

华中科技大学

2021

系统能力综合训练 课程实验报告

题 目: RISC-V 模拟器开发

专 业: 计算机科学与技术

班 级: CS1906 班

学 号: U201915108

姓 名: 杨天元

电 话: 15527657001

邮 件: 1937176774@qq.com

完成日期: 2022-10-11



目 录

1 课程实验概述.....	1
1.1 课设目的	1
1.2 设计任务	1
1.3 设计要求	1
2 开天辟地的篇章：最简单的计算机.....	2
2.1 单步执行	2
2.2 打印程序状态	3
2.3 扫描内存	4
2.4 表达式求值	5
2.5 实现表达式生成器	9
2.6 设置监视点	10
2.7 删除监视点	11
2.8 必答题	12
2.9 小结	13
3 简单复杂的机器：冯诺依曼计算机系统.....	14
3.1 在 NEMU 中运行第一个 C 程序 dummy	14
3.2 实现指令，运行所有 cputest	16
3.3 输入输出	18
3.4 小结	21
4 穿越时空的旅程：批处理系统.....	23
4.1 实现自陷操作	23
4.2 实现系统调用	25
4.3 实现文件系统和虚拟文件系统	26
4.4 在 NEMU 中运行仙剑奇侠传	28
4.5 展示批处理系统	29

4.6 小结.....	30
5 总结.....	31
参考文献.....	32

1 课程实验概述

1.1 课设目的

计算机系统能力是每一个计算机学生都应该具备的基本能力。系统能力综合培养 课程旨在为学生设计一个综合性的系统设计任务。该课程强调对计算机系统软硬件综合系统的设计。

1.2 设计任务

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目 (Programming Assignment, PA)将提出x86/mips32/riscv32架构相应的教学版子集, 指导学生实现一个经过简化但功能完备的x86/mips32/riscv32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:

- 图灵机与简易调试器
- 冯诺依曼计算机系统
- 批处理系统
- 分时多任务
- 程序性能优化

1.3 设计要求

- (1) 实现简易调试器；表达式求值；监视点与断点
- (2) 运行第一个 C 程序；丰富指令集，测试所有程序；实现 I/O 指令，测试打字游戏
- (3) 实现系统调用；实现文件系统；运行仙剑奇侠传
- (4) 实现分页机制；实现进程上下文切换；时钟中断驱动的上下文切换

2 PA1-开天辟地的篇章: 最简单的计算机

在本节中，主要是实现一个简易调试器。包含已经实现的帮助，继续执行，退出功能，以及需要自己实现的单步执行，打印程序状态，表达式求值，扫描内存，设置监视点，删除监视点这些功能。基于这些基本功能，可以很方便的调试在模拟器中运行的程序，并对这些程序加以修正。

2.1 单步执行

单步执行就是逐条执行程序指令，通常用于调试程序。这个功能只要理解了已有代码，很容易就实现了。提取出参数之后，直接调用 `void cpu_exec(uint64_t n)` 这个函数即可。其中入参为 `cpu` 单步执行的次数。

```
static int cmd_si(char *args) {  
    /* extract the first argument */  
    char *arg = strtok(NULL, " ");  
    int n = 1; // 缺省为1  
  
    if (arg != NULL) {  
        n = atoi(arg);  
    }  
  
    // 单步执行n次  
    cpu_exec(n);  
  
    return 0;  
}
```

图 1 单步执行代码

```

(nemu) help
help - Display informations about all supported commands
c - Continue the execution of the program
q - Exit NEMU
si - 'si [N]'. Execute N instructions in a single step. The default value of N is 1
info - Print register state
p - 'p EXPR'. Evaluate the expression EXPR
x - 'x N EXPR'. Scanning memory
w - 'w EXPR'. Suspends program execution when the value of the expression EXPR changes
d - 'd [N]'. Delete the number watchpoint
(nemu) si
80100000: b7 02 00 80          lui  0x80000,t0
(nemu) si 2
80100004: 23 a0 02 00          sw   0(t0),$0
80100008: 03 a5 02 00          lw   0(t0),a0
(nemu) si 3
8010000c: 6b 00 00 00          nemu trap
nemu: HIT GOOD TRAP at pc = 0x8010000c

```

图 2 单步执行结果

2.2 打印程序状态

程序状态即为当前各个寄存器中的值。寄存器已经在框架中定义好了，相关的接口也已给出，因此直接循环调用即可。

```

void isa_reg_display() {
    for (int i = 0; i < 32; i++) {
        printf("%s\t\t\t%x\t\t\t%d\n", regsl[i], reg_l(i), reg_l(i));
    }
}

```

图 3 访问寄存器代码

```

(nemu) info r
$0          0          0
ra          0          0
sp          0          0
gp          0          0
tp          0          0
t0          0x80000000    -2147483648
t1          0          0
t2          0          0

```

图 4 访问寄存器结果

之后实现的打印监视点信息通过 `void watchpoint_display()` 函数实现，类似 `gdb` 中的输出所有监视点信息。

```

(nemu) w $t0
[src/monitor/debug/expr.c,90,make_token] match rules[8] =
Watchpoint 0: $t0
(nemu) c
[src/monitor/debug/expr.c,90,make_token] match rules[8] =
Watchpoint 0: $t0
Old value = 0
New value = 2147483648

(nemu) info w
Num      Type      What
0        watchpoint $t0
          breakpoint already hit 1 time

