

华中科技大学

2022

系统能力培养 课程实验报告

题 目:	指令模拟器
专 业:	计算机科学与技术
班 级:	CS2003 班
学 号:	U202015360
姓 名:	胡沁心
电 话:	13656789548
邮 件:	13656789548@163. com
完成日期:	2023-01-16



目 录

1	课程实验概述	1
1.1	课设目的	1
1.2	课设任务	1
1.3	实验环境	1
2	实验方案设计	2
2.1	PA1 - 开天辟地的篇章: 最简单的计算机	2
PA1.1:	实现单步执行, 打印寄存器状态, 扫描内存	2
PA1.2:	实现表达式求值	2
PA1.3:	实现所有要求	3
2.2	PA2 - 简单复杂的机器: 冯诺依曼计算机系统	9
PA2.1:	在 NEMU 中运行第一个 C 程序 dummy	9
PA2.2:	实现更多的指令, 在 NEMU 中运行所有 cputest	11
PA2.3:	运行打字小游戏	14
2.3	PA3 - 穿越时空的旅程: 批处理系统	17
PA3.1:	实现自陷操作_yield() 及其过程	17
PA3.2:	实现用户程序的加载和系统调用, 支撑 TRM 程序的运行	18
PA3.3:	运行仙剑奇侠传并展示批处理系统	22
3	实验总结	24
	参考文献	25

1 课程实验概述

1.1 课设目的

探究“程序在计算机上运行”的机理，掌握计算机软硬协同的机制，进一步加深对计算机分层系统的理解，梳理大学 3 年所学的全部理论知识，提升计算机系统能力。

1.2 课设任务

在代码框架中实现一个简化的 RISC-V 模拟器，要求完成以下任务

- 可解释执行 RISC-V 执行代码

- 支持输入输出设备，支持异常流处理

- 支持精简操作系统---支持文件系统

- 持虚存管理

- 支持进程分时调度

- 最终在模拟器上运行“仙剑奇侠传”

1.3 实验环境

操作系统：Ubuntu 64 位 20.04.6 LTS

gcc 版本：8.1.0

2 实验方案设计

2.1 PA1 - 开天辟地的篇章: 最简单的计算机

PA1.1: 实现单步执行, 打印寄存器状态, 扫描内存

1. 实现单步执行

阅读文档, 得知执行指令的是 `cpu_exec` 这个函数, 参数为指令的条数。

模仿已有的指令函数, 在 `ui.c` 中写一个新函数调用 `cpu_exec(n)`, 并在菜单中添加 `si` 指令。

2. 打印寄存器状态

打开 `reg.c`, `riscv32` 的寄存器已经定义好了。我们需要完成函数 `isa_reg_display` 来打印寄存器状态, 循环输出名称和值就可以了。然后像上面一样, 在 `ui.c` 中加入新函数, 调用 `isa_reg_display`, 在菜单中加入 `info` 指令。

PA1.2: 实现表达式求值

阅读文档, 这个功能中需要能够识别十进制数字、十六进制数字、寄存器名, 支持加减乘除、与、或、异或、逻辑与、逻辑或、等于、不等于、小括号这些运算符。所以要在原来的基础上添加 `token` 类型和 `rules`。我实现的所有 `rules` 如下:

```
{ " ", TK_NOTYPE },
{ "0x[0-9a-fA-F]+", TK_HEX },
{ "[1-9][0-9]*|0", TK_DEC },
{ "\\$(0|[a|s|p|g|p|t|t[0-6]|s[0-9]|a[0-7])", TK_REG },
{ "==", TK_EQ },
{ "!=", TK_UNEQ },
{ "&&", TK_AND },
{ "\\|\\|\\|", TK_OR },
{ "&", '&' },
{ "\\|", '|' },
{ "\\+", '+' },
{ "\\-", '-' },
{ "\\*", '*' },
{ "/", '/' },
{ "\\(", '(' },
{ "\\)", ')' },
```

图 1-1

根据文档的要求, 还需要识别指针引用, 所以要加一个 `TK_DEREFERENCE` 类型的 `token`, 当 `*` 位于第二个运算符时, 识别为指针, 运算优先级为最高。

