

# 华中科技大学

## 2019

### 系统能力综合训练 课程设计报告

题 目： X86 模拟器设计

专 业： 计算机科学与技术

班 级： ACM1601

学 号： U201614787

姓 名： 宁嘉

电 话： 17371266562

邮 件： 2402224349@qq. com

完成日期： 2020-1-7



# 目 录

1	课程设计概述 .....	1
1.1	课设目的 .....	1
1.2	课设任务 .....	1
1.3	实验环境 .....	1
2	实验过程 .....	2
2.1	PA0 .....	2
2.1.1	总体设计 .....	2
2.1.2	运行结果 .....	2
2.2	PA1 .....	2
2.2.1	总体设计 .....	2
2.2.2	详细设计 .....	3
2.2.3	运行结果 .....	14
2.2.4	问题解答 .....	14
2.3	PA2 .....	19
2.3.1	总体设计 .....	19
2.3.2	详细设计 .....	19
2.3.3	运行结果 .....	26
2.3.4	问题解答 .....	28
2.4	PA3 .....	29
2.4.1	总体设计 .....	29
2.4.2	详细设计 .....	29
2.4.3	运行结果 .....	33
2.4.4	问题解答 .....	33
3	设计总结与心得 .....	35
3.1	课设总结 .....	35
3.2	课设心得 .....	35
	参考文献 .....	36

# 1 课程设计概述

## 1.1 课设目的

理解“程序如何在计算机上运行”的根本途径是从“零”开始实现一个完整的计算机系统。南京大学计算机科学与技术系计算机系统基础课程的小型项 (Programming Assignment, PA) 将提出 x86 架构的一个教学版子集 n86, 指导学生实现一个功能完备的 n86 模拟器 NEMU (NJU EMUlator), 最终在 NEMU 上运行游戏“仙剑奇侠传”, 来让学生探究“程序在计算机上运行”的基本原理。NEMU 受到了 QEMU 的启发, 并去除了大量与课程内容差异较大的部分。PA 包括一个准备实验 (配置实验环境) 以及 5 部分连贯的实验内容:

## 1.2 课设任务

## 1.3 实验环境

- Ubuntu 16.04

## 2 实验过程

### 2.1 PA0

#### 2.1.1 总体设计

PA0 阶段主要是环境的配置，主要是两个步骤。

先要修改一些用户信息(STUID 和 STUNAME)用于提交检查用。

再生成各个实验的分支。

除此之外由于是在校外做的实验，需要用反向代理连接到校内网，也要修改部分链接信息。

#### 2.1.2 运行结果

完成 PA0 的最终 git 记录如图 2-1 所示

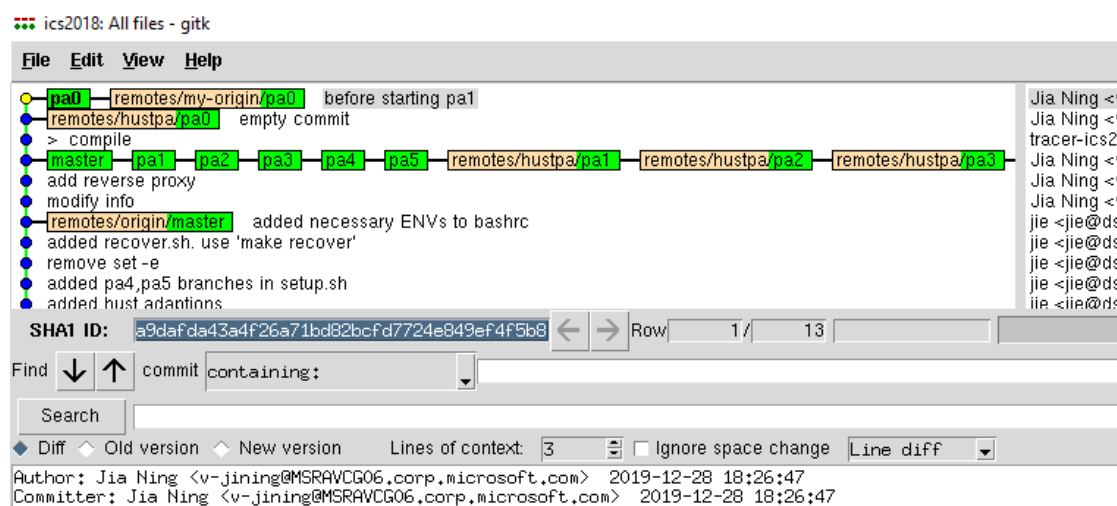


图 2-1 PA0 提交 git 记录

### 2.2 PA1

#### 2.2.1 总体设计

1. TRM 的实现:

- 存储器是个在 nemu/src/memory/memory.c 中定义的大数组
- PC 和通用寄存器都在 nemu/include/cpu/reg.h 中的结构体中定义
- 加法器等运算器 (PA2 介绍)
- TRM 的工作方式通过 cpu\_exec() 和 exec\_wrapper() 体现

2. 在 NEMU 中实现具有以下功能的简易调试器(相关部分的代码在 nemu/src/monitor/debug/目录下), 如表 2-1 所示

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

表 2-1 简易调试器功能

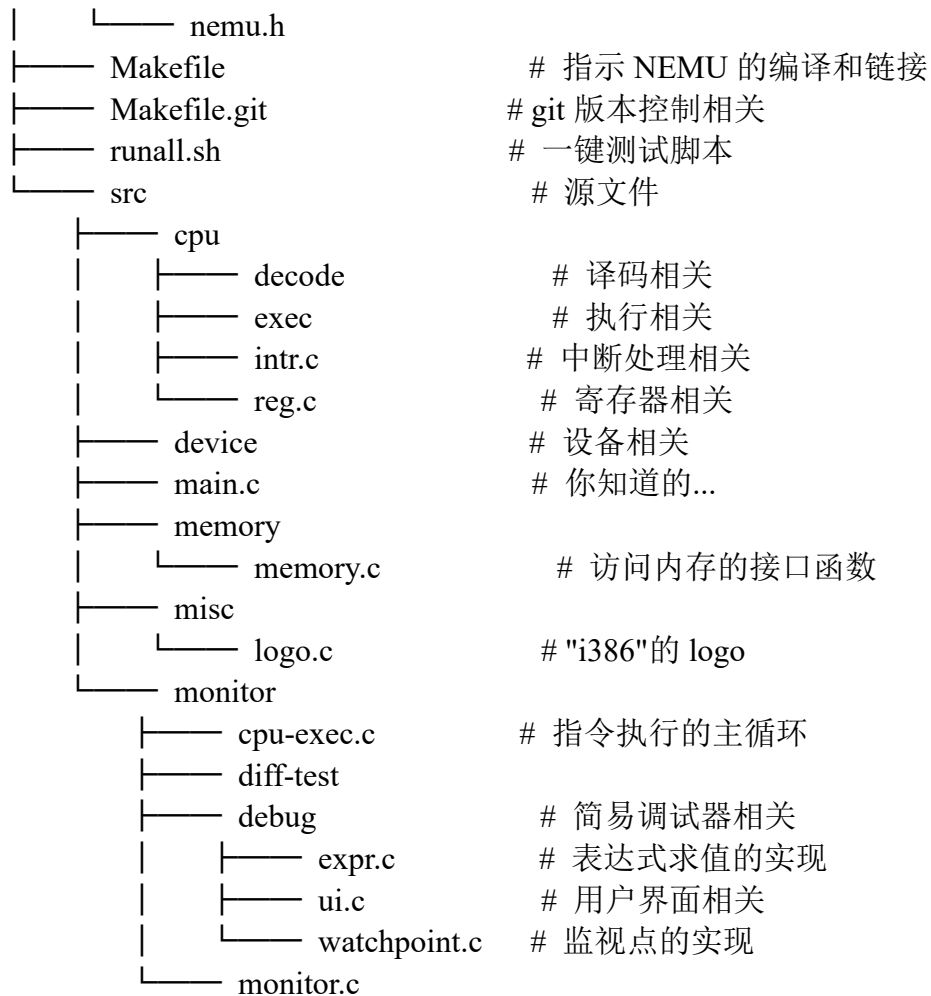
## 2.2.2 详细设计

code 的结构如下 (PA1 只关心 NEMU)

```

nemu
├── include                                # 存放全局使用的头文件
│   ├── common.h                         # 公用的头文件
│   ├── cpu
│   │   ├── decode.h                    # 译码相关
│   │   ├── exec.h                     # 执行相关
│   │   ├── reg.h                      # 寄存器结构体的定义
│   │   └── rtl.h                      # RTL 指令
│   ├── debug.h                         # 一些方便调试用的宏
│   ├── device                         # 设备相关
│   ├── macro.h                        # 一些方便的宏定义
│   ├── memory                         # 访问内存相关
│   ├── monitor
│   │   ├── expr.h                     # 表达式求值相关
│   │   ├── monitor.h
│   │   └── watchpoint.h               # 监视点相关

```



可以看出主结构分为三个部分 include 头文件部分、Makefile 与 sh 等脚本部分和 src 源代码文件部分。其中 include 或 src 又可以继续分为几个组成部分：cpu 部分、device 部分、memory 部分、monitor 部分，其中 monitor 部分主要用于 debug。

要知道我们的虚拟机系统从哪里开始运行的，只需要找到其中的 main.c 文件。如下：

```

int main(int argc, char *argv[]) {
    /* Initialize the monitor. */
    int is_batch_mode = init_monitor(argc, argv);

    /* Receive commands from user. */
    ui_mainloop(is_batch_mode);

    return 0;
}
  
```

可以看出先执行的是 init\_monitor 操作。

```

int init_monitor(int argc, char *argv[]) {
    /* Perform some global initialization. */
  
