PA1（下）- 表达式/监视点/i386

17307130178 宁晨然

一、概述

本次实验重心在于表达式求值、监视点和 i386 手册的功能实现。表达式求值是一个完整的流程，分为识别表达式和求值两个过程。

表达式求值实现了+-*/十进制十六进制、带括号、<< >> <= < == != > >= && || ! 负数、指针。（蓝框、选做均实现）

监视点实现了①初始化监视点②添加监视点 w expr③删除监视点 d n④显示监视点 info w⑤检查监视点 cpu_exec。

并且需要阅读 i386 手册回答必答题。

二、实验过程

（一）、表达式求值

整个表达式求值的过程主要分为词法分析、递归求值两个过程。词法分析需要定义识别 tokens 的规则 rules，然后使用规则识别出每个数值、运算符 make_token()，识别出来后存放在 tokens 中。递归求值中，需要实现括号识别 check_parentheses()/优先级最低运算符寻找 dominant_op(p,q)/递归求值 eval(p,q)。其中子函数有符号优先级查找 oprank(op)。

表达式求值基本实现内容在 nemu/src/monitor/debug/expr.c 中。

如果要按照顺序实现每个功能，只有当完整的功能实现完成后才能开始测试是否有 bug，所以按照 kiss 原则，我先实现+-*/，再考虑负数，最后填充后面所有的。

1.词法分析

(1):分析结构

①enum / rule

观察原先给出的 enum 和 rule 之后可以发现，此处 enum 是对应 rule.token_type 的类 ascii 值，实际上赋值任何非 ascii 范围内的值均可，所以后面的运算符只需从 256 降序顺序排列即可。Enum 中的值与 token_type 对应。按需求放入运算符。

```c
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 do nothing

        {"0x[0-9a-fA-F]{1,8}", TK_NUM_16},
        {"[0-9]{1,10}", TK_NUM_10},

        {"\\$e(ax|bx|cx|dx|sp|bp|si|di|ip)", TK_$},
        {"\\(", TK_LBRA},
        {"\\)", TK_RBRA},

        {"\\+", '+'}, // plus
        {"\\-", '-'}, // minus & neg
        {"\\*", '*'}, // multiply & pointer
        {"\\/", '/'},

        {"==", TK_EQ},   // equal
        {"!=", TK_NOT_EQ}, // not equal

        {"&&", TK_AND},
        {"\\|\\|", TK_OR},

        {"<<", TK_LSHIFT},
        {"<=", TK_SM_EQ},
        {"<", TK_SM},
        {">>", TK_RSHIFT},
        {">=", TK_BG_EQ},
        {">", TK_BG},

        {"!", TK_NOT},

};
```

Rule 的 struct 中，所有的 regex 字符串都是正则表达式匹配模式，其中+$()-*/|为防止歧义加上了转义字符\\。

①解析原有：” +”匹配一个或多个空格；加法’+’，直接对应 ascii 中’+’

②添加+-*/：对于数值而言，需要写十进制和十六进制，加减乘除按照 ascii 即可。

③此处对于多义：*指针和-负数，不在此处定义 rule，而是在词法分析中再次判断。

```c
enum
{
        TK_NOTYPE = 256,
        TK_NUM_10 = 255,
        TK_NUM_16 = 254,
        TK_$ = 253,
        TK_LBRA = 252,
        TK_RBRA = 251,
        TK_EQ = 250,
        TK_NOT_EQ = 249,
        TK_AND = 248,
        TK_OR = 247,
        TK_NEG = 246,
        TK_POINT = 245,
        TK_NOT = 244,
        TK_LSHIFT = 243,
        TK_RSHIFT = 242,
        TK_BG = 241,
        TK_BG_EQ = 240,
        TK_SM = 239,
        TK_SM_EQ = 238

        /* TODO: Add more token types */
};
```