看 `make_token` 函数, 它用于识别 `token` 类型。数字和寄存器名需要被记录下来, 之后的运算过程中还要用。

然后开始实现表达式求值功能。文档给出了很详细的框架, 我们需要完成

eval 函数进行递归求值，以及 check_parentheses 函数判断是否有合法的括号。

```
bool check_parentheses(int p, int q, bool *success) {
    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;
    }
    return tokens[p].type == '(' && tokens[q].type == ')';
}
```

图 1-2

然后发现在 eval 函数中，有一个未完成的功能：判断表达式中运算符的优先级，得到主运算符在表达式中的位置。

在开始 switch 运算符之前，先处理表达式中的括号。在 eval 函数处理之后，主运算符一定不在括号内，所以直接排除括号内的表达式。

```
if(tokens[i].type == '('){
    cnt = 1;
    while (i <= q && cnt) {
        i++;
        if (tokens[i].type == '(')
            cnt++;
        else if (tokens[i].type == ')')
            cnt--;
    }
    i++;
}
```

图 1-3

然后根据优先级 switch 运算符。

```
case TK_OR:
    op = i;
    break;
case TK_AND:
    if (tokens[op].type != TK_OR)
        op = i;
    break;
case TK_EQ:
case TK_UNEQ:
    if (tokens[op].type != TK_AND && tokens[op].type != TK_OR)
        op = i;
    break;
case '+':
case '-':
    if (tokens[op].type != TK_AND
        && tokens[op].type != TK_OR
        && tokens[op].type != TK_EQ
        && tokens[op].type != TK_UNEQ)
        op = i;
    break;
```

图 1-4

PA1. 3: 实现所有要求

根据文档，需要实现这些功能：

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

图 1-5

1. 扫描内存

表达式求值功能已经实现，循环调用 `paddr_read()` 输出 N 个地址就可以了

2. 监视点操作

在监视点的结构体中添加两项，用于记录寄存器名和寄存器值

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

    /* TODO: Add more members if necessary */
    char e[32];
    uint32_t value;
} WP;
```

图 1-6

(1) 设置监视点

在 `watchpoint.c` 中完成函数 `new_wp` 来添加监视点。接下来就是很基础的链表操作：将新结点加到链表的最后。然后调用 `isa_reg_str2val` 函数获得寄存器值，将寄存器名和寄存器值写进结构体里。

在监视点的添加中有遇到一个问题：当输入 `w $t0` 时，是监视 `t0` 的值，如果我需要在 `t0` 中存储的数值处添加监视点，那么指令应该如何设计？

```

WP* new = free_;
free_ = free_->next;
new->next = NULL;
if(head == NULL)
    head = new;
else{
    WP* wp = head;
    while(wp){
        if(!wp->next){
            wp->next = new;
            break;
        }
        wp = wp->next;
    }
}
bool success = false;
uint32_t value = isa_reg_str2val(e+1, &success);
if (e[0]!='$' || !success) {
    printf("Error Register name!\n");
    return NULL;
}
strcpy(new->e, e);
new->value= value;

```

图 1-7

isa_reg_str2val 的实现如下。传 success 参数的时候不要忘记加&，找了好久才找到这个小错误。

```

if(!strcmp(s, "0\0")){
    *success = true;
    return 0;
}
for(int i=1;i<32;++i){
    if(!strcmp(s, regsl[i])){
        *success = true;
        return cpu.gpr[i]._32;
    }
}

```

图 1-8

(2) 删除监视点

也是基础的链表操作：找到指定 NO，从链表中把该结点移除，并放回 free_链表中

```

WP* wp = NULL;
if (head->NO == n) {
    wp = head;
    head = head->next;
    wp->next = free_;
    free_ = wp;
}
else {
    WP* pre = head;
    while (1) {
        if (pre->next == NULL) {
            printf("Didn't find number %u watchpoint\n", n);
            break;
        }
        if(pre->next->NO == n){
            wp = pre->next;
            pre->next = pre->next->next;
            wp->next = free_;
            free_ = wp;
            break;
        }
        pre = pre->next;
    }
}
}

```

图 1-9

(3) 打印监视点信息

依然是基础的链表操作，遍历，输出节点信息

(4) 检查监视点的值

根据文档的要求，需要实现在程序运行过程中，如果监视点的值发生变化就暂停程序的功能。修改 `cpu_exec` 函数，使每执行一步，都检查监视点的值是否与链表中的值一致，如果变化了，就暂停程序，打印监视点信息

