PA1实验报告

进度：完成了PA1所有内容

必答题

查阅i386手册

CF位

先查阅手册2.3.4 Flags Register部分

```
2.3.4.1 Status Flags
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.
```

查看附录C，得到：

```
Carry Flag –– Set on high-order bit carry or borrow; cleared otherwise.
```

ModR/M字节

查阅2.5 Operand Selection部分，在2.5.3 Memory Operands部分发现

```
 1. Most data-manipulation instructions that access memory contain a byte
 that explicitly specifies the addressing method for the operand. A
 byte, known as the modR/M byte, follows the opcode and specifies
 whether the operand is in a register or in memory. If the operand is
 in memory, the address is computed from a segment register and any of
 the following values: a base register, an index register, a scaling
 factor, a displacement. When an index register is used, the modR/M
 byte is also followed by another byte that identifies the index
 register and scaling factor. This addressing method is the
 mostflexible.
```

具体定义在17.2.1 ModR/M and SIB Bytes

mov指令的具体格式

从目录可知在17.2.2.11 Instruction Set Detail里，查看P345~P347

shell命令

统计代码行数

使用命令 `find . -name "*.[ch]" | xargs wc -l | grep "total" | awk '{ print $1}'` 统计代码行数

```
garzon@sixstars-XPS-8300:~/pa$ git checkout 15d1
Note: checking out '15d1'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name
```

```
HEAD is now at 15d163b... init
garzon@sixstars-XPS-8300:~/pa$ find . -name "*.[ch]" | xargs wc -l | grep "total" | awk '{ print $1}'
100498
garzon@sixstars-XPS-8300:~/pa$ git checkout master
Previous HEAD position was 15d163b... init
Switched to branch 'master'
garzon@sixstars-XPS-8300:~/pa$ make count
find . -name "*.[ch]" | xargs wc -l | grep "total" | awk '{ print }'
 100913 total
```

得到结果100913-100498=415行

```
D:\programming-assignment>git pull origin master
remote: Counting objects: 362, done.
remote: Compressing objects: 100% (319/319), done.
Remote: Total 362 (delta 243), reused 18 (delta 18), pack-reused 23
Receiving objects:  63% (229/362)
Receiving objects: 100% (362/362), 43.28 KiB | 0 bytes/s, done.
Resolving deltas: 100% (257/257), completed with 5 local objects.
From https://github.com/garzon/programming-assignment
 * branch            master     -> FETCH_HEAD
   15d163b..aa19080  master     -> origin/master
Updating 15d163b..aa19080
Fast-forward
 .gitignore                          |   1 +
 Makefile                            |   2 +
 config/Makefile.git                 |   2 +-
 nemu/include/cpu/reg.h              |  24 ++--
 nemu/include/monitor/expr.h         |   2 +-
 nemu/include/monitor/watchpoint.h   |  10 +-
 nemu/src/monitor/cpu-exec.c         |   4 +-
 nemu/src/monitor/debug/expr.c       | 274 +++++++++++++++++++++++++++++++++---
 nemu/src/monitor/debug/ui.c         | 120 +++++++++++++++-
 nemu/src/monitor/debug/watchpoint.c |  57 +++++++-
 10 files changed, 458 insertions(+), 38 deletions(-)
```

使用git统计的结果，与上述部分结果基本一致。

除去空行的代码行数

使用命令 `find . -name "*.[ch]"|xargs cat|grep -v ^$|wc -l`

```
garzon@sixstars-XPS-8300:~/pa$ find . -name "*.[ch]"|xargs cat|grep -v ^$|wc -l
91972
garzon@sixstars-XPS-8300:~/pa$ git checkout 15d1
Note: checking out '15d1'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 15d163b... init
garzon@sixstars-XPS-8300:~/pa$ find . -name "*.[ch]"|xargs cat|grep -v ^$|wc -l
91593
```

91972-91593=379

使用**man**

执行 `$ man gcc` ，可查找到

- `-Wall`

```
-Wall
   This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or
   modify to prevent the warning), even in conjunction with macros.  This also enables some language-specific warnings
   described in C++ Dialect Options and Objective-C and Objective-C++ Dialect Options.
```

总之就是开启所有编译器警告，警告程序员可能有潜在问题的地方。

- `-Werror`

```
-Werror
    Make all warnings into hard errors.  Source code which triggers warnings will be rejected.
```

这是强制把警告视为错误，只有没有出现警告的时候编译才能通过，强制程序员处理所有警告

实验报告

寄存器数据结构

```c
typedef struct {
    union {
        union {
            uint32_t _32;
            uint16_t _16;
            uint8_t _8[2];
        } gpr[CPU_REG_NUM];

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

        struct {
            uint32_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
        };
    };

    swaddr_t eip;

} CPU_state;
```