④enum 顺序降序排列即可，对于没有 ascii 定义的运算符，使用大于 128 即可。

②make_token

这一步是实现将字符串根据 rule 规则识别成运算数组，存在 tokens 中。将 rule 遍历一遍，查找到能够 match 的规则即继续执行，所以在 rule 中的顺序很关键。

e.g. << < <=排列顺序一定是<< <= <否则会先识别成<和=，发生错误。规律就是先把长段识别，再识别小段。

识别完成后，需要把字符串复制进 tokens 中，所以使用 strncpy 函数，并且将最后一位填充为'\0'。这里注意到每次都是用的全局变量 tokens 数组，所以每次运算前需要把 tokens 数组手动清零。

```
Log("match rules[%d] = \"%s\" at position %d with len %d: %.*s", i,
r_len, substr_len, substr_start);
position += substr_len;
strncpy(tokens[nr_token].str, substr_start, substr_len);
tokens[nr_token].str[substr_len] = '\0';
```

根据识别的字符，填充至 tokens 的 token_type，nr_token 保存现在运算串的个数。其中大部分规则都是

```
tokens[nr_token].type = TOKEN;
nr_token++;break;
```

需要特殊处理的是* -的多重意思：

```
case '-':
    if (nr_token == 0 || (tokens[nr_token - 1].type != TK_NUM_10 && tokens[nr_token -
1].type != TK_NUM_16) || tokens[nr_token - 1].type == TK_LBRA)
        tokens[nr_token].type = TK_NEG;
    else
        tokens[nr_token].type = '-';
    nr_token++;
    break;
case '*':
    if (nr_token == 0 || (tokens[nr_token - 1].type != TK_NUM_10 && tokens[nr_token -
1].type != TK_NUM_16) || tokens[nr_token - 1].type == TK_LBRA)
        tokens[nr_token].type = TK_POINT;
    else
        tokens[nr_token].type = '*';
    nr_token++;
    break;
```

-中如果是表示负数，前面的 token 不是数字，若为空和(则必为负号；*同理。

(2):KISS 原则

从易到难，逐步推进。这个原则在本次实验体现的淋漓尽致，由于需要实现多个运算符，我先拆分了三步，实现+-*/十进制带括号数的操作，再添加单目运算符*-，再添加剩下的双目运算符，更新优先级。这样可以更好地 debug。

本身 KISS 原则应该是每一个单元进行测试，所以在做括号测试的时候，我也是单独写了 c 语言测试程序，查看是否可以识别正确的括号。在做优先级时，从易到难，先做加减乘除再添加单目优先级，会方便很多。

为 tokens 添加 rule 后，并且成功识别为 tokens 数组，所有信息都存放至 tokens 中，此时词法分析的步骤全部完成，后面发现前后没有直接联系。

2.递归求值

(1):check_parentheses()

```
//寻找括号匹配
bool check_parentheses(int p, int q)
{
    if (tokens[p].type == TK_LBRA && tokens[q].type == TK_RBRA) //外层需要括号
    {
        int left = 0, right = 0, i;
        for (i = p + 1; i < q && left >= right; i++)
        {
            if (tokens[i].type == TK_LBRA)
                left++;
            else if (tokens[i].type == TK_RBRA)
                right++;
        }
        if (left != right)
            return false;
    }
    else
        return false; //需要被括号包括
    return true;
}
```

括号匹配需要保证三件事：

①首为(，尾为)

②首尾括号匹配

③中间括号不能出错

左括号 level 升级，右括号 level 降级，保证 level>=0 且最后=0 即可。

返回值为布尔值。

```c
int oprank(int ch)
{
    switch (ch)
    {
    case TK_NEG:
    case TK_POINT:
    case TK_NOT:
        return 2;
    case '/':
    case '*':
        return 3;
    case '+':
    case '-':
        return 4;
    case TK_LSHIFT:
    case TK_RSHIFT:
        return 5;
    case TK_BG:
    case TK_BG_EQ:
    case TK_SM:
    case TK_SM_EQ:
        return 6;
    case TK_EQ:
    case TK_NOT_EQ:
        return 7;
    case TK_AND:
        return 11;
    case TK_OR:
        return 12;
    default:
        return 0;
    }
}
```

(2):oprank(op)

由于需要找到一个表达式中最后执行的运算符，需要考虑优先级和结合性。所以需要写一个判断 op 的优先级的函数，即 oprank(op)左图。

其中单目运算符均为从左到右结合，双目运算符从右到左结合

(3):dominant_op(p,q)

