - 1);
86.                     }
87.                 }
88.                 printf("( is not balance! Error 1!\n"); assert(0);
89.             }
90.         }

```

```

91.
92.     else if(tokens[p].type == TK_PTR) {
93.         /* For now this token is a ptr number
94.          * Return the value of the number.
95.          * 正常情况下，一个负号的后面只可能为数字，括号，负号
96.          */
97.         int i;
98.         int num = 1;
99.         // Continuing '*'
100.        for(i = p + 1; i < nr_token; i++) {
101.            if(tokens[i].type != TK_PTR)
102.                break;
103.            num++;
104.        }
105.        if (tokens[i].type != '(') {
106.            return ptr_dereference(eval(i, i), num);
107.        }
108.        // still have '('
109.        else {
110.            for(int j = i + 1; j < nr_token; j++) {
111.                if (check_parentheses(i, j) == true) {
112.                    return ptr_dereference(eval(i + 1, j - 1), num);
113.                }
114.            }
115.            printf("( is not balance! Error 2!\n"); assert(0);
116.        }
117.    }
118.    else {printf("Unkown type of token. Critical error!\n"); assert(0);}
119. }
120. }

```

(2) 表达式单目双目计算：即除了考虑双目运算外，需要考虑负号和取指针运算。首先，需要试图在加号与星号中识别出其本意，根据实验手册的提示，也不难想到，负号只可能出现在第一个位置，或其前面不是数字和右括号，解指针同理，于是可以在 `expr` 中编写代码修改其类型。而负数的计算在 `eval` 函数中已经呈现，这里不再赘述。

```

1.  /* TODO: Insert codes to evaluate the expression. */
2.  /*
3.     负号可能出现在第一个位置，或其前面不是数字和右括号
4.     指针解引用可能出现在第一个位置，或其前面不是数字和右括号
5.  */
6.  for (int i = 0; i < nr_token; i++) {
7.      if (tokens[i].type == '-' &&
8.          (i == 0 || (tokens[i - 1].type != ')') &&
9.          tokens[i - 1].type != TK_DEC_NUM &&

```

```

10. tokens[i - 1].type != TK_REG &&
11. tokens[i - 1].type != TK_HEX_NUM))) {
12.     tokens[i].type = TK_NEG;
13. }
14.
15. if (tokens[i].type == '*' &&
16. (i == 0 || (tokens[i - 1].type != ')') &&
17. tokens[i - 1].type != TK_DEC_NUM &&
18. tokens[i - 1].type != TK_REG &&
19. tokens[i - 1].type != TK_HEX_NUM))) {
20.     tokens[i].type = TK_PTR;
21. }
22. }

```

(3)表达式的测试：基本仿造框架代码，但为了保证在除号后的表达式不为0，且所有运算均为无符号数，这里做了一些处理。对于第一个问题，采用直接在除号后的表达式\*2+1的操作，对于第二个问题，可以将输入GCC计算答案的字符串在每个数值后面加上u，这样就能保证全部按照无符号数处理。

```

1. static void gen_rand_expr() {
2.     // To avoid the segmental fault
3.     switch (buf_index < 20 ? choose(6) : 0) {
4.         case 0: {gen_space(); gen_num(buf_index); gen_space(); break;}
5.         case 1: {gen('('); gen_rand_expr(); gen(')'); break;}
6.         case 2: { // To avoid /0
7.             gen_rand_expr(); gen_space(); gen('/'); gen_space();
8.             gen('('); gen('('); gen_rand_expr(); gen(')');
9.             gen('*'); gen_num_c(2); gen('+');
10.            gen_num_c(1); gen(')'); break;
11.        }
12.        default: {gen_rand_expr(); gen_space(); gen_rand_op(); gen_space(); gen_rand_expr(); break;}
13.    }
14.    return ;

```

宏定义如下所示：

```

1. #define choose(x) (rand() % x) // generate random num
2. #define gen(c)    n_input = sprintf(buf + buf_index,"%c",c); n_input_c = sprintf
    (buf_c + buf_index_c,"%c",c); buf_index = (buf_index >= 65530 ? 65530 : buf_inde
    x + n_input); buf_index_c = (buf_index_c >= 65530 ? 65530 : buf_index_c + n_inpu
    t_c); // print one char in the buf
3. #define gen_num    ran = rand() % 10; n_input = sprintf(buf + buf_index,"%u",ran)
    ; n_input_c = sprintf(buf_c + buf_index_c,"%uu",ran); buf_index = (buf_index >=
    65530 ? 65530 : buf_index + n_input); buf_index_c = (buf_index_c >= 65530 ? 6553
    0 : buf_index_c + n_input_c); // print one num in the buf

```

```

4. #define gen_num_c(r)    n_input = sprintf(buf + buf_index,"%u",r); n_input_c = sp
    rintf(buf_c + buf_index_c,"%uu",r); buf_index = (buf_index >= 65530 ? 65530 : bu
    f_index + n_input); buf_index_c = (buf_index_c >= 65530 ? 65530 : buf_index_c +
    n_input_c);

```

最后，我将测试功能移植到 nemu 的主函数，输入 e 则进行测试，10000 次测试结果如下：