```

图 5 打印监视点信息

2.3 扫描内存

扫描内存需要先通过 `expr` 求出表达式表示的地址，然后循环调用 `paddr_read()` 函数显示出内存的结果即可。此处要注意一个问题，每次打印4个字节，即32位，而内存是以字节编址，因此pc的值应该+4而不是+32。

```

(nemu) x 5 0x80100000
[src/monitor/debug/expr.c,87,make_token] match rules[0] = "0x[0-9a-fA-F]+"
0x80100000:    0x800002b7    0x2a023 0x2a503 0x6b
0x80100010:    0

```

图 6 打印内存

```

static int cmd_x(char *args) {
    /* extract the argument */
    char *N = strtok(NULL, " ");
    char *e = strtok(NULL, "\n");
    if (N == NULL || e == NULL) {
        // 参数错误
        printf("A syntax error in expression\n");
    } else {
        int n = atoi(N);

        // 求出表达式的值
        bool success;
        paddr_t base_addr = expr(e, &success);
        if (!success) {
            printf("Error expression!\n");
            return 0;
        }

        // 输出结果
        int i;
        for (i = 0; i < n; i++) {
            if (i % 4 == 0) {
                if (i != 0) printf("\n");
                printf("%#x:\t", base_addr);
            }
            printf("%#x\t", paddr_read(base_addr, 4));
            base_addr += 4;
        }
        printf("\n");
    }
    return 0;
}

```

图 7 打印内存实现

2.4 表达式求值

表达式求值的框架代码已给出，主要难点在于算法问题。

词法分析部分，需要记录的只有立即数和寄存器，+、-、*、/等运算符号，均可通过 token_type 确定。


```

assert(nr_token < TOKENS_SIZE);
tokens[nr_token].type = rules[i].token_type;
switch (rules[i].token_type) {
    // 数字和寄存器 需要记录下来
    case TK_REG:
    case TK_HEXINT:
    case TK_DECINT:
        assert(substr_len < 32);
        strncpy(tokens[nr_token].str, substr_start, substr_len);
        tokens[nr_token].str[substr_len] = '\0';
        break;
    default: break;
}
if (rules[i].token_type != TK_NOTYPE) ++nr_token;

```

图 8 词法分析

之后是递归求值部分。参照给出的模板，当左右下标相同时，表示当前为数字或则寄存器类型，分情况处理即可。其中寄存器的值只需要找到相应的寄存器并取出值即可。

```

} else if (p == q) {
    // 单个token, 数字 或者是 寄存器
    switch (tokens[p].type) {
        case TK_DECINT: return atoi(tokens[p].str);
        case TK_HEXINT: return strtol(tokens[p].str, NULL, 16);
        case TK_REG:    return isa_reg_str2val(tokens[p].str + 1, success);
        default:        *success = false; return 0;
    }
}

```

图 9 token的处理

对于括号中的表达式，去掉括号继续运算即可，这可以通过 `check_parentheses` 函数判断。函数中先通过左右括号数以及左括号是否多于右括号来判断括号是否匹配，表达式是否合法，表达式如果不合法将 `success` 设为 `false` 之后返回。然后将左右括号去掉，再判断一遍，看表达式被一对匹配的括号包围着。具体代码如下：

```

bool check_parentheses(int p, int q, bool *success) {
    // 判断是不是Bad expression, 不包括()
    int cnt = 0;
    for (int i = p; i <= q; i++) {
        if (tokens[i].type == '(') cnt++;
        else if (tokens[i].type == ')') cnt--;
        if (cnt < 0) break;
    }
    if (cnt != 0) {
        *success = false;
        return false;
    }
    // 判断表达式是否被一对匹配的括号包围着
    if (tokens[p].type != '(' || tokens[q].type != ')') return false;
    cnt = 0;
    for (int i = p + 1; i < q; i++) {
        if (tokens[i].type == '(') cnt++;
        else if (tokens[i].type == ')') cnt--;
        if (cnt < 0) break;
    }
    return cnt == 0;
}

```

图 2.11 判断表达式是否被括号包围

再之后就是寻找主运算符将表达式分裂成两个子表达式处理。文档中已经介绍了具体的方法，主要就是根据运算符的优先级和结合性来查找。之后将左右两个子表达式求出结果然后根据主运算符计算出最后的结果即可。

2.5 实现表达式生成器

这部分的代码框架已经给出，只需要在框架中完补充函数实现细节即可。

```
static inline void gen_num() {
    gen_blank(); // 随机生成小于BLANK_NUM个空格
    uint32_t num = choose(100);
    char num_str[8] = {0};
    sprintf(num_str, "%u", num);
    if (buf_index + strlen(num_str) < BUF_SIZE)
        strncpy(buf + buf_index, num_str, strlen(num_str));
    else return ;
    buf_index += strlen(num_str);
    gen_blank();
}

static inline void gen_rand_op() {
    gen_blank();
    switch (choose(4)) {
        case 0: gen('+'); break;
        case 1: gen('-'); break;
        case 2: gen('*'); break;
        case 3: gen('/'); break;
        default: assert(0); break;
    }
    gen_blank();
}

static inline void gen_rand_expr() {
    switch (choose(3)) {
        case 0: gen_num(); break;
        case 1: gen('('); gen_rand_expr(); gen(')'); break;
        default: gen_rand_expr(); gen_rand_op(); gen_rand_expr(); break;
    }
}
```

图 2.13 生成表达式

2.6 设置监视点

首先需要实现watchpoint结构体。通过了解了 gdb 调试中的监视点，我新增加了三个字段，一个是用来保存表达式的字符串数组 e，然后是无符号数 old_value 用来保存上一次表达式的值，以及用于记录命中次数的 hit_times。之后就是通过 WP* new_wp(char* expression) 函数来创建一个监视点。创建和删除监视点的操作都比较简单，就是添加和删除链表节点。

之后在每执行一条指令时都会检查所有的已设置的监视点，当新的表达式的

值和旧值不同时就会触发监视点，输出相关信息。在同一时刻触发两个以上的监视点，会将所有的监视点输出。

```
(nemu) w $t0
[src/monitor/debug/expr.c,90,make_token] match rules[8] =
Watchpoint 0: $t0
(nemu) c
[src/monitor/debug/expr.c,90,make_token] match rules[8] =
Watchpoint 0: $t0
Old value = 0
New value = 2147483648

(nemu) info w
Num      Type      What
0        watchpoint $t0
         breakpoint already hit 1 time
```

图 2.14 设置监视点

2.7 必答题

1. 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?

$$500 * 90\% * (30 - 10) * 20 = 180000s = 50 \text{ 小时}$$

2. 我选择的ISA为RISCV-32.