用于判断表达式中，最后执行的运算符。需要区分数值和运算符，首先对于 level>0 即括号内部的表达式均可忽略，最后执行的表达式一定是 level=0。其次保存最后执行的 op 位置和其优先级，如果 rank 更高，则最晚执行。此处

```c
int dominant_op(int p, int q)
{
    int level = 0, op = -1, i;
    int rank, lowest_rank = 0;
    for (i = p; i <= q && level >= 0; i++)
    {
        if (tokens[i].type == TK_LBRA)
        {
            level++;
            continue;
        }
        else if (tokens[i].type == TK_RBRA)
        {
            level--;
            continue;
        }
        if (level == 0)
        {
            rank = oprank(tokens[i].type);
            if (rank >= lowest_rank)
            {
                if (rank == 2 && lowest_rank == 2)
                    ; //proirity from left
                else
                { //update proirity from right
                    op = i;
                    lowest_rank = rank;
                }
            }
        }
    }
    if (op == -1)
        assert(0);
    return op;
}
```

需要单独考虑 rank==2 和 lowest_rank==2 时，说明此时有两个单目运算符，结合性从左向右，所以不需要更新。对于双目预算符需要更新为最新的双目运算符。

(4):expr(e,success)

```c
uint32_t expr(char *e, bool *success)
{
    clear();
    if (!make_token(e))
    {
        *success = false;
        return 0;
    }

    /* TODO: Insert codes to evaluate the expression. */

    int result;
    result = eval(0, nr_token - 1);

    return result;
}
```

此是完整的表达式流程，先调用 clear()清空数组，调用 make_token 分解字符串存放进 tokens，最后调用 eval(0,nr_token - 1)递归求值。

```c
static int cmd_p(char *args)
{
        bool *success = false;
        printf("%u\n",expr(args, success));
        return 0;
}
```

所以 cmd_p 中就是调用这个函数，更新一下。

(5):eval(p,q)

```c
int eval(int p, int q)
{
    /* Bad expression */
    if (p > q)
    {
        printf("p = %d, q = %d\n", p, q);
        printf("Bad expression!\n");
        assert(0);
    }
    else if (p == q) /* Single token.*/
    {
        if (tokens[p].type == TK_NUM_10)
        {
            int result;
            printf("tokens.str = %s\n", tokens[p].str);
            sscanf(tokens[p].str, "%d", &result);
            return result;
        }
        else if (tokens[p].type == TK_NUM_16)
        {
            int result;
            sscanf(tokens[p].str, "%x", &result);
            return result;
        }
```

```c
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
{
    printf("p = %d, q = %d\n", p, q);
    int op, val1, val2, op_type;
    op = dominant_op(p, q);
    op_type = tokens[op].type;
    printf("op_type = %d\n", op_type);

    switch (op_type) //priority = 2
    {
    case TK_NEG:
        return -eval(p + 1, q);
    case TK_POINT:
        return swaddr_read(eval(p + 1, q), 4);
    case TK_NOT:
        return !eval(p + 1, q);
    default:
        break;
    }
```

eval 递归求值分为几步:

①p>q:表达式出错

② p=q: 返回当前值，可能为十六进制、十进制、寄存器。寄存器没有想到很好的办法输出，直接暴力枚举了 9 个常用输出寄存器，common.h 中 reg_l()函数，取出寄存器值。

```c
else if (tokens[p].type == TK_$)
{
    if (strcmp(tokens[p].str, "$eax") == 0)
        return reg_l(0);
    else if (strcmp(tokens[p].str, "$ecx") == 0)
        return reg_l(1);
    else if (strcmp(tokens[p].str, "$edx") == 0)
        return reg_l(2);
    else if (strcmp(tokens[p].str, "$ebx") == 0)
        return reg_l(3);
    else if (strcmp(tokens[p].str, "$esp") == 0)
        return reg_l(4);
    else if (strcmp(tokens[p].str, "$ebp") == 0)
        return reg_l(5);
    else if (strcmp(tokens[p].str, "$esi") == 0)
        return reg_l(6);
    else if (strcmp(tokens[p].str, "$edi") == 0)
        return reg_l(7);
    else if (strcmp(tokens[p].str, "$eip") == 0)
        return cpu.eip;
    else
    {
        printf("No such register!\n");
        assert(0);
    }
}
```