```

th len 1:
[src/monitor/sdb/expr.c:104 make_token] match rules[9] = "[0-9]+" at position 22
0 with len 1: 4
[src/monitor/sdb/expr.c:104 make_token] match rules[0] = " +" at position 221 wi
th len 2:
[src/monitor/sdb/expr.c:104 make_token] match rules[3] = "\-" at position 223 wi
th len 1: -
[src/monitor/sdb/expr.c:104 make_token] match rules[0] = " +" at position 224 wi
th len 4:
[src/monitor/sdb/expr.c:104 make_token] match rules[9] = "[0-9]+" at position 22
8 with len 1: 0
[src/monitor/sdb/expr.c:104 make_token] match rules[0] = " +" at position 229 wi
th len 3:
[src/monitor/sdb/expr.c:104 make_token] match rules[1] = "\+" at position 232 wi
th len 1: +
[src/monitor/sdb/expr.c:104 make_token] match rules[0] = " +" at position 233 wi
th len 3:
[src/monitor/sdb/expr.c:104 make_token] match rules[9] = "[0-9]+" at position 23
6 with len 1: 9
[src/monitor/sdb/expr.c:104 make_token] match rules[0] = " +" at position 237 wi
th len 3:
Line: 9999    Result: 533    Answer: 533
Test pass.
(nemu)

```

### 监视点：

涉及基本的链表操作，没啥好说的，表达式求值只要正确这里不会有什么问题和难度，写好基本的插入和删除后将 check 放入 cpu 的执行函数即可。

```

1. WP* new_wp() {
2.     if (free_ == NULL) {
3.         printf("No free watchpoint.");
4.         assert(0);
5.     }
6.     else {
7.         WP *temp = free_;
8.         free_ = free_->next;
9.         temp->next = head;
10.        head = temp;
11.    }
12.    return head;
13. }
14.
15. /* move WP from head to free_

```

```

16. */
17. void free_wp(WP *wp) {
18.     if(head == NULL) {
19.         printf("Not find the watching point.\n");
20.         return;
21.     }
22.     if(wp == head) {
23.         head = head -> next;
24.         wp->next = free_;
25.         free_ = wp;
26.     }
27.     else {
28.         WP *temp = head;
29.         while(temp->next != wp) {
30.             if(temp->next == NULL) {
31.                 printf("Not find the watching point.\n");
32.                 return;
33.             }
34.             temp = temp->next;
35.         }
36.         temp->next = wp->next;
37.         wp->next = free_;
38.         free_ = wp;
39.     }
40. }
41.
42. /* print the value of all active watching point
43. */
44. void print_wp(void) {
45.     WP *temp = head;
46.     while(temp != NULL) {
47.         printf("Watching point %d: expr: %s, latest value: 0x%08x\n", temp->NO, temp
            ->expr, temp->result);
48.         temp = temp->next;
49.     }
50. }
51.
52. /* delete the certain watching point
53. */
54. void delete_wp(unsigned int index) {
55.     free_wp(wp_pool + index);
56.     return ;
57. }
58.

```

```

59. /* delete all watching point
60. * if the value one active point change then return 1
61. * else return 0
62. */
63. bool check_wp(void) {
64.     WP *temp = head;
65.     uint32_t new_value;
66.     bool flag = false;
67.     while(temp != NULL) {
68.         bool success;
69.         new_value = expr(temp->expr, &success);
70.         if (temp->result != new_value) {
71.             flag = true;
72.             printf("Watching point %d value change:  expr: %s  value: 0x%08x  ---
-> 0x%08x\n", temp->NO, temp->expr, temp->result, new_value);
73.             temp->result = new_value;
74.         }
75.         temp = temp->next;
76.     }
77.     return flag;
78. }

```

## 实验心得（踩坑记录）

1. 花时间最多的 bug: 本人不知道 `strncpy` 函数在写字符串时不会自带 `\0`，由于字符数组较长，初始化后有较多 `\0`，故没有出现 `segmental fault`，但答案出现异常，最后 `gdb` 单步调试了接近一个小时才解决问题。
2. 未解之谜: 第一次 OJ 在测试监视点时触发了我的 `assert(0)`，该点触发说明对 OJ 提供的监视点表达式在 `make_token` 中失败。之后检查了 `make_token` 函数应该不存在问题，于是我删掉了 `assert(0)`，结果 OJ 就通过了。现在不太清楚到底是 OJ 的测试监视点表达式存在非法字符，还是我的函数存在比较隐藏的 bug。

Compile OK

### Easy Tests

- Passed Simple exits by q.
- Assert Fail (Assert fail: 0 @ src/monitor/sdb/sdb.c:206) Simple set a watchpoint.
- Passed p for printing a constant or a simple expression.
- Passed p for printing a simple expression with a negative value.
- Passed Simple exits after si.
- Passed Simple exits after info r.
- Passed Simple exits after info w.
- Passed Simple exits after info si n.
- Passed Simple print for a hex number.
- Passed Simple print for a register.

### Hard Tests

- Passed p for evaluating expressions (\*,-,\*,0).
- Assert Fail (Assert fail: 0 @ src/monitor/sdb/sdb.c:206) set and delete watchpoints
- Passed p for evaluating expressions (\*,-,\*,0) and registers.
- Passed p for evaluating expressions with unary minus (e.g., -1) and registers.