这也算是c的奇技淫巧了吧...匿名struct和union

Watchpoint

Watchpoint 结构体的定义，加入了expr用于存放表达式字符串，last_value存放上次求值时表达式的值

```c
typedef struct watchpoint {
    int NO;
    struct watchpoint *next;
    char expr[100];
    uint32_t last_value;
} WP;
```

维护链表即可

检查是否有表达式值变动，遍历一遍链表，并求值比较

```c
bool check_wp() {
        WP *p = head;
        bool flag = false, dummy;
        uint32_t new_val;
        while(p) {
                new_val = expr(p->expr, &dummy);
                if(new_val != p->last_value) {
                        flag = true;
                        printf("Watchpoint #%d: %s\n", p->NO, p->expr);
                        printf("Old value: 0x%08X %d\n", p->last_value, p->last_value);
                        p->last_value = new_val;
                        printf("New value: 0x%08X %d\n\n", p->last_value, p->last_value);
                }
                p = p->next;
        }
        return flag;
}
```

`cpu-exec.c` 中加一句

```
if(check_wp()) nemu_state = STOP;
```

表达式求值

表达式求值模块部分，我在这里对框架代码稍做了修改，为每一个token都加入了一个属性，称为大类category，取值如下：

```
enum {
    VALUE,
    OPERATOR,
    OTHERS,
    UNARY_OPERATOR
};
```

加入了category属性后会使得程序逻辑清晰一点，比如说可以做这样的检查，eval一个token时，token的category必须为VALUE才是合法的表达式。以及实现对于单目运算符 `*` , `-` 等的区分（即，解引用算符和乘法算符的type均为 `ASTERISK` ，而category分别为 `UNARY_OPERATOR` 和 `OPERATOR` ）。
这版程序里实现了单目运算符 `-` , `*` ，没有实现 `+` ，即 `+1` 是不合法的。

token的类型如下定义，这里手动取了些字符便于记忆:

```
enum {
    NOTYPE = 256, EQ = '=',

    REGISTER = 'r',
    PLUS = '+',
    MINUS = '-',
    ASTERISK = '*',
    SLASH = '/',
    LEFT_PAR = '(',
    RIGHT_PAR = ')',
    INTEGER = 'i',
    HEX_INTEGER = 'X',
    BANG = '!',
    LOGIC_OR = '|',
    LOGIC_AND = '&',

    /* TODO: Add more token types */

};
```

以及对应的匹配方式及大类，这里也加入category

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

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

    {" +", OTHERS, NOTYPE},
    {"\\+", OPERATOR, PLUS},
    {"==", OPERATOR, EQ},
    {"!=", OPERATOR, BANG},
    {"!", UNARY_OPERATOR, BANG},
    {"&&", OPERATOR, LOGIC_AND},
    {"\\|\\|", OPERATOR, LOGIC_OR},
    {"\\$[a-zA-Z]{2,3}", VALUE, REGISTER},
    {"-", OPERATOR, MINUS},
    {"\\*", OPERATOR, ASTERISK},
    {"\\/", OPERATOR, SLASH},
    {"\\(", OTHERS, LEFT_PAR},
    {"\\)", OTHERS, RIGHT_PAR},
    {"0x[0-9a-fA-F]+", VALUE, HEX_INTEGER},
    {"[0-9]+", VALUE, INTEGER}
};
```

利用单目运算符前必为运算符的特性区分单目运算符和双目运算符

```
    if(rules[i].token_type == '*' ||
        rules[i].token_type == '-')
    {
        if(nr_token == 0 ||
            tokens[nr_token-1].category == OPERATOR ||
            tokens[nr_token-1].category == UNARY_OPERATOR)
        {

            tokens[nr_token].category = UNARY_OPERATOR;
        }
    }
```

检测括号匹配的函数，遍历一次，途中左括号数恒大于等于右括号数：

```
bool check_parentheses(int p, int q) {
    int i, counter;
    if(tokens[p].type != '(') return false;
    if(tokens[q].type != ')') return false;
    counter = 0;
    for(i=p; i<=q; i++) {
        if(tokens[i].type == '(') counter++;
        if(tokens[i].type == ')') counter--;
        if(counter < 0) return invalid_expr("Parentheses do not match.");
    }
    return counter == 0;
}
```

从寄存器名取值的函数, strcasecmp是linux的不计大小写的比较字符串函数：