```

```

/* Parse arguments. */
parse_args(argc, argv);

/* Open the log file. */
init_log();

/* Test the implementation of the `CPU_state' structure. */
reg_test();

/* Load the image to memory. */
long img_size = load_img();

/* Initialize this virtual computer system. */
restart();

/* Compile the regular expressions. */
init_regex();

/* Initialize the watchpoint pool. */
init_wp_pool();

/* Initialize devices. */
init_device();

init_difftest(diff_so_file, img_size);

/* Display welcome message. */
welcome();

return is_batch_mode;
}

```

第一个代码任务主要是完成结构体 `CPU_state` 的实现，结构体 `CPU_state` 定义在 `nemu/include/cpu/reg.h`，需要进行修改，修改过后的结构体定义如下：

```

typedef struct {
    union {
        union {
            uint32_t _32;
            uint16_t _16;
            uint8_t _8[2];
        } gpr[8];

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

/* In NEMU, rtlreg_t is exactly uint32_t. This makes RTL instructions

```

```

    * in PA2 able to directly access these registers.
    */
    struct{
        rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
    };
};

vaddr_t eip;

} CPU_state;

```

这个时候在 nemu 目录下 make run 可以成功（图 2-2）：

```

(base) v-jining@MSRAVCG06:~/ics2018/nemu$ make run
./build/nemu -l ./build/nemu-log.txt -d /home/v-jining/ics2018/nemu/tools/qemu-diff/build/qemu-so
[src/monitor/monitor.c,54,load_default_img] No image is given. Use the default build-in image.
[src/monitor/monitor.c,28,welcome] Debug: ON
[src/monitor/monitor.c,31,welcome] If debug mode is on, A log file will be generated to record every instruction NEMU executes. This
not necessary, you can turn it off in include/common.h.
[src/monitor/monitor.c,36,welcome] Build time: 13:00:33, Dec 29 2019
Welcome to NEMU!
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100026

[src/monitor/cpu-exec.c,22,monitor_statistic] total guest instructions = 8
(nemu) q

```

图 2-2 make run 成功运行 nemu

nemu/src/cpu/exec/exec.c 中的 exec\_wrapper()函数用来执行当前%eip 指向的一条指令并更新%eip。在 nemu 输入 c 会显示 HIT GOOD TRAP，退出并查看 log 文件可以看到相应的 log 如图 2-3

```

(base) v-jining@MSRAVCG06:~/ics2018/nemu$ cat ./build/nemu-log.txt
[src/monitor/monitor.c,54,load_default_img] No image is given. Use the default build-in image.
[src/monitor/monitor.c,28,welcome] Debug: ON
[src/monitor/monitor.c,31,welcome] If debug mode is on, A log file will be generated to record every instruction NEMU executes. This
not necessary, you can turn it off in include/common.h.
[src/monitor/monitor.c,36,welcome] Build time: 13:00:33, Dec 29 2019
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
100017: 66 c7 84 99 00 e0 ff ff 01 00 movw $0x1,-0x2000(%ecx,%ebx,4)
100021: b8 00 00 00 00          movl $0x0,%eax
100026: d6                      nemu trap
nemu: HIT GOOD TRAP at eip = 0x00100026

[src/monitor/cpu-exec.c,22,monitor_statistic] total guest instructions = 8

```

图 2-3 运行的 log

代码中有三个有用的宏，源码很好理解：

```

#define Log(format, ...) \
    printflog("\33[1;34m[%s,%d,%s] " format "\33[0m\n", \
        __FILE__, __LINE__, __func__, ## __VA_ARGS__)

```

Log 对 printflog 宏函数再次进行了封装，即添加了终端颜色修饰和[文件名,行数,函数名]的信息。



```

#define Assert(cond, ...) \
do { \
    if (!(cond)) { \
        fflush(stdout); \
        fprintf(stderr, "\33[1;31m"); \
        fprintf(stderr, __VA_ARGS__); \
        fprintf(stderr, "\33[0m\n"); \
        assert(cond); \
    } \
} while (0)

```

Assert 当测试条件为假时，也会进行颜色的添加修饰和输出额外的参数信息。

```

#define panic(format, ...) \
    Assert(0, format, ## __VA_ARGS__)

```

panic 调用 Assert 并且总是 assertion fail。

内存在 nemu/src/memory/memory.c 中定义：

```
uint8_t pmem[PMEM_SIZE];
```

通过(p/v)addr\_(read/write)四个函数来进行(物理/虚拟)地址的(读/写)操作。

接下来是完成基础设施部分，即简易调试器的设计。

观察下面结构体数组

```

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

    /* TODO: Add more commands */
};

```

可以发现，一条指令简单的由 name, description, handler 组成。已经实现好了”help”，“c”，“q”，我们只需要实现其他的指令即可。

往 table 中加上其余的命令选项：

```

{"si", "Step one instruction exactly", cmd_si},
{"info", "Generic command for showing things about the program being debugged",
cmd_info},
{"p", "Print value of expression EXP", cmd_p},

```

```
{"x", "Examine memory: x N ADDRESS", cmd_x},  
{"w", "Set a watchpoint for an expression", cmd_w},  
{"d", "Delete watchpoint", cmd_d}
```

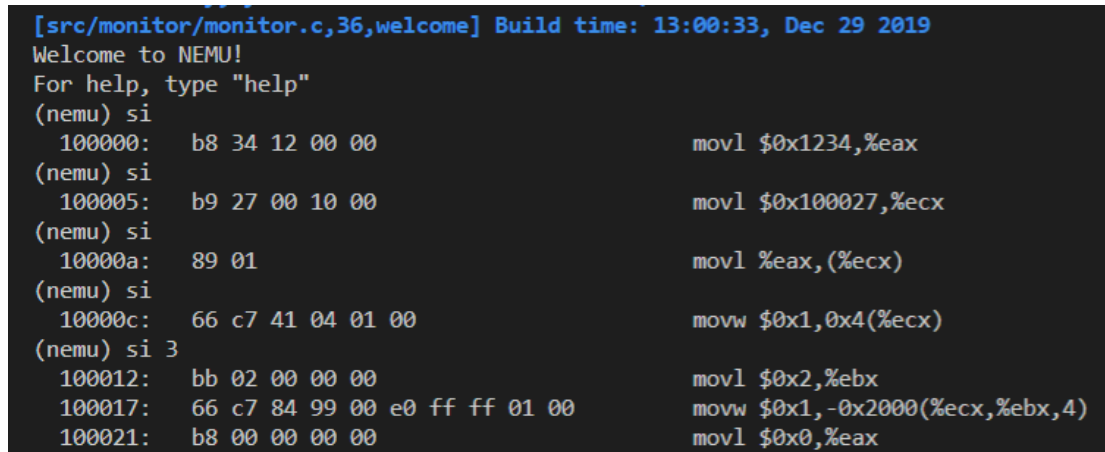
接下来实现这些选项（编写 cmd\_\* 对应的函数）

- cmd\_si 函数

```
static int cmd_help(char *args);  
  
static int cmd_si(char *args){  
    /* extract the first argument */  
    char *arg = strtok(NULL, " ");  
  
    if (arg == NULL) {  
        /* no argument given */  
        cpu_exec(1); // exec one step  
    }  
    else {  
        unsigned long count = strtoul(arg, NULL, 10);  
        cpu_exec(count); // exec count steps  
    }  
    return 0;  
}
```

cmd\_si 函数当输入只有一个参数的时候默认执行一条指令，否则执行其第二个参数那么多步。

完成函数后测试相应功能（图 2-4）：



```
[src/monitor/monitor.c,36,welcome] Build time: 13:00:33, Dec 29 2019  
Welcome to NEMU!  
For help, type "help"  
(nemu) si  
100000: b8 34 12 00 00      movl $0x1234,%eax  
(nemu) si  
100005: b9 27 00 10 00      movl $0x100027,%ecx  
(nemu) si  
10000a: 89 01              movl %eax,(%ecx)  
(nemu) si  
10000c: 66 c7 41 04 01 00    movw $0x1,0x4(%ecx)  
(nemu) si 3  
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)  
100021: b8 00 00 00 00      movl $0x0,%eax
```

图 2-4 si 功能测试

- cmd\_info 函数

```
static int cmd_info(char *args){  
    /* extract the first argument */  
    char *arg = strtok(NULL, "");  
    if (arg == NULL){
```

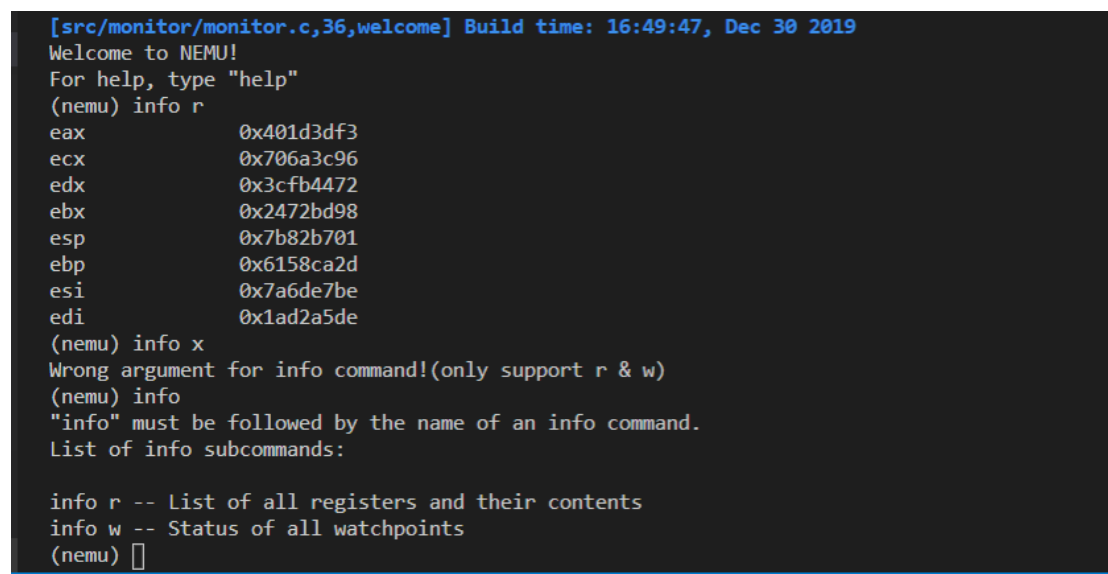
```

/* no argument given */
printf("\ninfo\n" must be followed by the name of an info command.\n"
"List of info subcommands:\n\n"
"info r -- List of all registers and their contents\n"
"info w -- Status of all watchpoints\n");
} else if (!strcmp(arg, "r")){
    for(int i = R_EAX; i <= R_EDI; ++i){
        printf("%s\t\t\t%x\n", reg_name(i, 4), reg_l(i));
    }

} else if (!strcmp(arg, "w")){
    print_wp();
} else {
    printf("Wrong argument for info command!(only support r & w)\n");
}
return 0;
}