3. riscv32 有哪几种指令格式?

文档中 23 页的 2.2 RV32I 指令格式小节中有介绍, RISC-V 指令具有六种基本指令格式:

R 类型指令: 用于寄存器 - 寄存器操作;

I 类型指令: 用于短立即数和访存 load 操作;

S 类型指令: 用于访存 store 操作;

B 类型指令: 用于条件跳转操作;

U 类型指令: 用于长立即数操作;

J 类型指令: 用于无条件操作;

4. LUI指令的行为是什么?

RISC-V-Reader-Chinese-v2p1.pdf 文档第 151 页。lui 高位立即数加载, 将符号位扩展的 20 位立即数 immediate 左移 12 位, 并将低 12 位置零, 写入 x[rd]中。

5. nemu/目录下的所有.c和.h和文件总共有多少行代码?

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

使用上述命令可以得到 PA1 之后代码行数为 5506, 然后切换到 PA0 分支 P 可以得到之前代码行数为 4968。

6. gcc 中的-Wall 和-Werror有什么作用? 为什么要使用-Wall 和-Werror?

都是编译器行为检错时使用的。-Wall 选项的含义是编译时报告所有的

Warning; -Werror 选项的含义是将所有的 Warning 都看待成 Error。这样做的好处是可以防止一切代码的不规范，保证代码在编译期是正确的，从而减少运行时因忽略 Warning 所导致的错误。

3 PA2-简单复杂的机器: 冯诺依曼计算机系统

本次实验的阶段安排如下:

task PA2.1: 在 NEMU 中运行第一个 C 程序 dummy

task PA2.2: 实现更多的指令, 在 NEMU 中运行所有 cputest task

PA2.3: 运行打字小游戏, 提交完整的实验报告

3.1 在 NEMU 中运行第一个C 程序 dummy

PA2最重要的就是阅读文档和手册。首先通在dummy的反汇编文件中找到尚未实现的指令: `li`, `auipc`, `addi`, `jal`, `mv`, `sw`, `jalr`。而通过文档我们知道, 为了实现一条新指令, 只需要在 `opcode_table` 中填写正确的译码辅助函数, 执行辅助函数以及操作数宽度; 之后用 RTL 实现正确的执行辅助函数, 需要注意使用 RTL 伪指令时要遵守上文提到的小型调用约定。

但是这部分代码相对比较晦涩, 其中使用了大量的宏定义来代替函数调用, 这样虽然使得代码更加精简, 但对于不熟悉c语言的人来说, 在阅读代码时会比较困难。

其中 `li`, `mv` 和 `addi` 指令在 有相同的`op_code`, 但通过阅读文档就可以知道他们的处理方式其实是相同的, 不过在执行辅助函数中还是需要通过 `rs1` 和 `simm11_0` 来判断他们是那一个指令, 从而在调试的时候可以正确输出指令信息。

```

make_EHelper(i) {
    switch (decinfo.isa.instr.funct3) {
        case 0: // addi
            rtl_addi(&id_dest->val, &id_src->val, decinfo.isa.instr.simm11_0);
            if (decinfo.isa.instr.rs1 == 0) { // li
                print_asm_template2(li);
            } else if (decinfo.isa.instr.simm11_0 == 0) { // mv
                print_asm_template2(mv);
            } else { // addi
                print_asm_template3(addi);
            }
            break;
        default:
            assert(0 && "Unfind the opcode");
            break;
    }
    rtl_sr(id_dest->reg, &id_dest->val, 4);
}

```

图 10 I 型指令执行辅助函数

auipc 指令与 lui 指令类似，可以参考其实现。只是在执行执行辅助函数中有所不同，auipc 指令需要先将立即数和 pc 值相加，再将结果存入目的寄存器。

sw 框架中已经实现。

jal 和 jalr 指令类似，阅读手册进行具体处理即可。

```

// jal
make_DHelper(JAL) {
    decode_op_i(id_src, (decinfo.isa.instr.simm20<<20) +
                    (decinfo.isa.instr.imm19_12<<12) +
                    (decinfo.isa.instr.imm11_1<<11) +
                    (decinfo.isa.instr.imm10_1<<1), true);
    decode_op_r(id_dest, decinfo.isa.instr.rd, false);

    print_Dop(id_src->str, OP_STR_SIZE, "0x%x", s0+cpu.pc);
}

// jalr
make_DHelper(JALR) {
    decode_op_r(id_src, decinfo.isa.instr.rs1, true);
    decode_op_i(id_src2, decinfo.isa.instr.simm11_0, true);
    decode_op_r(id_dest, decinfo.isa.instr.rd, false);

    print_Dop(id_src->str, OP_STR_SIZE, "%d(%s)", id_src2->val, reg_name(id_src->reg, 4));
}