```
uint32_t register_eval(const char *reg_name) {
    const char *names_32[] = {
        "$EAX", "$ECX", "$EDX", "$EBX",
        "$ESP", "$EBP", "$ESI", "$EDI"
    };
    const char *names_16[] = {
        "$AX", "$CX", "$DX", "$BX",
        "$SP", "$BP", "$SI", "$DI"
    };
    const char *names_8l[] = {
        "$AL", "$CL", "$DL", "$BL"
    };
    const char *names_8h[] = {
        "$AH", "$CH", "$DH", "$BH"
    };
    int i;
    if(!strcasecmp(reg_name, "$EIP")) return cpu.eip;
    for(i=0; i<8; i++) {
        if(!strcasecmp(reg_name, names_32[i]))
            return cpu.gpr[i]._32;
    }
    for(i=0; i<8; i++) {
        if(!strcasecmp(reg_name, names_16[i]))
            return cpu.gpr[i]._16;
    }
    for(i=0; i<4; i++) {
        if(!strcasecmp(reg_name, names_8h[i]))
            return cpu.gpr[i]._8[1];
        if(!strcasecmp(reg_name, names_8l[i]))
            return cpu.gpr[i]._8[0];
    }
    return invalid_expr("Invalid register name");
}
```

这个函数定义运算的优先级：

```c
int op_priority(int op_pos) {
    switch(tokens[op_pos].type) {
        case '*': case '/': return 10;
        case '&': return 7;
        case '|': return 6;
        case '+': case '-': return 5;
        case '!': case '=': return 0;
        default:
            break;
    }
    printf("%s: ", tokens[op_pos].str);
    return invalid_expr("Unknown operator");
}
```

由于单目运算符的出现，我这里引入了“单项式”的概念，即最外层中没有双目运算符，形如 `!--(1+1)` 这种，此函数判断token区间内是否是“单项式”

```c
bool unary_operator_only(int p, int q) {
    int i, counter = 0; bool flag = false;
    for(i=p; i<=q; i++) {
        if(tokens[i].type == '(') {
            counter++;
            flag = true;
        }
        if(tokens[i].type == ')') counter--;
        if(counter) continue;
        if(flag && tokens[i].type != ')')
            return false;
        if(tokens[i].category == OPERATOR) return false;
    }
    return true;
}
```

最终的eval函数：

```c
uint32_t eval(int p, int q) {
    int op, counter = 0, old_op = -1;
    uint32_t val1, val2;
    if(p > q) {
        return invalid_expr("Unknown - overflow");
    }
    else if(p == q) {
        // 求值
        if(tokens[p].category != VALUE) {
            printf("%s: ", tokens[p].str);
            return invalid_expr("Not a valid value");
        }
        switch(tokens[p].type) {
            case INTEGER:
                return atoi(tokens[p].str);
            case HEX_INTEGER:
                return strtol(tokens[p].str, NULL, 16);
            case REGISTER:
                return register_eval(tokens[p].str);
            default:
                printf("%s: ", tokens[p].str);
                return invalid_expr("Unknown value type");
        }
    }
    else if(check_parentheses(p, q) == true) {
        // 去括号
        return eval(p + 1, q - 1);
    }
    else if(unary_operator_only(p, q)) {
        // 单目运算，即表达式为单项式
        assert(tokens[p].category == UNARY_OPERATOR);
        switch(tokens[p].type) {
            case '*':
                return swaddr_read(eval(p+1, q), 4);
            case '-':
                return -eval(p+1, q);
            case '!':
                return !eval(p+1, q);
            default:
                printf("%s: ", tokens[p].str);
                return invalid_expr("Unknown unary operator");
        }
    }
    else {
        // 处理“多项式”

        // 找到优先级最低的括号外的双目算符
        for(op = q; op >= p; op--) {
            if(tokens[op].type == ')') counter++;
            if(tokens[op].type == '(') counter--;
            if(counter == 0) {
                if(tokens[op].category == OPERATOR) {
                    if(check_op_priority(old_op, op)) {
                        old_op = op;
                    }
                }
            }
        }

        op = old_op;
        if(op == -1) return invalid_expr("No valid operator found");


        val1 = eval(p, op - 1);
        if(invalid) return invalid_expr("");
        val2 = eval(op + 1, q);
        if(invalid) return invalid_expr("");

        // 所有可能的双目运算
        switch(tokens[op].type) {
            case '+': return val1 + val2;
            case '-': return val1 - val2;
            case '*': return val1 * val2;
            case '/': return val1 / val2;
            case '|': return val1 || val2;
            case '&': return val1 && val2;
            case '=': return val1 == val2;
            case '!': return val1 != val2;
            default: return invalid_expr("Unknown operator");
        }
    }
}
```

其他部分

其他部分就没什么好说的了，都只是对已有函数进行封装，请参考 `ui.c` ，不再冗述。