```

由于还未设置 watchpoint，所以此处当 `arg == "w"` 的情况先暂时保留待完成 watchpoint 的时候再运行测试，这里只用 `print_wp` 函数进行占位。运行测试如图 2-5



```

[src/monitor/monitor.c,36,welcome] Build time: 16:49:47, Dec 30 2019
Welcome to NEMU!
For help, type "help"
(nemu) info r
eax          0x401d3df3
ecx          0x706a3c96
edx          0x3cfb4472
ebx          0x2472bd98
esp          0x7b82b701
ebp          0x6158ca2d
esi          0x7a6de7be
edi          0x1ad2a5de
(nemu) info x
Wrong argument for info command!(only support r & w)
(nemu) info
"info" must be followed by the name of an info command.
List of info subcommands:

info r -- List of all registers and their contents
info w -- Status of all watchpoints
(nemu) 

```

图 2-5 info 功能测试

- `cmd_p` 函数

由于 `cmd_x` 函数比较依赖于 `cmd_p` 函数进行表达式求值，所以这里先编写测试 `cmd_p` 函数，该函数用于表达式的求解，需要对表达式先进行词法分析再进行语法分析。

`make_token(char *e)` 函数对 `e` 进行词法分析，其中主要用到了 `regex` 函数来进行正则匹配，遍历所有的规则，若找到匹配的规则，则将该 `token` 从原 `string` 中取出，放入 `tokens` 数组中，放入数组代码如下：

```
switch (tokens[op].type)
```

```

{
case '+':
    return val1 + val2;
case '-':
    return val1 - val2;
case '*':
    return val1 * val2;
case '/':
    if (val2 == 0)
    {
        *success = false;
        printf("Divide zero encounting!\n");
        return 0;
    }
    else
        return val1 / val2;
default:
    *success = false;
    return 0;
}

```

即实现了从字符串

"4 +3\*(2- 1)"

到 tokens 数组

```

+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| NUM | '+' | NUM | '*' | '(' | NUM | '-' | NUM | ')' |
| "4" |      | "3" |      |    | "2" |      | "1" |      |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

的功能

词法分析完之后，接着对表达式进行语法分析，其中对表达式求值是通过栈很容易实现。但是在这里，按照材料提供的递归实现（虽然效率不高）这里就不具体阐述是如何实现的了，带 Log 的运行结果如图 2-6 所示：

```

Welcome to NEMU!
For help, type "help"
(nemu) p (1+2)*3
[src/monitor/debug/expr.c,99,make_token] match rules[5] = "\"(\" at position 0 with len 1: (
[src/monitor/debug/expr.c,99,make_token] match rules[8] = "[0-9][0-9]*" at position 1 with len 1: 1
[src/monitor/debug/expr.c,99,make_token] match rules[1] = "\"+\" at position 2 with len 1: +
[src/monitor/debug/expr.c,99,make_token] match rules[8] = "[0-9][0-9]*" at position 3 with len 1: 2
[src/monitor/debug/expr.c,99,make_token] match rules[6] = "\"(\" at position 4 with len 1: )
[src/monitor/debug/expr.c,99,make_token] match rules[3] = "\"*\" at position 5 with len 1: *
[src/monitor/debug/expr.c,99,make_token] match rules[8] = "[0-9][0-9]*" at position 6 with len 1: 3
[src/monitor/debug/expr.c,359,expr] nr_token = 7
[src/monitor/debug/expr.c,275,eval] inpar ++
[src/monitor/debug/expr.c,283,eval] find mult or div
[src/monitor/debug/expr.c,295,eval] success = 1
[src/monitor/debug/expr.c,299,eval] operator

[src/monitor/debug/expr.c,289,eval] find plus or minus
[src/monitor/debug/expr.c,295,eval] success = 1
[src/monitor/debug/expr.c,299,eval] operator
9

```

图 2-6 p expr 功能测试

进一步添加了对寄存器的识别与运算，如图 2-7 所示

```
Welcome to NEMU!
For help, type "help"
(nemu) info r
eax      0x6a805ce5
ecx      0x6a1c6dbf
edx      0x48b69fd2
ebx      0x4afcfa42
esp      0x5cc35ef4
ebp      0x19601981
esi      0x18851cab
edi      0x36e90979
(nemu) p ($eax / 2) * 3
[src/monitor/debug/expr.c,99,make_token] match rules[5] = "\" at position 0 with len 1: (
[src/monitor/debug/expr.c,99,make_token] match rules[9] = "\[a-zA-Z][a-zA-Z]+" at position 1 with len 4:
[src/monitor/debug/expr.c,99,make_token] match rules[0] = " " at position 5 with len 1:
[src/monitor/debug/expr.c,99,make_token] match rules[4] = "/" at position 6 with len 1: /
[src/monitor/debug/expr.c,99,make_token] match rules[0] = " " at position 7 with len 1:
[src/monitor/debug/expr.c,99,make_token] match rules[8] = "[0-9][0-9]*" at position 8 with len 1: 2
[src/monitor/debug/expr.c,99,make_token] match rules[6] = "\" at position 9 with len 1: )
[src/monitor/debug/expr.c,99,make_token] match rules[0] = " " at position 10 with len 1:
[src/monitor/debug/expr.c,99,make_token] match rules[3] = "\" at position 11 with len 1: *
[src/monitor/debug/expr.c,99,make_token] match rules[0] = " " at position 12 with len 1:
[src/monitor/debug/expr.c,99,make_token] match rules[8] = "[0-9][0-9]*" at position 13 with len 1: 3
[src/monitor/debug/expr.c,359,expr] nr_token = 7
[src/monitor/debug/expr.c,275,eval] inpar ++
[src/monitor/debug/expr.c,283,eval] find mult or div
[src/monitor/debug/expr.c,295,eval] success = 1
[src/monitor/debug/expr.c,299,eval] operator

[src/monitor/debug/expr.c,283,eval] find mult or div
[src/monitor/debug/expr.c,295,eval] success = 1
[src/monitor/debug/expr.c,299,eval] operator

2680195926
```

图 2-7 带寄存器值的 expr 求值

除了手动测试之外，为了更加完善的进行单元测试，我们需要随机生成大量表达式数据来进行测试，同样采用如下范式：

```
<expr> ::= <number>      # 一个数是表达式

| "(" <expr> ")"          # 在表达式两边加个括号也是表达式

| <expr> "+" <expr>      # 两个表达式相加也是表达式

| <expr> "-" <expr>

| <expr> "*" <expr>

| <expr> "/" <expr>
```

用于生成的代码文件是 nemu/tools/gen-expr/gen-expr.c，编译运行后会出现类似除 0 的情况

```
.code.c:2:90: warning: division by zero [-Wdiv-by-zero]
```

```
int main() { unsigned result = (((25)+(24)/6+24*((10*12)-(8)+28)*0/24/21/13-
20+(30)-5-2/((11/28)*(27)/30)*22+18+(12-9)+(14-9/(27*24)+24+11+1/8))/18-21-
18/20/2-13/(19+8)
```

要解决这个问题，得通过加-Werror 选项来使得 warning 变成 error，这样便可以检查相应的返回值来判断编译错误，从而达到剔除 divide by 0 的效果。

```
int ret = system("gcc .code.c -Werror -o .expr");
if (ret != 0)
```

```

{
    i--;
    continue;
}

```

这样便随机生成了一个 input 文件，随机枚举部分效果如下

```

4 18+14-28-6/(17+18*10+16)
236 26-8/(28)+3/11/(30)*14-30*((30-7+12-13)+((2)-1/1-30))
357 21*17
4199565112 ((27/12)*5*(27)*2/22-12*9+0+12*14/10-0+20-12/21/22/(30)-
15/12+(16*(17))/27-(17)+27-(26)*0+12+(5)/22*10-27)-26/30+20-
9*(15)*5*(0+16)-((29*27)*(21)*30)+(18)*21+(8/(24))-2*3+(0/4)*20-27+7*9-(3-
6-10/14-((9+5)+1-(10)))*(19*(15)*(2/3)/6+14+(29)+9+9+24*15+16/(20*(4)*(14)-
(22))-5-15/15/22+(18+3-(7*6))*(23-(7)+9+0)-26+27-15/19-((22*14)-22*30*(10)-
6+12-11)*(8)-22*26/25-(7*14*13/24)-((17+23)-1/21)*27-27/10-29/3/25+15-
4/9+4/11*(10))*(2-15*15-9*(0)-11/24/8)+(29)-13+12-
(21)/17/27/19*28*(26/25*(6+(16))+11*(6)+(12-9)-15+9/17+(3)+17-
(24)*(17/27))+7/5-29-(2)+((24/24)*29-11)-(26-30/10*11/13*(6)+(29)/7)+(4-16*2-
6)+9*9/(21)+(10)/24+(7)*(13)-19-15*24*12*21*((8)/8/1+(26)-(25)/22+17-
27)*((1*7)-11/4)+29+21+26/22-((21+(0))/21/24)+22+0*4+(7)-(24/22)*(6)-30-
(8/7)-(22)*29/(0/15+19-5)-(7/(3))/(4-18)/((12)*25)*18+(24)-5+(9)/(13)-
(0)+((4+5*12+12)-6+17+9/10)/23/28*30+10-22-6*(25-2)
0 (11/16)/14/6
15 15+6/(3*13)*2*1*26-15/(((24)+28)/1/(14)-4/4-(28)+10)
164 (24-6*(21*2))+(14)*28-12*0
4294967084 (29/5-(2)-4)-(1+29)-27/27+(0-5*(13+15))-7-3*(2)-27
4294957977 (29/2*19/15)/28+(3)-22/8+(0)-(2)/(4+(18))/3+6-(26-1)*3+20-
(11)*26-(22*23+12+0)-(19)*8/27-22*1*(23)*27-26*(20-10+(9+20)+(21)-6-
11*20+(8/(18)/19-6)-(30)/3+29-19-23-4*(16)-29/(22*9)+5+21-0-
2/18/(16)*26/5+6+27)
0 2-2
453 (5+9)*16-12+13*20-3-16
4294967294 18/10+(26)-29
4294966621 (8/19)-25*27
24 23/21-12/7/(26/20)/(10)+23

```

这里生成 1000 个样例进行测试，并在 main 函数中主要添加以下内容