③p<q / 有匹配括号：删除括号继续求值

④p<q / 无匹配括号：

先寻找到最后执行的 op，判断是否为单目运算符，需要 switch 一下，分别为负数、指针、取反运算符。指针引用 swaddr_read 函数对内存进行调用。

如果为双目运算符，先计算双目运算符左边 val1 和右边 val2，根据 switch(op_type)，进行运算操作即可。

```
switch (op_type)
{
case '+':
    return val1 + val2;
case '-':
    return val1 - val2;
case '*':
    return val1 * val2;
case '/':
    return val1 / val2;
case TK_LSHIFT:
    return val1 << val2;
case TK_RSHIFT:
    return val1 >> val2;
case TK_BG:
    return val1 > val2;
case TK_BG_EQ:
    return val1 >= val2;
case TK_SM:
    return val1 < val2;
case TK_SM_EQ:
    return val1 <= val2;
case TK_EQ:
    return val1 == val2;
case TK_NOT_EQ:
    return val1 != val2;
case TK_AND:
    return val1 && val2;
case TK_OR:
    return val1 || val2;
default:
    printf("something wrong!\n");
    assert(0);
}
```

这里实现了常用的几个双目操作符:

+ - * /

<< >>

< <= > >= == !=

&& ||

之后需要再添加。(选做操作符已做)

3.调试过程

由于本实验需要实现多个子函数，所以最好进行单元测试。

(1):括号匹配 单元测试较简单，直接 c 语言加一个头和输出即可。

(2):实现加减乘除 保证括号匹配完成后，优先级我只写了+- */分开，没有考虑其他单目运算符。这个时候的表达式很单纯，例如 10+1，这个就让我调试了好久。总结一下问题:

①switch case 需要添加 default，如果不加没有办法运行，可能会没有返回值。

②dominant_op 需要最后添加 return true 每次都记得 return false 却不忘了 true

③expr 调用前一定要先清空 tokens 数组，否则无法更新表达式计算

④rule 中正则表达式需要添加转义符号，并且是两个\\

⑤优先级不需要判断非操作符，即立即数和寄存器值

⑥rule 的顺序极其重要，否则会匹配失败。

(3):实现负号和指针（蓝框选做）

讲义中的方法是，所有的 tokens 已经识别结束后，再遍历一遍查看哪些是单目运算符。感觉这种方法有点耗费时间，其实在识别 tokens 的时候已经可以得知前方运算符还是立即数，没有必要单独再遍历一遍，在 switchcase 中即可判断。负号判断方法已经在前面讲过了。

(4):实现优先级 操作符多了之后，优先级和结合性不能错乱。所以我专门查了一下表格，

C语言优先级

优先级	运算符	名称或含义	使用形式
1	[]	数组下标	数组名[整型表达式]
	()	圆括号	(表达式)/函数名(形参表
	.	成员选择（对象）	对象.成员名
	->	成员选择（指针）	对象指针->成员名
	-	负号运算符	-算术类型表达式

类似左边的表格的东西，按照其中的表格优先级，即可很轻松的写出优先级顺序。注意优先级越高的运算符越晚执行，也是我们需要找到的最晚执行的运算符。本次实验中的单目运算符都是从左到右结合，则取最左为最后执行。双目则取最右为最后执行。

(5):register 寄存器的实现我才用暴力枚举的方法实现的，感觉确实有点问题，也不太清楚正规的方法应该怎么匹配。还有读取内存的时候报错 swaddr_t 是个 label 不是类型，但是明明包含了定义的头文件的……很迷惑