```

if (check_watchpoint()) {
    nemu_state.state = NEMU_STOP;
    watchpoint_display();
    break;
}

```

图 1-10

检查函数的实现：遍历列表，与结构体中的值对比。

```

WP* wp = head;
bool flag = false;
while (wp) {
    bool success = false;
    uint32_t value = isa_reg_str2val(wp->e+1, &success);
    if(wp->value != value){
        flag = true;
        wp->value = value;
    }
    wp = wp->next;
}
return flag;

```

图 1-11

所有要求的运行结果如下：

1. 单步执行

```
(nemu) si 3
80100000: b7 02 00 80          lui    0x800000,t0
80100004: 23 a0 02 00          sw     0(t0),$0
80100008: 03 a5 02 00          lw     0(t0),a0
```

图 1-12

2. 打印寄存器状态

```
(nemu) info r
$0      0
ra      0
sp      0
gp      0
tp      0
t0      0x80000000
t1      0
t2      0
s0      0
s1      0
a0      0
a1      0
a2      0
a3      0
a4      0
a5      0
a6      0
a7      0
s2      0
s3      0
s4      0
s5      0
s6      0
s7      0
s8      0
s9      0
s10     0
s11     0
t3      0
t4      0
t5      0
t6      0
```

图 1-13

3. 表达式求值

```
(nemu) p 1+2*3-(3+4)
[src/monitor/debug/expr.c,90,make_token] match rules[2] = "[1-9][0-9]*|0" at position 0 with len 1: 1
[src/monitor/debug/expr.c,90,make_token] match rules[10] = "\"+\"" at position 1 with len 1: +
[src/monitor/debug/expr.c,90,make_token] match rules[2] = "[1-9][0-9]*|0" at position 2 with len 1: 2
[src/monitor/debug/expr.c,90,make_token] match rules[12] = "\"*" at position 3 with len 1: *
[src/monitor/debug/expr.c,90,make_token] match rules[2] = "[1-9][0-9]*|0" at position 4 with len 1: 3
[src/monitor/debug/expr.c,90,make_token] match rules[11] = "\"-" at position 5 with len 1: -
[src/monitor/debug/expr.c,90,make_token] match rules[14] = "\"(\" at position 6 with len 1: (
[src/monitor/debug/expr.c,90,make_token] match rules[2] = "[1-9][0-9]*|0" at position 7 with len 1: 3
[src/monitor/debug/expr.c,90,make_token] match rules[10] = "\"+\"" at position 8 with len 1: +
[src/monitor/debug/expr.c,90,make_token] match rules[2] = "[1-9][0-9]*|0" at position 9 with len 1: 4
[src/monitor/debug/expr.c,90,make_token] match rules[15] = "\"\"" at position 10 with len 1: )
0
```

图 1-14

4. 扫描内存

```

(nemu) x 10 0x80100000
[src/monitor/debug/expr.c,90,make_token] match rules[1] = "0x[0-9a-fA-F]"
0x80100000:      0x800002b7      0x2a023 0x2a503 0x6b
0x80100010:      0          0          0          0
0x80100020:      0          0

```

图 1-15

5. 监视点的添加、打印、检查、删除