```

init_regex();
FILE *fp;
if(!(fp = fopen("tools/gen-expr/input", "r")))
{

    printf("cannot open file input\n");
    exit(1);
}

```

```

unsigned res;
char str[2048];
bool success;
bool pass = true;
int cnt = 0;
while (fscanf(fp, "%u %[^\\n]", &res, str) == 2)
{
    // printf("%u\\t%s\\n", res, str);
    unsigned tmp = expr(str, &success);
    // printf("%d\\n", tmp);
    pass = pass && success && (tmp == res);
    cnt++;
}
fclose(fp);

if (pass)
    printf("success fully pass all %d test cases\\n", cnt);
else
    printf("error!\\n");

```

运行结果如下：

```

(base) v-jining@MSRAVC606:~/ics2018/nemu$ make run
+ CC src/main.c
+ LD build/nemu
./build/nemu -l ./build/nemu-log.txt -d /home/v-jining/ics2018/nemu/tools/qemu-diff/build/qemu-so
Current working dir: /home/v-jining/ics2018/nemu
success fully pass all 1000 test cases
[src/monitor/monitor.c,54,load_default_img] No image is given. Use the default build-in image.
[src/monitor/monitor.c,28,welcome] Debug: ON
[src/monitor/monitor.c,31,welcome] If debug mode is on, A log file will be generated to record every instruction NE
urn it off in include/common.h.
[src/monitor/monitor.c,36,welcome] Build time: 16:49:47, Dec 30 2019
Welcome to NEMU!
For help, type "help"
(nemu) 

```

图 2-8 expr 单元测试

发现 1000 条测试样例全部通过，测试成功。

- cmd\_x

实现 cmd\_x 指令较为简单，只需要用提供的 vaddr\_read 接口即可。实现完之后，x 10 0x100000 运行结果如图 2-9 所示

```

[src/monitor/monitor.c,54,load_default_img] No image is given. Use the default build-in image.
[src/monitor/monitor.c,28,welcome] Debug: ON
[src/monitor/monitor.c,31,welcome] If debug mode is on, A log file will be generated to record every instruction NE
urn it off in include/common.h.
[src/monitor/monitor.c,36,welcome] Build time: 16:49:47, Dec 30 2019
Welcome to NEMU!
For help, type "help"
(nemu) x 10 0x100000
0x100000:      0x1234b8      0x27b900      0x1890010     0x441c766
0x100010:      0x2bb0001     0x66000000    0x9984c7     0x1ffffe0
0x100020:      0xb800      0xd60000

```

图 2-9 x 10 0x100000 在 nemu 中的结果

而 image 的真实值可以在对应数组中观察到：

```

const uint8_t img [] = {
    0xb8, 0x34, 0x12, 0x00, 0x00,          // 100000:  movl
$0x1234,%eax
    0xb9, 0x27, 0x00, 0x10, 0x00,          // 100005:  movl
$0x100027,%ecx
    0x89, 0x01,                             // 10000a:
movl  %eax,(%ecx)
    0x66, 0xc7, 0x41, 0x04, 0x01, 0x00,    // 10000c:  movw
$0x1,0x4(%ecx)
    0xbb, 0x02, 0x00, 0x00, 0x00,          // 100012:  movl  $0x2,%ebx
    0x66, 0xc7, 0x84, 0x99, 0x00, 0xe0,    // 100017:  movw  $0x1,-
0x2000(%ecx,%ebx,4)
    0xff, 0xff, 0x01, 0x00,
    0xb8, 0x00, 0x00, 0x00, 0x00,          // 100021:  movl  $0x0,%eax
    0xd6,                                   // 100026:  nemu_trap
};

```

经过比对，发现确实是一致的。

- `cmd_w`

断点资源保存在断点池中，定义为 `static WP wp_pool[NR_WP];`；其中有两条链，一条 `free_` 指向的空闲链表，其上代表可以分配的断点；还有一条由 `head` 指向的使用链表，其上表示已经分配了的断点。

主要实现几个功能：

1. 初始化阶段，所有断点全在空闲链上。

```

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[NR_WP - 1].next = NULL;

    head = NULL;
    free_ = wp_pool;
}

```

### 2.2.3 运行结果

### 2.2.4 问题解答

1. 如果没有寄存器，计算机还可以工作吗？如果可以，这会对硬件提供的编程模型有什么影响呢？

答：如果没有寄存器，计算机还可以工作，那么每步计算都需要进行访存操作，会比较耗时。

2. 我们知道，时序逻辑电路里面有“状态”的概念。那么，对于 TRM 来说，是不是



答：也有这样的概念呢？具体地，什么东西表征了 TRM 的状态？在状态模型中，执行指令和执行程序，其本质分别是什么？

TRM 的状态是由程序计数器所表示的，即指明程序运行到了哪一步。在状态模型中，执行指令和执行程序，本质是一步操作的影响加上状态转移过程。

3. 阅读 `reg_test()` 的代码，思考代码中的 `assert()` 条件是根据什么写出来的。

答：下面一行行分析 `reg_test` 函数的源码

`srand(time(0));` 用当前时间做为随机数的种子。

`uint32_t eip_sample = rand();`

`cpu.eip = eip_sample;`

是的 `cpu_eip` 是随机生成的值进行测试。

下面是几个宏函数定义：

`#define reg_l(index) (cpu.gpr[check_reg_index(index)]._32)`

`#define reg_w(index) (cpu.gpr[check_reg_index(index)]._16)`

`#define reg_b(index) (cpu.gpr[check_reg_index(index) & 0x3]._8[index >> 2])`

参考修改过的 `reg` 结构体可抑制知道 `reg_l(index)` 表示取 32 位的 `index` 索引的寄存器。

同样 `reg_w(index)` 表示取的 `index` 索引的寄存器的低 16 位。`reg_b` 用来索引寄存器的后两个 8 位。

`sample[i] = rand();`

`reg_l(i) = sample[i];`

初始化每个寄存器的 32 位，再用 `assert` 比较其对应的低 16 位与两个 8 位是否正确。

4. 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？

答：-1 在传参的时候会自动转换为  $2^{64}-1$ ，即最大的执行长度。

5. "调用 `cpu_exec()` 的时候传入了参数 -1"，这一做法属于未定义行为吗？请查阅 C99 手册确认你的想法。

答：不属于，这一做法是 `safe` 的。可以参见 c99 标准

#### 6.3.1.8 Usual arithmetic conversions

1. If both operands have the same type, then no further conversion is needed.

2. Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.

3. Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.

4. Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of

the operand with signed integer type.

5. Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

参考第 3 条，由于 signed 和其对应的 unsigned 有相同的 rank，所以转换是安全的。具体转换方法手册中也有说明：

#### 6.3.1.3 Signed and unsigned integers

1. When a value with integer type is converted to another integer type other than `_Bool`, if the value can be represented by the new type, it is unchanged.

2. Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.

3. Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

6. `opcode_table` 到底是个什么类型的数组？

答：opcode 是 `opcode_entry` 类型的数组。`opcode_entry` 定义如下

```
typedef struct {  
    DHelper decode;  
    EHelper execute;  
    int width;  
} opcode_entry;
```

7. 对于 GNU/Linux 上的一个程序，怎么样才算开始？怎么样才算是结束？

对于在 NEMU 中运行的程序，问题的答案又是什么呢？

答：对于 GNU/Linux 上的程序，严格来说是一个进程，加载进内存被调度算运行开始，而结束即运行完成退出或被杀死。对于 NEMU 中的程序，包括 NEMU 事实上只有本身一个进程，不存在进程的调度，严格来说，在它上面跑的程序可以认为从他开始执行第一条指令起“运行”，在遇到 `nemu_trap` 函数的时候结束。

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？

答：500 \* 90% \* 20 \* 30 = 270000s = 75h

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？

答：节省  $500 * 90\% * 20 * (30-10) = 50h$  调试时间

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。

- 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：
  - EFLAGS 寄存器中的 CF 位是什么意思？

答：CF 位包含了无符号数加减法进位/退位信息，可以在 i386 手册 2.3.4 节，3.2 节和 Appendix C 中找到。

- ModR/M 字节是什么？

答：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

在 17.2.1 节可以找到。

- mov 指令的具体格式是怎么样的？

答: 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
8E	/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 ib		MOV reg8,imm8	2	Move immediate byte to register
B8 + rw iw		MOV reg16,imm16	2	Move immediate word to register
B8 + rd id		MOV reg32,imm32	2	Move immediate dword to register
C6 ib		MOV r/m8,imm8	2/2	Move immediate byte to r/m byte
C7 iw		MOV r/m16,imm16	2/2	Move immediate word to r/m word
C7 id		MOV r/m32,imm32	2/2	Move immediate dword to r/m dword

- shell 命令 完成 PA1 的内容之后, `nemu/` 目录下的所有 .c 和 .h 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 `pa1` 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?

答: 通过使用 `find . -name *.c -or -name *.h | xargs cat | wc -l` 指令可以获得代码的行数。

- 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`?

答: `man 1` 中是这样解释的:

`-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.

而

-Werror

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

可以看出 Wall 是显示所有 warning，而 Werror 是将所有 warning 视位 error，这两个选项有助于两方面，一方面是提前发现未定义行为造成的错误，另一方面，防止用户犯常见错误（如优先级等等）。

## 2.3 PA2

### 2.3.1 总体设计

PA2 实现了虚拟机的底层，即机器码的翻译过程，采用 x86 架构，并实现了 AM。

### 2.3.2 详细设计

先了解一下 exec\_wrapper 的执行流程：