```

图 11 跳转指令译码辅助函数

值得一提的是，文件中的最后的 return 表示的是 ret 指令。ret 指令其实是一条伪指令，实际调用的是 jalr 指令。

其中实现的所有辅助函数都需要在 decode.h 和 all-instr.h 中申明。最后结果也与预期相同。

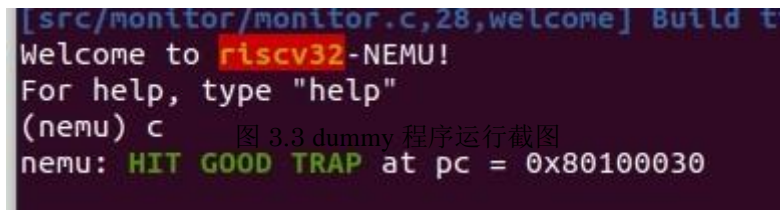


图 3.3 dummy 程序运行截图

3.2 实现指令，运行所有cputest

有了前面的铺垫，剩下的指令都大致相同了，但是需要实现的指令很多，有些指令又很相似，一不小心就会出错，然后就是漫长的调试发现解决问题的过程，这部分需要的时间很多很多。虽然不是很复杂，但是很繁琐。

```
make_EHelper(ld) {
    rtl_lm(&s0, &id_src->addr, decinfo.width);
    switch (decinfo.width) {
        case 4: print_asm_template2(lw); break; // lw
        case 2: // lhu和lh的区别就在于funct3的第三位
            if (decinfo.isa.instr.funct3 >> 2) { // lhu
                print_asm_template2(lhu);
            } else { // lh
                rtl_sext(&s0, &s0, 2);
                print_asm_template2(lh);
            }
            break;
        case 1: // lbu和lb的区别就在于funct3的第三位
            if (decinfo.isa.instr.funct3 >> 2) { // lbu
                print_asm_template2(lbu);
            } else { // lb
                rtl_sext(&s0, &s0, 1);
                print_asm_template2(lb);
            }
            break;
        default: assert(0);
    }
    rtl_sr(id_dest->reg, &s0, 4);
}
```

图 12 指令执行辅助函数

最后要实现字符串处理函数。这部分内容在完成了其他的文件测试后，还是比较简单的。都是比较常见的函数，功能实现起来也不复杂，这部分完成的比较顺利。

实现 `sprintf`。这个函数的实现一开始是真的无从下手，困难点就在于怎显示出来，如何格式化输出的可变参数如何获取，以及输出数据的格式化处理。但随着对 `vsprintf` 函数和 `va_list` 结构的了解后，慢慢有了思路。可变参数可以通过 `va_list` 结构体的相关操作来进行处理，从而获得每一个参数；然后对参数进行相应的格式化处理，最后可以通过调用已有的 `_putc` 函数一个一个的显示结果。其中格式化处理数据主要是在 `vsprintf` 函数中体现，主要是通过%后面带的参数来对数据进行不同的处理，包括输出宽度，对齐方式，填充字符以及输出形式等。有了思路，这部分实现起来就并不困难了，最后这部分也顺利通过了测试。

```
hust@hust-desktop:~/Desktop/lcs2019/nemu$ bash runall.sh ISA=riscv32
compiling NEMU...
Building riscv32-nemu
+ CC src/monitor/cpu-exec.c
+ CC src/isa/riscv32/diff-test.c
+ LD build/riscv32-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 13 程序测试结果

3.3 输入输出

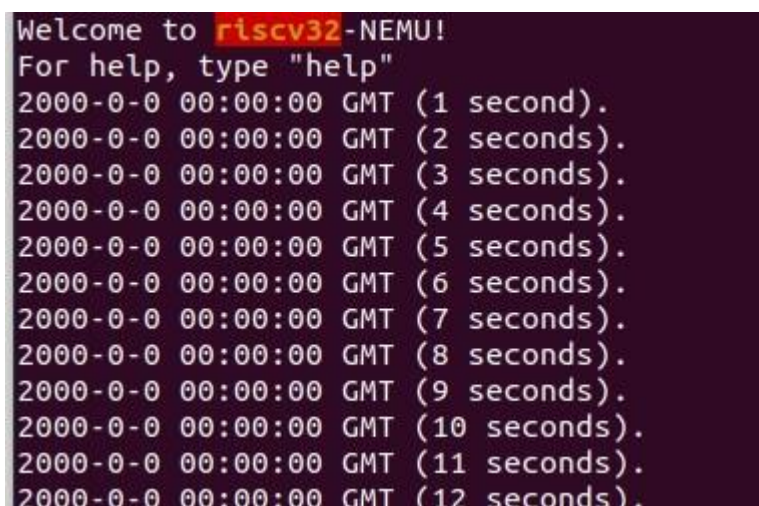
这部分实验总体来说比较简单，主要是对输入输出设备的理解以及相关 API 的调用。

(1) 运行 Hello World。

定义宏 HAS_IOE之后，因为我使用的是riscv指令集，因此不需要做其他的工作，可以直接运行。

(2) 实现_DEVREG_TIMER_UPTIME 的功能

需要先在初始化函数_am_timer_init 中将 boot_time 初始化，然后在 _DEVREG_TIMER_UPTIME 这个case 中将uptime->lo 赋值为当前时间和初始时间的差值。

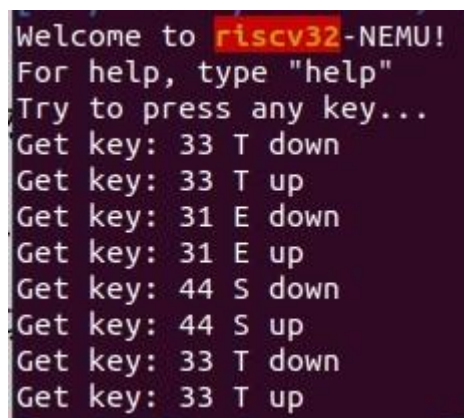