```

(nemu) w $t0
Watchpoint 0: $t0
(nemu) w $t1
Watchpoint 1: $t1
(nemu) info w
Watchpoint 0: $t0 = 0
Watchpoint 1: $t1 = 0
(nemu) si 3
80100000:  b7 02 00 80                                lui  0x80000,t0
Watchpoint 0: $t0 = 2147483648
Watchpoint 1: $t1 = 0
(nemu) d 0
Watchpoint 0 have been removed
(nemu) info w
Watchpoint 1: $t1 = 0

```

图 1-16

2.2 PA2 – 简单复杂的机器：冯诺依曼计算机系统

这部分首先要做的是理解 riscv32 的各种指令的操作码和执行的原理。这些内容在 The RISC-V Reader 中都写得很清楚。然后就是阅读代码，尤其是 rtl 部分要熟悉每个函数，还有译码和执行过程中用到的各种结构体。以及最重要的：熟悉文件结构，PA2 部分需要理解和补充的代码比 PA1 更多也更分散。

PA2.1：在 NEMU 中运行第一个 C 程序 dummy

RISCV32 指令分为 R 型、I 型、S 型、B 型、U 型和 J 型六种类型。所有需要实现的指令的操作码都可以在 The RISC-V Reader 中找到。

先根据指令结构修改 opcode_table。其中 jalr 指令是 I 型指令，虽然 opcode 与其他 I 型不同，但译码部分的操作是相同的。

```
static OpcodeEntry opcode_table [32] = {
    /* b00 */ IDEX(ld, load), EMPTY, EMPTY, EMPTY, IDEX(I, I), IDEX(U, auipc), EMPTY, EMPTY,
    /* b01 */ IDEX(st, store), EMPTY, EMPTY, EMPTY, IDEX(R, R), IDEX(U, lui), EMPTY, EMPTY,
    /* b10 */ EMPTY, EMPTY, EMPTY, EMPTY, EMPTY, EMPTY, EMPTY, EMPTY,
    /* b11 */ IDEX(B, B), IDEX(I, jalr), EX(nemu_trap), IDEX(J, jal), EMPTY, EMPTY, EMPTY, EMPTY,
};
```

图 2-1

1. 译码

R 型、I 型、B 型和 J 型指令的译码辅助函数需要补充，仔细阅读 The RISC-V Reader 中的解释就可以完成。

```
make_DHelper(R) {
    decode_op_r(id_src, decinfo.isa.instr.rs1, true);
    decode_op_r(id_src2, decinfo.isa.instr.rs2, true);
    decode_op_r(id_dest, decinfo.isa.instr.rd, false);
}

make_DHelper(I) {
    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);
}

make_DHelper(B) {
    decode_op_r(id_src, decinfo.isa.instr.rs1, true);
    decode_op_r(id_src2, decinfo.isa.instr.rs2, true);
    s0 = (decinfo.isa.instr.simm12<<12)
        + (decinfo.isa.instr.imm11<<11)
        + (decinfo.isa.instr.imm10_5<<5)
        + (decinfo.isa.instr.imm4_1<<1);
    rtl_add(&decinfo.jmp_pc, &s0, &cpu.pc);
}

make_DHelper(J) {
    decode_op_i(id_src, (decinfo.isa.instr.simm20<<20)
        + (decinfo.isa.instr.imm19_12<<12) +
        + (decinfo.isa.instr.imm11<<11) +
        + (decinfo.isa.instr.imm10_1<<1), true);
    decode_op_r(id_dest, decinfo.isa.instr.rd, false);
}
```

图 2-2

2. 执行

R 型、I 型和 U 型指令为运算指令，执行辅助函数写在 `compute.c` 中。

查书可知 R 型指令操作码由在 `funct3` 和 `funct7` 区分。

```
int funct7 = (unsigned)decinfo.isa.instr.funct7;
switch(decinfo.isa.instr.funct3){
case 0:
    if (funct7 == 0) {           // add
        rtl_add(&id_dest->val, &id_src->val, &id_src2->val);
        print_asm_template3(add);
    } else if (funct7 == 0b01000000) { // sub
        rtl_sub(&id_dest->val, &id_src->val, &id_src2->val);
        print_asm_template3(sub);
    } else if (funct7 == 0b1) {      // mul
        rtl_mul_lo(&id_dest->val, &id_src->val, &id_src2->val);
        print_asm_template3(mul);
    } else {
        assert(0);
    }
break;
```

图 2-3

I 型指令主要由 `funct3` 区分，其中 `srli` 和 `shri` 还需要由 `funct7` 区分。对于 `addi`, `li`, `mv` 三个指令，查书得到：

mv `rd, rs1`

移动 (*Move*). 伪指令 (*Pseudoinstruction*), RV32I and RV64I.

把寄存器 `x[rs1]` 复制到 `x[rd]` 中。实际被扩展为 `addi rd, rs1, 0`

li `rd, immediate`

`x[rd] = immediate`

立即数加载 (*Load Immediate*). 伪指令 (*Pseudoinstruction*), RV32I and RV64I.

使用尽可能少的指令将常量加载到 `x[rd]` 中。在 RV32I 中，它等同于执行 `lui` 和/或 `addi`；对所以这三个指令一起处理。

`Jalr` 指令的操作码时和其他 I 型指令不同，执行函数要另外写