(6):git log 每次都需要更新 git 记录

```
commit 7eadbbb2ba91a487a68f8059cae7de3a52b59ac1 (HEAD -> master)
Author: 17307130178-Ning Chenran <17307130178@fudan.edu.cn>
Date:   Thu May 2 23:34:22 2019 -0700

    first try on expression. I add "+-*/" and "()" expressions

    and I try some exms,it fits quite well.expressions can be done
    perfectly.
```

```
commit 6983cd4766af199871cb3fb9106c752facf3e626 (HEAD -> master)
Author: 17307130178-Ning Chenran <17307130178@fudan.edu.cn>
Date:   Fri May 3 01:17:05 2019 -0700

    realized other exps,like "<< >> != ==" and "&& ||"

    updated the "-" : minus and negtive
```

```
commit daf81e2f6d91d0e171a6771cc85da5e04c1d0e67 (HEAD -> master)
Author: 17307130178-Ning Chenran <17307130178@fudan.edu.cn>
Date:   Fri May 3 00:02:33 2019 -0700

    add negtive "-1" exp
```

```
commit ae740f2eb6119a945cc870f3f3dbfd267831b895 (HEAD -> master)
Author: 17307130178-Ning Chenran <17307130178@fudan.edu.cn>
Date:   Fri May 3 02:00:01 2019 -0700

    realized register
```

```
commit 0767954451a841d6dad7e354e7fd03ab5e0279ce (HEAD -> master)
Author: 17307130178-Ning Chenran <17307130178@fudan.edu.cn>
Date:   Fri May 3 02:11:40 2019 -0700

    realized * pointer,now all the exps are done!
```

总结：表达式求值充分体现了分治法，每次单元测试，递归思想，最终合成复杂的表达式求值。

（二）、监视点

监视点 watchpoint，一开始对 gdb 里的调试监视点不太了解，看了资料后发现，主要分为：①初始化监视点②添加监视点 w expr③删除监视点 d n④显示监视点 info w⑤检查监视点 cpu_exec，操作集中在 nemu/src/monitor/debug/watchpoint.c ui.c cpu_exec.c

先定义头文件 watchpoint.h，设置 struct watchpoint 中添加表达式字符串，和初始表达式求值，以及该监视点是否备用的 bool 值（后来发现没有用到，可用做辅助）

```c
typedef struct watchpoint {
	int NO;
	struct watchpoint *next;

	/* TODO: Add more members if necessary */
	char * expr;    //expr to do
	int value;
	bool used;

} WP;

void init_wp_pool();
void new_wp(char * expr, int value);
void free_wp(int num);
void print_wp();
int expr_wp();
```

(1):初始化监视点

```c
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].expr = (char*)malloc(sizeof(char)*80);
        wp_pool[i].value = 0;
        wp_pool[i].used = false;
    }
    wp_pool[NR_WP - 1].next = NULL;

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

初始化 init_wp_pool()需要在每次开启 nemu 的时候初始化，只使用一次，需要对所有的值赋初值，此时表达式需要开空间。

Head 存放使用中的监视点的链表头；free_存放未使用的监视点的链表头。

(2):添加监视点 void new_wp(char * expr, int value)

```c
void new_wp(char *expr, int value)
{
    if (free_ == wp_pool + NR_WP){
        printf("No more free watchpoints!\n");
        assert(0);
    }
    //printf("wtf\n");
    WP *new_wp = free_;
    free_ = free_->next;
    //printf("free = %d\n",free_->NO);

    if(head == NULL) head = new_wp;
    else{
    WP* p = head;
    while(p){
        if(p->next == NULL)
            {p->next = new_wp; break;}
        p = p->next;
    }
    }
    new_wp->next = NULL;
    strcpy(new_wp->expr, expr);
    //printf("wtf2\n");
    new_wp->value = value;
    new_wp->used = true;

    printf("New watchpoint %d: \"%s\" = 0x%08x\n",new_wp->NO,new_wp->expr,new_wp->value);
}
```

添加监视点基本是链表操作。

如果 head 为空，则 head 更新为 new_wp；new_wp 指向 NULL；如果非空，head 不变，找到尾部指向 new_wp。同时需要修改 new_wp 中的内容，存放表达式字符串和当前表达式值，并且输出提示语句。

(3):删除监视点 void free_wp(int num)

删除标记为 num 的监视点，首先需要判断 head 是否为空，如果为空则说明没有监视点。

如果 head 非空，则遍历一遍使用中的监视点，需要注意如果 head 就是需要删除的 num，需要单独处理。此处的链表删去是返回初始化状态，并且把空的链表连接到 free_中，更新 free_。