```
Welcome to riscv32-NEMU!  
For help, type "help"  
2000-0-0 00:00:00 GMT (1 second).  
2000-0-0 00:00:00 GMT (2 seconds).  
2000-0-0 00:00:00 GMT (3 seconds).  
2000-0-0 00:00:00 GMT (4 seconds).  
2000-0-0 00:00:00 GMT (5 seconds).  
2000-0-0 00:00:00 GMT (6 seconds).  
2000-0-0 00:00:00 GMT (7 seconds).  
2000-0-0 00:00:00 GMT (8 seconds).  
2000-0-0 00:00:00 GMT (9 seconds).  
2000-0-0 00:00:00 GMT (10 seconds).  
2000-0-0 00:00:00 GMT (11 seconds).  
2000-0-0 00:00:00 GMT (12 seconds).
```

图 14 real-time测试结果

(3) 实现_DEVREG_INPUT_KBD 的功能

先通过 `inl(KBD_ADDR)` 从 MMIO 中获取键盘码，然后通过键盘码和 `KEYDOWN_MASK` 相与得到是否为键盘按下的状态，通过键盘码和 `~KEYDOWN_MASK` 相与得到没有按键时的状态。



```
Welcome to riscv32-NEMU!  
For help, type "help"  
Try to press any key...  
Get key: 33 T down  
Get key: 33 T up  
Get key: 31 E down  
Get key: 31 E up  
Get key: 44 S down  
Get key: 44 S up  
Get key: 33 T down  
Get key: 33 T up
```

图 15 readkey 测试截图

(4) 输出的颜色动画

_DEVREG_VIDEO_INFO 中通过 inl(SCREEN_ADDR)就可以获得屏幕的宽和高信息，分别为高十六位和第十六位。

_DEVREG_VIDEO_FBCTL 中，通过结构体指针 ctl 可以得到要绘制矩形的坐标和长宽，以及图像像素信息 pixels。再通过 screen_width 和 screen_height 获取屏幕的宽和高，之后就可以一行一行地将 pixels 信息拷贝到 video memory 的 MMIO 空间。然后运行以下可以找到未实现的 vga_io_handler 函数，函数中在需要的时候更新一下屏幕就好。最终也成功运行了 amtest 中的 display test 测试。可以看到窗口中颜色条在逆时针旋转。

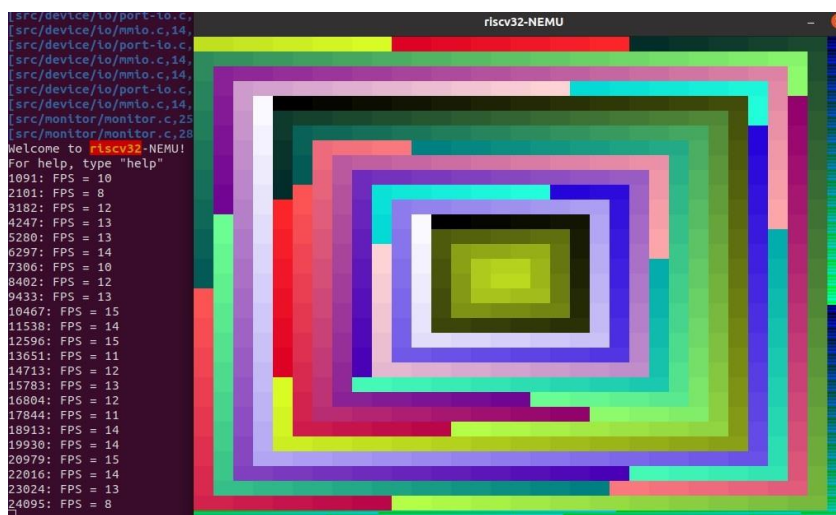


图 16 display 测试截图

(5) 运行打字小游戏

相关代码已在上述实验完成，这部分直接运行即可。



图 17 打字小游戏运行截图

3.4 小结

在本次 PA2 中，所有相关指令均已实现，并在 NEMU 中成功通过了所有 cputest 的测试，完成了输入输出相关的代码，成功运行了打字小游戏。

这部分实验大部分时间都花在了定位bug与修复bug上，指令的编写倒不是特别的复杂。只不过这些指令又特别多特别相似，有一点错误就会调试半天。每次遇到了不能一眼看出来的错误，只能够一步步地用调试器去检查。这个时候就体现出基础设施 DiffTest 的强大，可以不会再有因为找不到哪里的错误而苦恼。

同时，pa的框架代码也让我体会到C语言中宏定义的强大之处，虽然看上去比较晦涩，但却体现出了无与伦比的灵活性和丰富的功能。

4 PA3-穿越时空的旅程: 批处理系统

本次实验的阶段安排如下:

task PA3.1: 实现自陷操作`_yield()`及其过程

task PA3.2: 实现用户程序的加载和系统调用, 支撑 TRM 程序的运行

task PA3.3: 运行仙剑奇侠传并展示批处理系统, 提交完整的实验报告

4.1 实现自陷操作

(1) 实现新指令

这部分需要实现的新的指令有 `ecall` 环境调用指令, `sret` 管理员模式例外返回指令, 以及一系列的控制状态寄存器相关操作的指令。此处与PA2中的实现指令类似。

他们的指令格式相同, 所以只需要在 `opcode_table [28]`的位置设置为 `IDEX(SYSTEM, system)`, 然后实现相应的译码辅助函数和执行辅助函数即可。其中执行辅助函数部分实现实现如下图:

```
make_EHelper(system){
    Instr instr = decinfo.isa.instr;
    switch(instr.funct3){
        case 0b0:
            if (instr.val & ~(0x7f) == 0) { // ecall 环境调用
                raise_intr(reg_l(17), cpu.pc);
            } else if (instr.val == 0x10200073) { // sret 管理员模式例外
                decinfo.jump_pc = decinfo.isa.sepc + 4;
                rtl_j(decinfo.jump_pc);
            } else {
                assert(!"Undo the system opcode");
            }
            break;
        case 0b001: // csrrw
            s0 = read_csr(instr.csr);
            write_csr(instr.csr, id_src->val);
            rtl_sr(id_dest->reg, &s0, 4);
            print_asm_template3(csrrw);
            break;
    }
}
```

图 4.1 系统调用执行辅助函数

操作码相同的指令可根据 funct3 的不同来区分。其中 `ecall` 指令会调用 `raise_intr` 函数来完成触发异常后的响应过程；`sret` 指令会跳转到之前保存的指令的下一条指令；CSR 相关指令都是类似的，其中 `get_csr` 函数是根据参数返回具体的一个 csr 寄存器，`write_csr` 函数是将值写入到相应的 csr 寄存器中。

```
void raise_intr(uint32_t NO, vaddr_t epc) {
    /* TODO: Trigger an interrupt/exception with ``NO``.
     * That is, use ``NO`` to index the IDT.
     */
    decinfo.isa.sepc = epc;           // 将当前PC值保存到sepc寄存器
    decinfo.isa.scause = NO;         // 在scause寄存器中设置异常号
    decinfo.jmp_pc = decinfo.isa.stvec; // 从stvec寄存器中取出异常入口地址
    rtl_j(decinfo.jmp_pc);           // 跳转到异常入口地址
}
```

图 18 异常时的操作

(2) 重新组织_Context 结构体

根据 `trap.S` 中的代码可以得到 `_Context` 结构体中的成员顺序为 `gpr`, `cause`, `status`, `epc` 和 `as`。

(3) 实现事件分发和恢复上下文

先在 `_am_irq_handle()` 函数中通过异常号识别出自陷异常，然后将 `event` 设置为编号为 `_EVENT_YIELD` 的自陷事件。

```
_Event ev = {0};
switch (c->cause) {
    case -1: // 自陷异常
        ev.event = _EVENT_YIELD;
        break;
```

图 19 自陷异常设置

之后在 `do_event()` 函数中识别出自陷事件 `_EVENT_YIELD`，然后输出 “Self trap!” 即可。

```
static Context* do_event( Event e, Context* c) {
    switch (e.event) {
        case EVENT_YIELD:
            printf("Self trap!\n");
            break;
    }
}
```

图 20 自陷事件输出

重新运行 Nanos-lite, 可以成功看到在 do_event()中输出的“Self trap!”, 并且最后仍然触发了 main()函数末尾设置的 panic()。

```
Welcome to riscv32-NEMU!
For help, type "help"
(nemu) c
[/home/hust/Desktop/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/Desktop/ics2019/nanos-lite/src/main.c,15,main] Build time: 21:49:38, Jan 10 2022
[/home/hust/Desktop/ics2019/nanos-lite/src/randisk.c,28,init_randisk] randisk info: start = , end = , size = -2146427889 bytes
[/home/hust/Desktop/ics2019/nanos-lite/src/device.c,35,init_device] Initializing devices...
[/home/hust/Desktop/ics2019/nanos-lite/src/irq.c,20,init_irq] Initializing interrupt/exception handler...
[/home/hust/Desktop/ics2019/nanos-lite/src/proc.c,25,init_proc] Initializing processes...
[/home/hust/Desktop/ics2019/nanos-lite/src/main.c,33,main] Finish initialization
Self trap!
[/home/hust/Desktop/ics2019/nanos-lite/src/main.c,39,main] system panic: Should not reach here
nemu: HIT BAD TRAP at pc = 0x801006f0
```

图 21 自陷测试

4.2 实现系统调用

系统调用的实现和前面的实现自陷流程类似。先要在 _am_irq_handle 函数中新增当 c->cause 为 SYS_exit, SYS_yield, SYS_write, SYS_brk 等时将 event 设置为 _EVENT_SYSCALL。之后就可以在 do_event 函数中调用 do_syscall 函数, 完成相应的系统调用。

```
Context* do_syscall(Context *c) {
    uintptr_t a[4];
    a[0] = c->GPR1;
    a[1] = c->GPR2;
    a[2] = c->GPR3;
    a[3] = c->GPR4;

    switch (a[0]) {
        case SYS_yield:
            c->GPRx = sys_yield();
            break;
        case SYS_exit:
            sys_exit(a[1]);
            break;
        case SYS_write:
            c->GPRx = sys_write(a[1], (void*)(a[2]), a[3]);
            break;
        case SYS_brk:
            c->GPRx = sys_brk(a[1]);
            break;
        default:
            panic("Unhandled syscall ID = %d", a[0]);
    }

    return NULL;
}
```

图 22 增加新的系统调用

其中 $GPR1=gpr[17]$, $GPR2=gpr[10]$, $GPR3=gpr[11]$, $GPR4=gpr[12]$, $GPRx=gpr[10]$, 分别表示系统调用类型,系统调用的第 1, 2, 3 个参数以及系统调用的返回值。对于上述每一个系统调用在文档中都有具体的实现方法,按照上面实现即可,没有特别复杂的地方。最终可以看到将格式化完毕的字符串通过一次 `write()`系统调用进行输出。