```
vaddr_t ori_eip = cpu.eip;
```

```
decoding.seq_eip = ori_eip;
```

第一步，将全局译码信息 decoding.seq\_eip 设置为当前 eip，然后

```
exec_real(&decoding.seq_eip);
```

其中 exec\_real 函数是通过 make\_EHelper 宏来定义：

```
make_EHelper(real) {  
    uint32_t opcode = instr_fetch(eip, 1);  
    decoding.opcode = opcode;  
    set_width(opcode_table[opcode].width);  
    idex(eip, &opcode_table[opcode]);  
}
```

可以看出先通过 instr\_fetch 取指令，得到指令的第一个字节，将其解释为 opcode，同时送入全局 decoding.opcode 中，取完指令 opcode 后，eip++一个字节。set\_width 函数根据查找到的 opcode\_table 相应指令所对应的操作数宽度，将其记录在全局 decoding 中（decoding.src.width = decoding.dest.width = decoding.src2.width = width;）。最后再调用 idex() 对指令进行进一步的译码和执行。

运行 nexus-am/tests/cputest/tests/dummy.c 程序（make ARCH=x86-nemu ALL=dummy run）会出现如图 2-10 所示错误

```

Welcome to NEMU!
For help, type "help"
invalid opcode(eip = 0x0010000a): e8 01 00 00 00 90 55 89 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x0010000a is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x0010000a) in the disassembling result to distinguish which case it is.

If it is the first case, see
BOB Manual
for more details.

If it is the second case, remember:
* The machine is always right!
* Every line of untested code is always wrong!

nemu: ABORT at eip = 0x00100012

dummy

```

图 2-10 dummy 运行失败

失败原因是因为还有指令未实现。为了能成功运行 dummy 程序，需要完成以下指令：

- call

在此处只需要暂时实现 call rel32 的形式

查找到 call rel32 对应的 opcode 为 E8 cd，在 opcode\_table 中找到第 0xE8 个数据，发现暂时还是 EMPTY，需要为其填写入口（对应的 decode 和 exec 函数）。

decode 的过程事实上就是读取其之后的立即数，然后将跳转的相对地址对应的绝对地址存储在 decoding.jump\_eip 中，供执行的时候使用。这需要用到 make\_DHelper(J)宏，定义如下：

```

make_DHelper(J) {
    decode_op_SI(eip, id_dest, false);
    // the target address can be computed in the decode stage
    decoding.jump_eip = id_dest->siml + *eip; // relative jump
}

```

其中调用了 decode\_op\_SI 函数，而 decode\_op\_SI 函数如下定义（完成代码后）：

```

/* sign immediate */
static inline make_DopHelper(SI) {
    assert(op->width == 1 || op->width == 4);

    op->type = OP_TYPE_IMM;

    op->type = OP_TYPE_IMM;
    op->siml = instr_fetch(eip, op->width); // get op->width length signed imm
    op->siml = ((op->siml << (8*(4-op->width))) >> (8*(4-op->width))); //
signed extend the imm

    rtl_li(&op->val, op->siml);
}

```

```

#ifdef DEBUG
    snprintf(op->str, OP_STR_SIZE, "$0x%x", op->simmm);
#endif
}

```

其中立即数要注意进行符号扩展。

这样 decode 部分就完成了。接着完成 call 的 exec 部分,即 make\_EHelper(call) 宏展开对应的函数。

可以将 call 分解为两个部分,先是 rtl\_push 压栈,再是 rtl\_j 跳转。

- push

此处暂时只需要实现 push r32 的形式

查找到 push r32 对应的 opcode 为 50+rd, 译码的过程就是取出其 r32 作为目的的操作数。调用 decode\_op\_r(eip, id\_dest, true);, 此处 load\_val 参数取为 true, 即调用 rtl\_lr(&op->val, op->reg, op->width);, 执行句柄可以调用 make\_Ehelper(push) 来进行压栈操作

```

make_EHelper(push) {
    id_dest->val = id_dest->val << (8*(4-id_dest->width)) >> (8*(4-
id_dest->width)); // signed expend to id_dest->val
    rtl_push(&id_dest->val);
    print_asm_template1(push);
}

```

该函数先进行符号扩展,再调用 rtl\_push 进行压栈操作。

- sub

sub 会改变 eflags, 为此,我们需要先实现 eflags。

通过查阅手册可以获得 flags register 的结构如图 2-11 所示

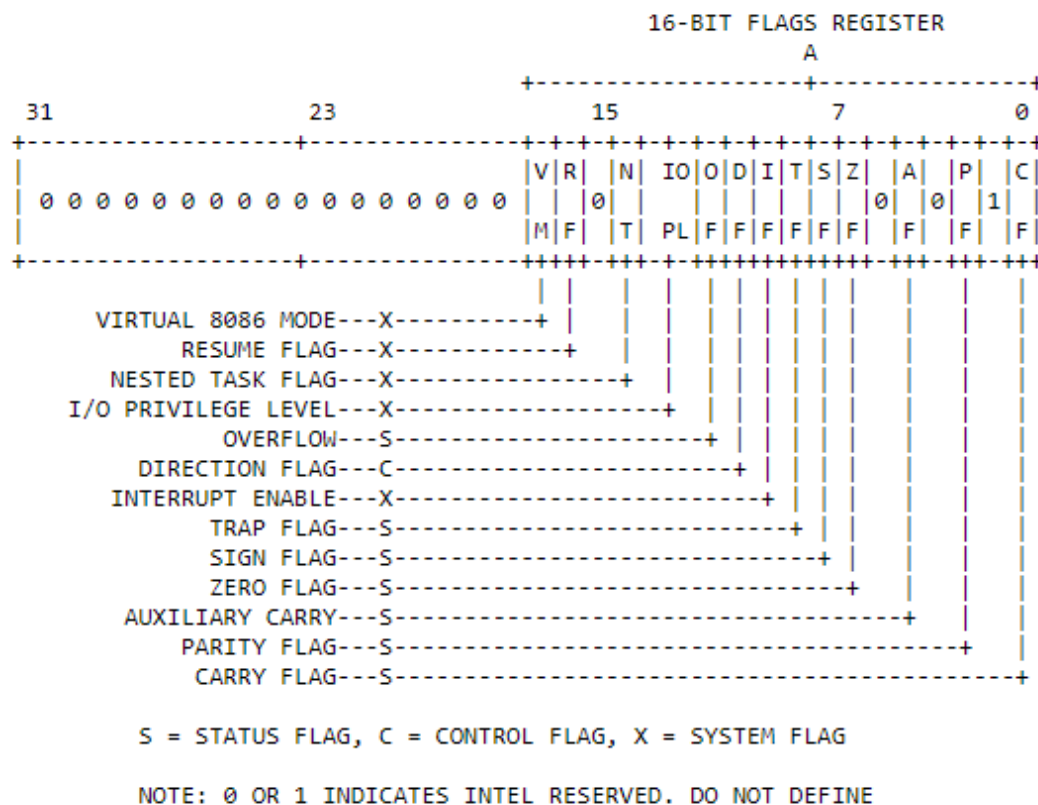


图 2-11 EFLAGS 结构

从而定义如下的 eflags 结构体:

```
union {
struct {
    uint8_t CF    : 1;
    uint8_t B1    : 1; // always 1, do not define
    uint8_t PF    : 1;
    uint8_t B2    : 1; // always 0
    uint8_t AF    : 1;
    uint8_t B3    : 1; // always 0
    uint8_t ZF    : 1;
    uint8_t SF    : 1;
    uint8_t TF    : 1;
    uint8_t IF    : 1;
    uint8_t DF    : 1;
    uint8_t OF    : 1;
    uint8_t IOPL  : 2;
    uint8_t NT    : 1;
    uint8_t B4    : 1; // always 0
    uint8_t RF    : 1;
    uint8_t VM    : 1;
    uint16_t B5   : 14; // always 0
} eflags;
```



```
uint32_t flags;
};
```

实现所有的 SUB 对应若干 opcode (opcode 从 0x28 到 0x2D 以及 0x80 到 0x83)。其中 0x80 到 0x83 的区间是若干指令共享 opcode 的。

其余 xor, pop, ret 等指令类似方法去实现。

完成指令和相应部分 rtl 之后, 再次运行, 结果如图 2-11 所示:

```
dummy
(base) v-jining@MSRAVCG06:~/ics2018/nexus-am/tests/cputest$ make ARCH=x86-nemu ALL=dummy run
Building dummy [x86-nemu] with AM_HOME {/home/v-jining/ics2018/nexus-am}
Building am [x86-nemu]
Building klib [x86-nemu]
Building compiler-rt [x86-nemu]
+ CC src/cpu/exec/exec.c
+ LD build/nemu
[src/monitor/monitor.c,72,load_img] The image is /home/v-jining/ics2018/nexus-am/tests/cputest/build/dummy-x86-nemu.t
[src/monitor/monitor.c,28,welcome] Debug: ON
[src/monitor/monitor.c,31,welcome] If debug mode is on, A log file will be generated to record every instruction NEMU
not necessary, you can turn it off in include/common.h.
[src/monitor/monitor.c,36,welcome] Build time: 10:52:13, Jan 3 2020
Welcome to NEMU!
For help, type "help"
nemu: HIT GOOD TRAP at eip = 0x0010001b