```c
void free_wp(int num)
{
    WP *p = head;
    if (head == NULL)
    {
        printf("No watchpoint!\n");
        assert(0);
    }
    else
    {
        if (p->NO == num)
        {
            printf("Watchpoint %d deleted.\n", num);
            head = head->next;
            p->next = free_;
            free_ = p;
            p->value = 0;
            p->expr = (char*)malloc(sizeof(char)*80);
            p->used = false;
            return;
        }
        WP *q = p;
        p = p->next;
        while (p)
        {
            if (p->NO == num)
            {
                q->next = p->next;
                p->next = free_;
                free_ = p;
                p->value = 0;
                p->expr = (char*)malloc(sizeof(char)*80);
                p->used = false;
                printf("Watchpoint %d deleted.\n", num);
                return;
            }
            p = p->next;
            q = q->next;
        }
        printf("Watchpoint %d not exist.\n",num);
    }
    return;
}
```

(4):显示监视点

按顺序输出监视点，先检查 head 非空。

```c
void print_wp(){
    WP* p = head;
    if(p==NULL){
        printf("No watchpoints!\n");
        return;
    }
    else{
        printf("Watchpoints: \n");
        while(p){
            printf("Watchpoint %d : %s  = 0x%08x\n",p->NO,p->expr,p->value);
            p = p->next;
        }
    }
    return;
}
```

(5):检查监视点

```
int expr_wp()
{
    WP *p = head;
    if (p == NULL)
        return -1;
    else
    {
        while (p)
        {
            bool *success = false;
            if(expr(p->expr,success) != p->value){
                printf("Stop at watchpoint %d : %s = 0x%08x with old value = 0x%08x!\n", p->NO, p->expr, expr(p->expr, success), p->value);
                p ->value = expr(p->expr,success);
                return p->NO;
            }
            p = p->next;
        }
    }
    return -1;
}
```

每一次执行程序的时候都需要检查一下监视点的值是否发生变化，即遍历 head 的链表串，检查每一个表达式是否为原来存放的表达式求值结果，如果发生变化，则提示已经发生变化，停在了某个监视点。

(6):其他函数的更新

```
/* TODO: check watchpoints here. */

if(expr_wp()!=-1){
    nemu_state = STOP;
    return;
}
```

①Cpu_exec 函数中需要每次检查监视点:
如果发生变化，返回值不为-1，则 nemu_state=STOP 使程序停下，返回 nemu 界面。

②cmd_w()
这个函数是添加表达式监视点的，所以先设置 static 变量查看是否初始化过，正确初始化后，调用 new_wp 添加表达式监视点。

```
static int cmd_w(char *args)
{
    static bool whether_init = false;
    if(whether_init == false) {
        init_wp_pool();
        whether_init = true;
    }
    bool * success = false;
    new_wp(args,expr(args, success));
    return 0;
}
```

```
static int cmd_d(char *args)
{
    int num = 0;
    sscanf(args,"%d",&num);
    free_wp(num);
    return 0;
}
```

③cmd_d()
先把 num 正确求出来，再调用 free_wp 删除 num 监视点

④cmd_info()
直接 print_wp()即可

```
else if(strcmp(arg,"w") == 0)
{
    print_wp();
}
```

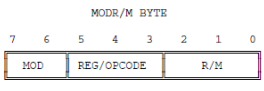
最后 git 记录。
这里发生了小插曲，由于代码版本有点问题，我想恢复到原来的状态，结果使用了 git reset –hard 导致表达式的 git log 全部消失了，但是代码却没有什么变化……很迷惑，而且害怕之前的 git 会导致我的代码有一些不一样，之后再看看。

```
commit 4aedbe78c19bdebd3c8cde885e9642fb9a7b9652 (HEAD -> master)
Author: 17307130178-Ning Chenran <17307130178@fudan.edu.cn>
Date:   Sat May 4 06:40:33 2019 -0700