```
Welcome to riscv32-NEMU!
For help, type "help"
[/home/hust/Desktop/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/Desktop/ics2019/nanos-lite/src/main.c,15,main] Build time: 09:55:12, Jan 11 2022
[/home/hust/Desktop/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start =
[/home/hust/Desktop/ics2019/nanos-lite/src/device.c,35,init_device] Initializing devices...
[/home/hust/Desktop/ics2019/nanos-lite/src/irq.c,20,init_irq] Initializing interrupt/excepti
[/home/hust/Desktop/ics2019/nanos-lite/src/proc.c,25,init_proc] Initializing processes..
[/home/hust/Desktop/ics2019/nanos-lite/src/loader.c,29,naive_uload] Jump to entry = 83000120
Hello World!
Hello World from Navy-apps for the 2th time!
Hello World from Navy-apps for the 3th time!
Hello World from Navy-apps for the 4th time!
Hello World from Navy-apps for the 5th time!
Hello World from Navy-apps for the 6th time!
```

图 23 hello 测试截图

4.3 实现文件系统和虚拟文件系统

(1) 实现完整的文件系统

这部分就是增加几个文件系统的系统调用, 需要实现`fs_open()`, `fs_read()`,`fs_close()`,`fs_write()`和`fs_lseek()`函数。这几个函数的实现比较简单, 每一个函数的实现在文档上都有相关描述, 跟着实现即可。但有一个需要注意的地方, 就是代码中并没有 `Finfo` 结构体中的 `open_offset` 字段, 需要自己增加, 并且修改相应的 `file_tables` 数组中的数据, 否则会有越界问题。

实现完成后, 运行测试程序`/bin/text`。这个测试程序用于进行一些简单的文件读写和定位操作。可以看到程序成功输出了 `PASS!!!`的信息。

```
Welcome to riscv32-NEMU!
For help, type "help"
[/home/hust/Desktop/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/Desktop/ics2019/nanos-lite/src/main.c,15,main] Build time: 16:08:59, Jan 11 2022
[/home/hust/Desktop/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = , end = , size = -2146423336 bytes
[/home/hust/Desktop/ics2019/nanos-lite/src/device.c,49,init_device] Initializing devices...
[/home/hust/Desktop/ics2019/nanos-lite/src/irq.c,19,init_irq] Initializing interrupt/exception handler...
[/home/hust/Desktop/ics2019/nanos-lite/src/proc.c,27,init_proc] Initializing processes...
[/home/hust/Desktop/ics2019/nanos-lite/src/fs.c,50,fs_open] open file : /bin/text
[/home/hust/Desktop/ics2019/nanos-lite/src/loader.c,49,naive_uload] Jump to entry = 830002f4
[/home/hust/Desktop/ics2019/nanos-lite/src/fs.c,50,fs_open] open file : /share/texts/num
PASS!!!
nemu: HLT GOOD TRAP at pc = 0x80100dc4
```

图 24 text 测试截图

(2) 把设备输入抽象成文件

需要实现 `events_read()` 函数，就是根据按键码来判断有没有按键事件，然后将相应的信息拷贝到 `buf` 数组中即可。

```
size_t events_read(void *buf, size_t offset, size_t len) {
    int keycode = read_key();
    if ((keycode & 0xffff) == _KEY_NONE) { // 没有按键事件
        len = sprintf(buf, "t %d\n", uptime());
    } else if (keycode & 0x8000) { // 按下按键
        len = sprintf(buf, "kd %s\n", keyname[keycode & 0xffff]);
    } else { // 松开按键
        len = sprintf(buf, "ku %s\n", keyname[keycode & 0xffff]);
    }
    return len;
}
```

图 25 事件读取函数实现

在 Nanos-lite 中运行 `/bin/events`，可以看到程序输出时间事件的信息，敲击按键时会输出对应的按键事件。