[src/monitor/cpu-exec.c,23,monitor_statistic] total guest instructions = 13
dummy
```

图 2-11 dummy 运行完成

将大部分要用到的 TODO()完成之后, 运行结果如图 2-12 所示, 发现仍然有部分 fail。

```
(base) v-jining@MSRAVCG06:~/ics2018/nemu$ bash runall.sh
compiling NEMU...
NEMU compile OK
compiling testcases...
testcases compile OK
[ add-longlong] FAIL! see add-longlong-log.txt for more information
[ add] PASS!
[ bit] PASS!
[ bubble-sort] FAIL! see bubble-sort-log.txt for more information
[ div] PASS!
[ dummy] PASS!
[ fact] PASS!
[ fib] PASS!
[ goldbach] PASS!
[ hello-str] PASS!
[ if-else] FAIL! see if-else-log.txt for more information
[ leap-year] PASS!
[ load-store] PASS!
[ matrix-mul] PASS!
[ max] FAIL! see max-log.txt for more information
[ min3] FAIL! see min3-log.txt for more information
[ mov-c] PASS!
[ movsx] PASS!
[ mul-longlong] PASS!
[ pascal] PASS!
[ prime] PASS!
[ quick-sort] FAIL! see quick-sort-log.txt for more information
[ recursion] runall.sh: line 24: 48072 Aborted (core dumped) $nemu -b -l $ori_log $file &> $logfile
FAIL! see recursion-log.txt for more information
[ select-sort] FAIL! see select-sort-log.txt for more information
[ shift] PASS!
[ shuixianhua] PASS!
[ string] FAIL! see string-log.txt for more information
[ sub-longlong] FAIL! see sub-longlong-log.txt for more information
[ sum] PASS!
[ switch] PASS!
[ to-lower-case] PASS!
[ unalign] PASS!
[ wanshu] PASS!
```

图 2-12 部分 fail

为了找到 bug, 先完成 difftest 部分, 再回过来用这个强大的工具来进行调试。其部分调试结果如图 2-12 所示:

```

LD build/nemu
[src/monitor/monitor.c:72,load_img] The image is /home/v-jining/ics2018/nexus-am/tests/cputest/build/add-longlong-x86-nemu.bin
[src/monitor/diff-test/diff-test.c:44,init_difftest] Differential testing: ON
[src/monitor/diff-test/diff-test.c:47,init_difftest] The result of every instruction will be compared with /home/v-jining/ics2018/nemu/tools/qemu-diff/build/qemu-so. This will help you a lot for
it also significantly reduce the performance. If it is not necessary, you can turn it off in include/common.h.
connect to QEMU successfully
[src/monitor/monitor.c:28,welcome] Debug: ON
[src/monitor/monitor.c:31,welcome] If debug mode is on, A log file will be generated to record every instruction NEMU executes. This may lead to a large log file. If it is not necessary, you can
an include/common.h.
[src/monitor/monitor.c:36,welcome] Build time: 19:46:21, Jan  4 2020
welcome to NEMU!
For help, type "help"
[src/monitor/diff-test/diff-test.c:79,difftest_step] QEMU: eax:ffffff ecx:0     edx:7fffffff ebx:18     esp:7ba4     ebp:7bd8     esi:0      edi:0      eip:100097
[src/monitor/diff-test/diff-test.c:81,difftest_step] NEMU: eax:ffffff ecx:0     edx:0       ebx:18     esp:7ba4     ebp:7bd8     esi:0      edi:0      eip:100097
[src/monitor/diff-test/diff-test.c:82,difftest_step] Registers differ at eip = 0x00100097
nemu: ABORT at eip = 0x00100097
qemu-system-x86_64: terminating on signal 15 from pid 47749
add-longlong

```

图 2-12 difftest 进行调试

最终发现是 CF 标志的计算有问题导致 adc 出错。修正完所有 bug 之后可以得到全部 pass。如图 2-13 所示：

```

(base) v-jining@MSRAVCG06:~/ics2018/nemu$ bash runall.sh
compiling NEMU...
NEMU compile OK
compiling testcases...
testcases compile OK
[ add-longlong] PASS!
[      add] PASS!
[      bit] PASS!
[ bubble-sort] PASS!
[      div] PASS!
[   dummy] PASS!
[    fact] PASS!
[    fib] PASS!
[ goldbach] PASS!
[ hello-str] PASS!
[  if-else] PASS!
[ leap-year] PASS!
[ load-store] PASS!
[ matrix-mul] PASS!
[    max] PASS!
[   min3] PASS!
[ mov-c] PASS!
[ movsx] PASS!
[ mul-longlong] PASS!
[  pascal] PASS!
[   prime] PASS!
[ quick-sort] PASS!
[  recursion] PASS!
[ select-sort] PASS!
[   shift] PASS!
[ shuixianhua] PASS!
[   string] PASS!
[ sub-longlong] PASS!
[    sum] PASS!
[  switch] PASS!
[ to-lower-case] PASS!
[   unalign] PASS!
[   wanshu] PASS!

```

图 2-13 测试样例 pass

对于设备的读取，要实现相应的 IO 指令，在 NEMU 层面对应的是 IN, OUT 的函数接口，完成之后，输出的测试结果如下：

```

[src/monitor/monitor.c,36,welcome] Build time: 10:52:27, Jan  5 2020
Welcome to NEMU!
For help, type "help"
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
nemu: HIT GOOD TRAP at eip = 0x00100037

```

图 2-14 IO 测试结果

接着完成时间的 timer.c 中函数的编写。

在 timetest 目录下 make run, 结果如图 2-15 所示

```

make[1]: Entering directory '/home/v-jining/ics2018/nexus-am/build'
./build/nemu -b -l /home/v-jining/ics2018/nexus-am/tests/timetest/build/nemu-log.txt /home/v-jining/ics2018/nexus-am/tests/timetest/build/qemu-so
[src/monitor/monitor.c,72,load_img] The image is /home/v-jining/ics2018/nexus-am/tests/timetest/build/qemu-so
[src/monitor/monitor.c,33,welcome] Debug: OFF
[src/monitor/monitor.c,36,welcome] Build time: 13:52:19, Jan  5 2020
Welcome to NEMU!
For help, type "help"
2018-0-0 00:00:00 GMT (1 second).
2018-0-0 00:00:00 GMT (2 seconds).
2018-0-0 00:00:00 GMT (3 seconds).
2018-0-0 00:00:00 GMT (4 seconds).
2018-0-0 00:00:00 GMT (5 seconds).
2018-0-0 00:00:00 GMT (6 seconds).
2018-0-0 00:00:00 GMT (7 seconds).
2018-0-0 00:00:00 GMT (8 seconds).
2018-0-0 00:00:00 GMT (9 seconds).
2018-0-0 00:00:00 GMT (10 seconds).
2018-0-0 00:00:00 GMT (11 seconds).
2018-0-0 00:00:00 GMT (12 seconds).
2018-0-0 00:00:00 GMT (13 seconds).
2018-0-0 00:00:00 GMT (14 seconds).
2018-0-0 00:00:00 GMT (15 seconds).
2018-0-0 00:00:00 GMT (16 seconds).
2018-0-0 00:00:00 GMT (17 seconds).

```

图 2-15 time 功能测试

再到 microbench 目录下 make run 进行跑分, 结果如图 2-16 所示

```

[src/monitor/monitor.c,36,welcome] Build time: 13:52:19, Jan  5 2020
Welcome to NEMU!
For help, type "help"
[qsrt] Quick sort: * Passed.
min time: 1872 ms [294]
[queen] Queen placement: * Passed.
min time: 2810 ms [183]
[bfs] Brainf**k interpreter: * Passed.
min time: 18223 ms [143]
[fib] Fibonacci number: * Passed.
min time: 37016 ms [77]
[sieve] Eratosthenes sieve: * Passed.
min time: 30274 ms [140]
[15pz] A* 15-puzzle search: * Passed.
min time: 5468 ms [105]
[dinic] Dinic's maxflow algorithm: * Passed.
min time: 4693 ms [288]
[lzip] Lzip compression: * Passed.
min time: 13308 ms [198]
[ssort] Suffix sort: * Passed.
min time: 2710 ms [218]
[md5] MD5 digest: * Passed.
min time: 25364 ms [77]
=====
MicroBench PASS          172 Marks
                        vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x00100032

```

图 2-16 microbench 跑分

完成 input.c 后，在 keytest 目录下 make run 结果如图 2-17 所示