    update watchpoint,all the works
```

(三)、i386 手册

科学阅读手册的能力很重要，检索与查找功能需要熟练运用。下面为必答题。
(1):查阅 i386 手册
①EFLAGS 寄存器中的 CF 位是什么意思？ （阅读范围 P33-34 P419）
答：CF 位是 carry flags - status flags 设置为高位进位或借位
搜索EFLAGS，可知是flag register，从目录看到为P33，找到CF的定义发现需要跳转至Appendix C的定义，搜索可知在P419，Carry Flag —— Set on high-order bit carry or borrow; cleared otherwise.
②ModR/M字节是什么？ （阅读范围 P241-242）
答：三部分：



The ModR/M byte contains three fields of information:

● The mod field, which occupies the two most significant bits of the byte, combines with the r/m field to form 32 possible values: eight registers and 24 indexing modes

● The reg field, which occupies the next three bits following the mod field, specifies either a register number or three more bits of opcode information. The meaning of the reg field is determined by the first (opcode) byte of the instruction.

● The r/m field, which occupies the three least significant bits of the byte, can specify a register as the location of an operand, or can form part of the addressing-mode encoding in combination with the field as described above

③mov指令的具体格式是怎么样的？（阅读范围 P345-347）

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
C1i1111	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

(2):shell 命令

find nemu/ -name *.c -or -name *.h | xargs cat | wc -l

筛选出 nemu 目录下 name 后缀为.c 和.h 的文件，并且输出总行数

如果需要去掉空行，则添加 grep -v ‘^$’过滤掉空行

find nemu/ -name *.c -or -name *.h | xargs cat | grep -v '^$$' |wc -l

```
chty627@ubuntu:~/ics2015$ make count2
find nemu/ -name *.c -or -name *.h | xargs cat | grep -v '^$' |wc -l
3556
chty627@ubuntu:~/ics2015$ make count
find nemu/ -name *.c -or -name *.h | xargs cat | wc -l
4346
```

(3):man -Wall -Werror

Gcc 中就有-Wall 和-Werror，目的是产生有用的警告信息。-Wall 能让编译器产生尽可能多的警告信息，有些虽然不是错误，但是可能会成为错误的来源，所以需要注意。而-Werror 要求 gcc 将所有的警告信息当作错误处理，这时候会在警告信息的地方停下来。

使用-Wall 有利于程序员养成优秀的代码习惯，-Werror 有利于程序员认真对待每一个不规范的警告处。

三、实验结果

（一）、表达式求值

```
(nemu) p 10 - 5 + 6 * (4 - -1) / 30
6
(nemu) p $eax
935382938
(nemu) p *0x100000
184
(nemu) p 1<<4-1||0
1
(nemu) p 1<<2>>2
1
(nemu) p ------10
10
```

检查之后无误，其中修改了很多次 debug。

（二）、监视点

所有功能均实现并检查

```
(nemu) w 3+4
New watchpoint 0: "3+4" = 0x00000007
(nemu) w 1*0
New watchpoint 1: "1*0" = 0x00000000
(nemu) w $eip == 0x100029
New watchpoint 2: "$eip == 0x100029" = 0x00000000
(nemu) info w
Watchpoints:
Watchpoint 0 : 3+4  = 0x00000007
Watchpoint 1 : 1*0  = 0x00000000
Watchpoint 2 : $eip == 0x100029  = 0x00000000
(nemu) d 0
Watchpoint 0 deleted.
(nemu) info w
Watchpoints:
Watchpoint 1 : 1*0  = 0x00000000
Watchpoint 2 : $eip == 0x100029  = 0x00000000
(nemu) c
Stop at watchpoint 2 : $eip == 0x100029 = 0x00000001 with old value = 0x00000000!
(nemu) info w
Watchpoints:
Watchpoint 1 : 1*0  = 0x00000000
Watchpoint 2 : $eip == 0x100029  = 0x00000001
```

（三）、必答题见上

实验源码见打包 nemu 文件

四、实验感受

本次实验更加深入了解了计算机的运作原理，实现表达式的时候经常会被递归的方法搞晕，不过熟练之后发现表达式只需要实现每一个单元的工作，并且保证完美无误，它就可以自动做出最好的结果。当然 KISS 原则也很重要，一定要多测试单元，以免 debug 的时候不知道怎么做，而且需要养成多写 printf 或者 debug 语句，否则会在中途很浪费时间。

监视点的知识大部分都是链表操作，所以还不是特别难。只是一定要记得每次需要分配字符串空间，还有避免空指针的操作。I386 的阅读，有点类似查资料的过程，数据库总是很大，需要我们自己阅读目录或者使用检索功能，这个在任何领域和学科都有用。

期待下次与你相遇！

[reference]
https://blog.csdn.net/qq_36982160/article/details/79717016
https://blog.csdn.net/DreamBitByBit/article/details/81088845