```
Start to receive events...
receive time event for the 1024th time: t 4030
receive time event for the 2048th time: t 5262
receive event: kd 0
receive event: ku 0
receive time event for the 3072th time: t 6422
receive event: kd K
receive event: ku K
receive time event for the 4096th time: t 7555
receive time event for the 5120th time: t 8680
receive time event for the 6144th time: t 9848
receive time event for the 7168th time: t 11049
receive event: kd LCTRL
receive event: ku LCTRL
```

图 26 events 函数测试

(3) 把 VGA 显存抽象成文件

这部分需要实现的内容以及如何实现都已在文档中给出。完成编码后，让 Nanos-lite 加载/bin/bmptest, 可以看到屏幕上显示出了 Project-N 的logo。

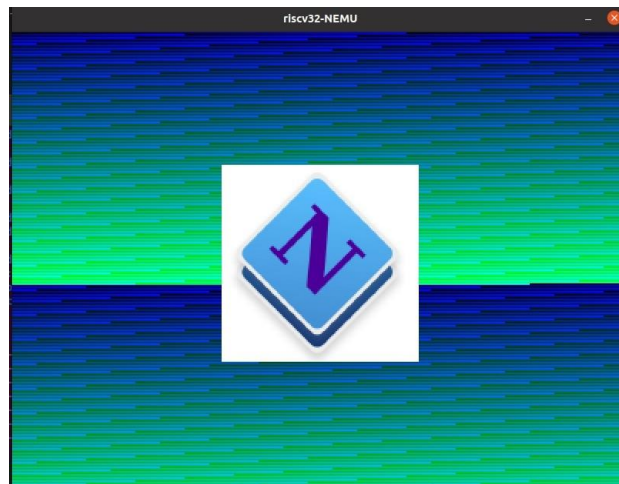


图 4.11 logo 显示测试

4.4 在 NEMU 中运行仙剑奇侠传

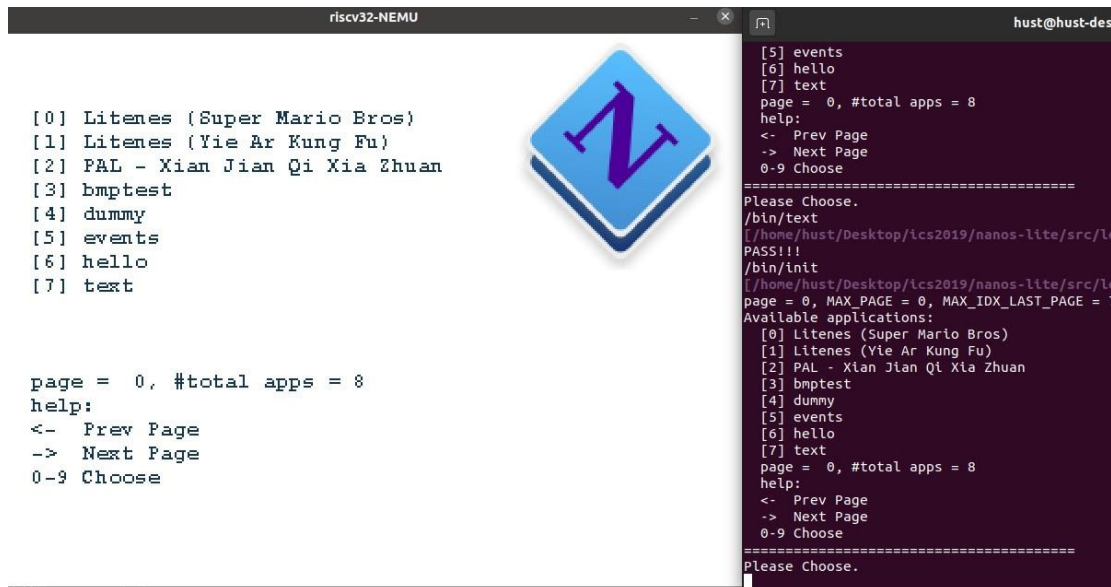
下载仙剑奇侠传的数据文件,并放到navy-apps/fsimg/share/games/pal/目录下,更新ramdisk之后,在Nanos-lite 中加载并运行/bin/pal。游戏运行截图如下。



图 27 仙剑奇侠传运行测试

4.5 展示批处理系统

接下来还需要在 VFS 中添加一个特殊文件 `/dev/tty`，只要让它往串口写入即可。再然后需要实现一个新的系统调用 `SYS_execve`，之后在 Nanos-lite 中加载 `/bin/init` 就可以运行它了。



```
riscv32-NEMU
[0] Litenes (Super Mario Bros)
[1] Litenes (Yie Ar Kung Fu)
[2] PAL - Xian Jian Qi Xia Zhuan
[3] bmptest
[4] dummy
[5] events
[6] hello
[7] text

page = 0, #total apps = 8
help:
<- Prev Page
-> Next Page
0-9 Choose

hust@hust-des
[5] events
[6] hello
[7] text
page = 0, #total apps = 8
help:
<- Prev Page
-> Next Page
0-9 Choose
=====
Please Choose.
/bin/text
[/home/hust/Desktop/ics2019/nanos-lite/src/lo
PASS!!!
/bin/init
[/home/hust/Desktop/ics2019/nanos-lite/src/lo
page = 0, MAX_PAGE = 0, MAX_IDX_LAST_PAGE = 7
Available applications:
[0] Litenes (Super Mario Bros)
[1] Litenes (Yie Ar Kung Fu)
[2] PAL - Xian Jian Qi Xia Zhuan
[3] bmptest
[4] dummy
[5] events
[6] hello
[7] text
page = 0, #total apps = 8
help:
<- Prev Page
-> Next Page
0-9 Choose
=====
Please Choose.
```

图 4.13 批处理系统展示

4.6 小结

在本次 PA3 中，我实现了自陷操作 `_yield()` 及其过程，实现了用户程序加载和系统调用，并实现了文件系统和 IOE 的抽象，最终成功运行了仙剑奇侠传游戏。完成了最终目标。这部分在代码方面并没有特别复杂的地方，整体上都是基于系统调用实现的，但是需要熟悉整个系统的结构与工作流程，因此需要熟读手册与文档拿爱国。通过本次实验，也让我对操作系统的系统调用的原理和实现有了更深的体会。

5 总结

在本次课程设计中，我基于设计好的模拟器代码框架，实现了一个简易调试器，完成了所需的所有 RISC-V 指令，实现 I/O 指令测试了打字小游戏，实现了系统调用和文件系统，最终在实现的模拟器上运行了游戏——仙剑奇侠传。

本次课程设计的难度还是比较大的，需要的时间也是很多的。对于 pa1 实现简易调试器，算是比较简单热身项目，主要还是对代码的熟悉。而 pa2 实现相关的指令这部分可能是这个实验最难最麻烦的了，因为这部分是最底层的，实现和调试起来都比较麻烦和抽象。特别是在充斥着各种晦涩不清宏定义的情况下，看懂代码都不容易。实现起来更是要有足够的细心和耐心，一旦有一点点错误，就要调试很久。而之后的输入输出以及 pa3，难度会稍微低一点，但是需要对整个系统的结构以及工作流程要有良好的理解。

参考文献

- [1] 张晨曦, 王志英. 计算机系统结构. 高等教育出版社, 2008 年.

电子签名:

A handwritten signature in black ink, appearing to be '张永元' (Zhang Yongyuan), written in a cursive style.