```
Get key: 31 E up
Get key: 51 L down
Get key: 51 L up
Get key: 51 L down
Get key: 51 L up
Get key: 37 O down
Get key: 37 O up
Get key: 30 W down
Get key: 30 W up
Get key: 37 O down
Get key: 37 O up
Get key: 32 R down
Get key: 32 R up
Get key: 51 L down
Get key: 51 L up
Get key: 45 D down
Get key: 45 D up
Get key: 66 RSHIFT down
Get key: 66 RSHIFT up
Get key: 75 LEFT down
Get key: 75 LEFT up
Get key: 75 LEFT down
Get key: 75 LEFT up
Get key: 76 RIGHT down
Get key: 76 RIGHT up
Get key: 73 UP down
Get key: 73 UP up
Get key: 74 DOWN down
Get key: 74 DOWN up
Get key: 55 LSHIFT down
```

图 2-17 读取键盘输入测试

类似的在 video.c 中添加支持，并且完成内存映射 I/O 的添加。

### 2.3.3 运行结果



图 2-18 彩色图

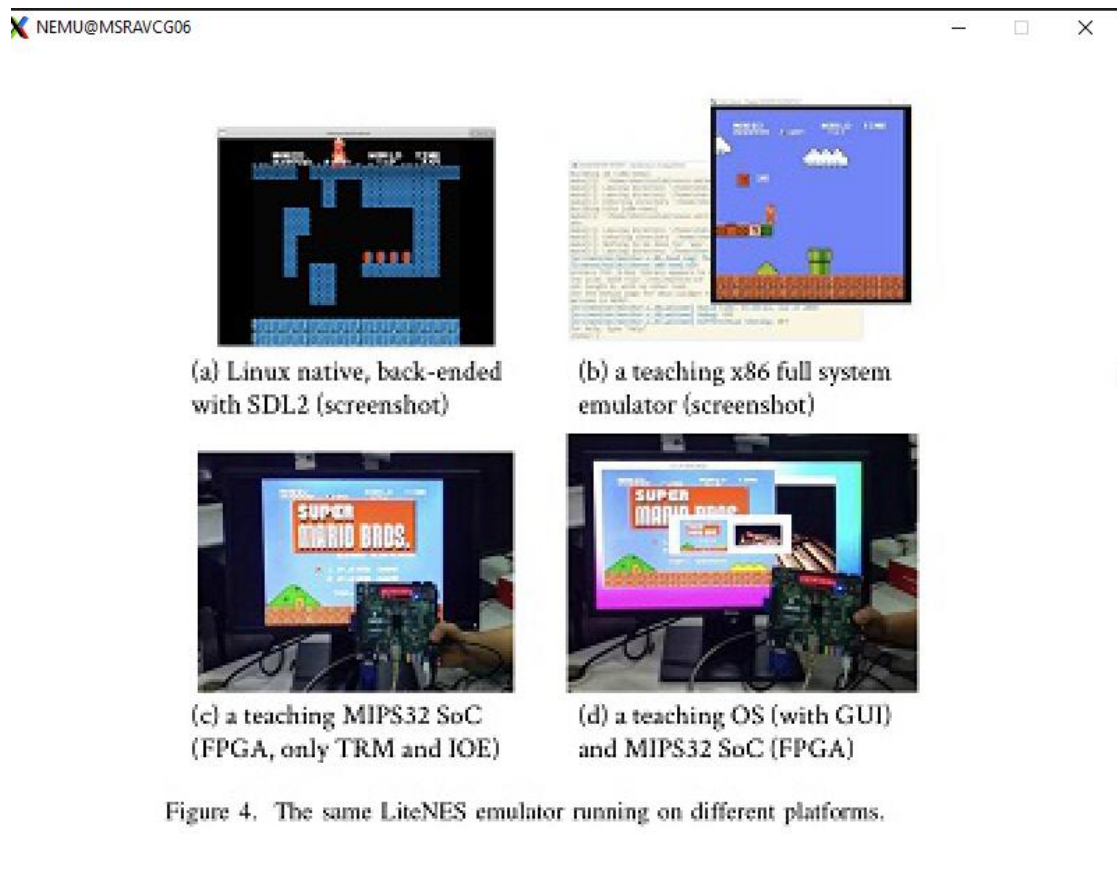


图 2-19 PPT 系统展示



图 2-20 打字小游戏

### 2.3.4 问题解答

必答题：

- 编译与链接 在 `nemu/include/cpu/rtl.h` 中，你会看到由 `static inline` 开头定义的各种 RTL 指令函数。选择其中一个函数，分别尝试去掉 `static`，去掉 `inline` 或去掉两者，然后重新进行编译，你可能会看到发生错误。请分别解释为什么这些错误会发生/不发生？你有办法证明你的想法吗？

答：分别去掉 `static` 或者 `inline` 都不会发生错误，但是一起去掉就会发生错误。都去掉会产生重定义的冲突，因为它在所有文件都可见了。

- 编译与链接

1. 在 `nemu/include/common.h` 中添加一行 `volatile static int dummy`；然后重新编译 NEMU。请问重新编译后的 NEMU 含有多少个 `dummy` 变量的实体？你是如何得到这个结果的？

答：`static` 修饰表示每个文件独有一个 `dummy`，而 `volatile` 告诉系统不需要优化掉 `dummy`，所以有多少个文件包含了 `common.h` 头就有多少个变量实体。

2. 添加上题中的代码后，再在 `nemu/include/debug.h` 中添加一行 `volatile static int dummy`；然后重新编译 NEMU。请问此时的 NEMU 含有多少个 `dummy` 变量的实体？与上题中 `dummy` 变量实体数目进行比较，并解释本题的结果。

答：和上题数目相同，只是添加了声明而已。

3. 修改添加的代码，为两处 `dummy` 变量进行初始化：`volatile static int dummy = 0`；然后重新编译 NEMU。你发现了什么问题？为什么之前没有出现这样的问题？（回答完本题后可以删

除添加的代码.)

答: 出现了 **error**。因为变量可以声明多次但是不能定义多次。

- 了解 Makefile 请描述你在 **nemu/**目录下敲入 **make** 后, **make** 程序如何组织.c 和.h 文件, 最终生成可执行文件 **nemu/build/nemu**。(这个问题包括两个方面:Makefile 的工作方式和编译链接的过程。)关于 Makefile 工作方式的提示:

- Makefile 中使用了变量, 包含文件等特性
- Makefile 运用并重写了一些 implicit rules
- 在 **man make** 中搜索-n 选项, 也许会对你有帮助
- RTFM

```
SRCS = $(shell find src/ -name "*.c" | grep -v "isa")
```

```
SRCS += $(shell find src/isa/${ISA} -name "*.c")
```

```
INC_DIR += ./include ./src/isa/${ISA}/include
```

## 2.4 PA3

### 2.4.1 总体设计

PA3 要实现一个最简单的批处理操作系统。

### 2.4.2 详细设计

PA3 开始要注意几个东西。

首先要将代码中的 **printk** 改为/定义为 **printf**, 不然会出现错误。

然后开始要先在 **restart** 函数初始化中将 **CS** 寄存器定义为初始化为 8, 将 **EFLAGS** 初始化为 0x2, 接着完成 **lidt** 指令。**LIDT** 指令的行为描述经查手册如下:

```
IF Instruction is LIDT
    THEN
        IF OperandSize = 16
            THEN
                IDTR(Limit) ← SRC[0:15];
                IDTR(Base) ← SRC[16:47] AND 00FFFFFFFH;
            ELSE IF 32-bit Operand Size
                THEN
                    IDTR(Limit) ← SRC[0:15];
                    IDTR(Base) ← SRC[16:47];
                FI;
            ELSE IF 64-bit Operand Size (* In 64-Bit Mode *)
                THEN
                    IDTR(Limit) ← SRC[0:15];
                    IDTR(Base) ← SRC[16:79];
                FI;
            FI;
```

按照以上伪代码的 16 位和 32 位形式即可完成 **LIDT** 函数的编写。

完成了第一个小任务之后, 再次 **make ARCH=x86-nemu run**, 仍然会有如图



2-21 所示 error。

```
[src/monitor/monitor.c,36,welcome] Build time: 20:44:06, Jan 5 2020
Welcome to NEMU!
For help, type "help"
[src/main.c,15,main] 'Hello World!' from Nanos-lite
[src/main.c,16,main] Build time: 20:36:47, Jan 5 2020
[src/ramdisk.c,28,init_ramdisk] ramdisk info: start = 1051689, end = 1058065, size = 6376 bytes
[src/device.c,31,init_device] Initializing devices...
[src/irq.c,12,init_irq] Initializing interrupt/exception handler...
[src/irq.c,5,do_event] system panic: Unhandled event ID = 1
nemu: HIT BAD TRAP at eip = 0x00100032

[src/monitor/cpu-exec.c,23,monitor_statistic] total guest instructions = 20600
make[1]: Leaving directory '/home/v-jining/ics2018/nemu'
```

图 2-21 system panic: Unhandled event ID = 1

接着完成之后的任务，在 `irq_handle` 中查看 `tf` 确定事件，最终在 `do_event` 中进行事件处理。完成阶段一后，我们在 `Trap` 的时候输出一条自定义的信息，如图 2-22 所示。

```
s/qemu-diff/build/qemu-so
[src/monitor/monitor.c,72,load_img] The image is /home/v-jining/ics2018/nanos-lite/build/
[src/monitor/monitor.c,33,welcome] Debug: OFF
[src/monitor/monitor.c,36,welcome] Build time: 20:44:06, Jan 5 2020
Welcome to NEMU!
For help, type "help"
[src/main.c,15,main] 'Hello World!' from Nanos-lite
[src/main.c,16,main] Build time: 21:51:50, Jan 5 2020
[src/ramdisk.c,28,init_ramdisk] ramdisk info: start = 1051769, end = 1058145, size = 637
[src/device.c,31,init_device] Initializing devices...
[src/irq.c,14,init_irq] Initializing interrupt/exception handler...
System Trap in do_event.
[src/main.c,34,main] system panic: Should not reach here
nemu: HIT BAD TRAP at eip = 0x00100032

[src/monitor/cpu-exec.c,23,monitor_statistic] total guest instructions = 21467
make[1]: Leaving directory '/home/v-jining/ics2018/nemu'
```

图 2-22 trap 时输出自定义信息

开始完成第二个任务，完成了 `loader` 函数并在 `init_proc()` 中调用 `naive_uload(NULL, NULL)`，它会调用 `loader` 来加载第一个用户程序，然后跳转到用户程序中执行。运行结果如图 2-23 所示。触发了一个未处理的 1 号事件。