```
make_EHelper(jalr){
    s0 = cpu.pc + 4;
    rtl_sr(id_dest->reg, &s0, 4);
    rtl_add(&decinfo.jmp_pc, &id_src->val, &id_src2->val);
    rtl_j(decinfo.jmp_pc);
    print_asm_template2(jalr);
}
```

图 2-4

`ld` 和 `st` 指令已经在 `ldst.c` 中实现。

`auipc` 和 `lui` 类似。`lui` 已经实现，`auipc` 多了一个 `pc` 与立即数相加的过程。

B 型和 J 型指令为控制指令，执行辅助函数写在 `control.c` 中。仔细查书，这些函数都不难实现，但非常琐碎，需要熟悉 `rtl` 函数和结构体。

3. 伪指令

指令的详细实现在 `rtl.h` 中都已经完成，还需要完成伪指令部分。查看 `TODO`

的部分，发现注释其实已经基本帮我们写好了。

```
static inline void rtl_not(rtlreg_t *dest, const rtlreg_t* src1) {
    // dest <- ~src1
    // TODO();
    *dest = ~*src1;
}

static inline void rtl_sext(rtlreg_t* dest, const rtlreg_t* src1, int width) {
    // dest <- signext(src1[(width * 8 - 1) .. 0])
    // TODO();
    int32_t n = (4-width)*8, tmp = *src1;
    tmp = tmp << n;
    *dest = tmp >> n;
}

static inline void rtl_setrelopi(uint32_t relop, rtlreg_t *dest,
    const rtlreg_t *src1, int imm) {
    rtl_li(&ir, imm);
    rtl_setrelop(relop, dest, src1, &ir);
}

static inline void rtl_msb(rtlreg_t* dest, const rtlreg_t* src1, int width) {
    // dest <- src1[width * 8 - 1]
    // TODO();
    *dest = *src1 << ((width * 8 - 1) & 1);
}

static inline void rtl_mux(rtlreg_t* dest, const rtlreg_t* cond, const rtlreg_t* src1,
    const rtlreg_t* src2) {
    // dest <- (cond ? src1 : src2)
    // TODO();
    *dest = *cond ? *src1 : *src2;
}
```

图 2-5

PA2.2: 实现更多的指令，在 NEMU 中运行所有 cputest

指令基本都完成了。但运行检测时，发现 load-store fail 了。查看反汇编文件发现是 ld 指令出问题了。在执行辅助函数部分，虽然 ld 长得像完成了，其实没有完成。还需要区分 lhu 和 lh，lbu 和 lb，对于有符号非全字的数据，要将比最高位更高位的数据清除。rtl 中完成的 sext 伪指令部分在这里就有了作用。

```
make_EHelper(ld) {
    rtl_lm(&s0, &id_src->addr, decinfo.width);
    switch (decinfo.width) {
        case 4: print_asm_template2(lw); break;
        case 2:
            if (decinfo.isa.instr.funct3 >> 2) {
                print_asm_template2(lhu);
            } else {
                rtl_sext(&s0, &s0, 2);
                print_asm_template2(lh);
            }
            break;
        case 1:
            if (decinfo.isa.instr.funct3 >> 2) {
                print_asm_template2(lbu);
            } else {
                rtl_sext(&s0, &s0, 1);
                print_asm_template2(lb);
            }
            break;
    }
    rtl_sr(id_dest->reg, &s0, 4);
}
```

图 2-6

再阅读文档，还需要完成 `sprintf` 函数。`sprintf` 函数的主体部分是 `vsprintf` 函数，先完成这个函数。

判断%后面是否有 0，如果有，说明输出的数字需要加宽处理。如果是输出 d 类型，调用 `number` 函数来处理。如果是输出 s 类型，直接将参数拷贝到 `out` 字符串。调试中发现的问题：识别%之后不要忘记 `p++`；函数返回前不要忘记加结束符，因为检查的 `strcmp` 可能会非零。