```
s/qemu-diff/build/qemu-so
[src/monitor/monitor.c,72,load_img] The image is /home/v-jining/ics2018/nanos-lite/build/nanos-lite-x86-nemu
[src/monitor/monitor.c,33,welcome] Debug: OFF
[src/monitor/monitor.c,36,welcome] Build time: 20:44:06, Jan 5 2020
Welcome to NEMU!
For help, type "help"
[src/main.c,15,main] 'Hello World!' from Nanos-lite
[src/main.c,16,main] Build time: 21:51:50, Jan 5 2020
[src/ramdisk.c,28,init_ramdisk] ramdisk info: start = 1051953, end = 1058329, size = 6376 bytes
[src/device.c,31,init_device] Initializing devices...
[src/irq.c,14,init_irq] Initializing interrupt/exception handler...
[src/irq.c,7,do_event] system panic: Unhandled event ID = 1
nemu: HIT BAD TRAP at eip = 0x00100032

[src/monitor/cpu-exec.c,23,monitor_statistic] total guest instructions = 52572
make[1]: Leaving directory '/home/v-jining/ics2018/nemu'
(base) v-jining@MSRAVCG06:~/ics2018/nanos-lite$
```

图 2-23 loader dummy

添加 `syscall` 调用之后，会显示 `[src/syscall.c,15,do_syscall] system panic: Unhandled syscall ID = 0`，如图 2-24 所示。



```

make[1]: Leaving directory '/home/v-jining/ics2018/nexus-ami/lib/compiler-rt'
make[1]: Entering directory '/home/v-jining/ics2018/nemu'
./build/nemu -b -l /home/v-jining/ics2018/nanos-lite/build/nemu-log.txt /home/v-jining/ics2018/nanos-lite/
s/qemu-diff/build/qemu-so
[src/monitor/monitor.c,72,load_img] The image is /home/v-jining/ics2018/nanos-lite/build/nanos-lite-x86-ne
[src/monitor/monitor.c,33,welcome] Debug: OFF
[src/monitor/monitor.c,36,welcome] Build time: 20:44:06, Jan 5 2020
Welcome to NEMU!
For help, type "help"
[src/main.c,15,main] 'Hello World!' from Nanos-lite
[src/main.c,16,main] Build time: 10:42:09, Jan 6 2020
[src/ramdisk.c,28,init_ramdisk] ramdisk info: start = 1052233, end = 1058609, size = 6376 bytes
[src/device.c,31,init_device] Initializing devices...
[src/irq.c,17,init_irq] Initializing interrupt/exception handler...
system yeild
[src/syscall.c,15,do_syscall] system panic: Unhandled syscall ID = 0
nemu: HIT BAD TRAP at eip = 0x00100032

[src/monitor/cpu-exec.c,23,monitor_statistic] total guest instructions = 53567
make[1]: Leaving directory '/home/v-jining/ics2018/nemu'

```

图 2-24 未处理 0 号 syscall id

查看 nanos-lite/src/syscall.h, 你会发现 0 号是一个 SYS\_exit 系统调用. 这说明之前的 SYS\_yield 已经成功返回, 触发 SYS\_exit 是因为 dummy 已经执行完毕, 准备退出了。只需要完成 SYS\_exit 使得程序安全退出即可。

```
case SYS_exit:
```

```
    printf("system exit\n");
```

```
    _halt(a[1]);
```

```
    break;
```

添加了 SYS\_exit 入口之后, 通过 \_halt 可以让程序 hit good trap, 如图 2-25 所示。

```

s/qemu-diff/build/qemu-so
[src/monitor/monitor.c,72,load_img] The image is /home/v-jining/ics2018/nanos-lite/build/nan
[src/monitor/monitor.c,33,welcome] Debug: OFF
[src/monitor/monitor.c,36,welcome] Build time: 20:44:06, Jan 5 2020
Welcome to NEMU!
For help, type "help"
[src/main.c,15,main] 'Hello World!' from Nanos-lite
[src/main.c,16,main] Build time: 10:42:09, Jan 6 2020
[src/ramdisk.c,28,init_ramdisk] ramdisk info: start = 1052273, end = 1058649, size = 6376 by
[src/device.c,31,init_device] Initializing devices...
[src/irq.c,17,init_irq] Initializing interrupt/exception handler...
system yeild
system exit
nemu: HIT GOOD TRAP at eip = 0x00100032

```

图 2-25 进程正常退出 (HIT GOOD TRAP)

完成堆区的分配之后, 得到的效果仍然如图 2-25 所示, 只是从 LOG 中可以看到 是一次输出而不是多次输出的。

之后是完成一系列文件操作, 在 nanos-lite 中的文件函数接口遵循 linux 系统调用的接口, 查看 MAN 可以得到各个参数的意义, 从而编写相应的函数。以 lseek man 为例子, 如下, 可以看出各个参数的含义和函数的功能。

lseek() repositions the file offset of the open file description associated with the file descriptor fd to the argument offset according to the directive whence as follows:

SEEK\_SET

The file offset is set to offset bytes.

SEEK\_CUR

The file offset is set to its current location plus offset

bytes.  
SEEK\_END  
The file offset is set to the size of the file plus offset  
bytes.

同时修改 loader 的文件镜像和方式，将 proc.c 中加载运行的程序改为/bin/text。  
运行成功后如图 2-26 所示

```
[src/monitor/monitor.c,36,welcome] Build time: 20:44:06, Jan 5 2020
Welcome to NEMU!
For help, type "help"
[src/main.c,15,main] 'Hello World!' from Nanos-lite
[src/main.c,16,main] Build time: 10:42:09, Jan 6 2020
[src/ramdisk.c,28,init_ramdisk] ramdisk info: start = 1054200, end = 332657
[src/device.c,36,init_device] Initializing devices...
[src/irq.c,17,init_irq] Initializing interrupt/exception handler...
PASS!!!
nemu: HIT GOOD TRAP at eip = 0x00100032
```

图 2-26 文件系统通过

接下来按照提示将 VGA 显存抽象成文件，完成相应函数之后，会出现 projectN 的 logo，如图 2-27 所示。



图 2-27 projectN 图标

完成将输入输出设备抽象成文件之后，运行/bin/events，结果如图 2-28 所示

```

receive event: time 726
receive event: time 1422
receive event: time 2043
receive event: time 2693
receive event: time 3316
receive event: time 4548
receive event: time 5137
receive event: time 5791
receive event: time 6433
receive event: time 7034
receive event: time 7647
receive event: time 8236
receive event: time 8867
receive event: time 9481
receive event: time 10104
receive event: time 10731
receive event: time 11363
[src/device.c,33,events_read] key = 46

receive event: kd F
[src/device.c,33,events_read] key = 46

receive event: ku F
[src/device.c,33,events_read] key = 66

receive event: kd RSHIFT
[src/device.c,33,events_read] key = 66

receive event: ku RSHIFT
[src/device.c,33,events_read] key = 46

receive event: kd F
[src/device.c,33,events_read] key = 46

```

图 2-28 时间键盘设备读取

### 2.4.3 运行结果

### 2.4.4 问题解答

文件读写的具体过程 仙剑奇侠传中有以下行为:

在 navy-apps/apps/pal/src/global/global.c 的 PAL\_LoadGame()中通过 fread()读取游戏存档

在 navy-apps/apps/pal/src/hal/hal.c 的 redraw()中通过 NDL\_DrawRect()更新屏幕

请结合代码解释仙剑奇侠传, 库函数, libos, Nanos-lite, AM, NEMU 是如何相互协助, 来分别完成游戏存档的读取和屏幕的更新.

答: 在 PAL\_LoadGame()中通过 fread()读档, 首先调用了 libc 中的 fread()函数, 然后 libc 调用 libo 中的 \_read()函数, 进行一系列的系统调用, 陷入到内核态, 即通过 \_\_syscall\_\_()函数中的 int 指令, 进行中断处理, 将控制权交付给系统内核。系统调用的入口为 \_\_am\_vecsys, 此时还在 nexus-am 中, 可以认为 am 是操作系统的内核部分, 与 Nasnos-lite 紧密联系, 由 Nasnos-lite 处理相应的 syscall, 在 do\_syscall 中接收到该事件, 根据系统调用号执行 sys\_read()中的 fs\_read 进行文件系统的文件操作。

NDL\_DrwaRect()函数类似，首先调用 libnd1 库，在该库中和以上同样的步骤进行设备文件的操作，打开/dev/fb 和/dev/fbsync 设备文件。然后再类似的进行文件的写入操作，判断出是/dev/fb 设备文件之后，调用 fb\_write()函数中的 drwa\_rect 部分，执行 IOE 中\_io\_write 函数，进一步执行 out 指令，将数据输送到 vga 设备中达到更新屏幕的目的。

## 3 设计总结与心得

### 3.1 课设总结

该课设实现了一个小的虚拟机，虚拟机主要可以拆分为两个部分，底层的指令部分和操作系统部分，而在这个虚拟机中引入了 AM 的概念，进行了进一步的层次抽象，总的来说，AM 应该属于操作系统的概念，不过是提供更底层的接口，是与底层架构相关的部分，而更高层的操作系统则可以独立于架构。

### 3.2 课设心得

这次课设还是收获了很多，从如何阅读文档，到系统的方方面面，可能 pa2 的指令实现有点复杂，bug 也出现的最多，最后实现的操作系统离可用还差很远，但是正如手册中所说的，先跑起来再完美，PA 实验正是采用循序渐进的方法来进行的。

希望测试样例能更全面点，特别是 PA2 中的测试样例都是 32 位类型的数，用户在实现的时候很有可能会出现错误但是能通过所有的测试样例，导致之后出现难以察觉与找到的 bug。

## 参考文献

- [1] DAVID A. PATTERSON(美). 计算机组成与设计硬件/软件接口(原书第4版). 北京: 机械工业出版社.
- [2] David Money Harris(美). 数字设计和计算机体系结构(第二版). 机械工业出版社
- [3] 秦磊华, 吴非, 莫正坤. 计算机组成原理. 北京: 清华大学出版社, 2011 年.
- [4] 袁春风编著. 计算机组成与系统结构. 北京: 清华大学出版社, 2011 年.
- [5] 张晨曦, 王志英. 计算机系统结构. 高等教育出版社, 2008 年.