```
int vsprintf(char *out, const char *fmt, va_list ap) {
    int len = 0;
    const char *p = fmt;
    int arg_num, neg=0, width=0;
    char *arg_s;
    while (*p) {
        if(*p != '%')
            out[len++] = *(p++);
        else {
            p++;
            if (*p == '0'){
                width = *(++p) - '0';
                p++;
            }
            switch (*p) {
                case 'd':
                    arg_num = va_arg(ap, int);
                    if (arg_num < 0) {
                        neg = 1;
                        arg_num = -arg_num;
                    }
                    len = add_number(out, len, arg_num, neg, width);
                    break;
                case 's':
                    arg_s = (char*) va_arg(ap, char*);
                    while (*arg_s) {
                        out[len++] = *(arg_s++);
                    }
                    break;
                default:
                    break;
            }
            p++;
        }
    }
    out[len] = '\\0';
    return len;
}
```

图 2-7

Number 函数中，数字要分正负数来处理：负数另外写进字符串。如果数字的宽度比长度大，在前面补 0。

```

int number(char *out, int len, unsigned int num, int neg, int width) {
    char str[STR_LEN] = {0};
    int i = 0;
    if(num == 0)
        str[i++] = '0';
    while (num) {
        str[i++] = num % 10 + '0';
        num /= 10;
    }
    int l=i;
    if (width > l){
        while(width != l){
            str[i++] = '0';
            width--;
        }
    }
    if (neg) {
        str[i++] = '-';
    }
    while (--i >= 0) {
        out[len++] = str[i];
    }
    return len;
}

```

图 2-8

先用 va_start 指针把参数列表从栈里取出来，最后要调用 va_end 置空指针。

```

int sprintf(char *out, const char *fmt, ...) {
    va_list ap;
    va_start(ap, fmt);
    int length = vsprintf(out, fmt, ap);
    va_end(ap);
    return length;
}

```

图 2-9

runall 运行结果:

```

superbug@ubuntu:~/Desktop/U202015360/nemu$ bash runall.sh ISA=riscv32
compiling NEMU...
Building riscv32-nemu
make: Nothing to be done for 'app'.
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 2-10

PA2. 3: 运行打字小游戏

这部分要实现输入输出，按照文档一步步来。先定义 HAS_IOE。然后写 printf 函数。printf 与 sprintf 非常像，只需要最后将字符串 buf 用 _putc 打印出来。

1. 时钟

在 __am_timer_init 函数中调用 inl(RTC_ADDR) 将 boot_time 初始化。

在 __am_timer_read 的 _DEVREG_TIMER_UPTIME case 中将 uptime->lo 改为当前时间和 boot_time 的差值。

```
case _DEVREG_TIMER_UPTIME: {
    _DEV_TIMER_UPTIME_t *uptime = (_DEV_TIMER_UPTIME_t *)buf;
    uint32_t past_time = inl(RTC_ADDR);
    uptime->hi = 0;
    uptime->lo = past_time - boot_time;
    return sizeof(_DEV_TIMER_UPTIME_t);
}
```

图 2-11

运行结果：

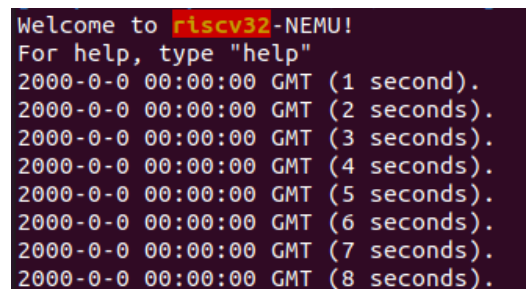


图 2-12

2. 键盘

调用 inl(KBD_ADDR) 获得键盘输入码。

通过 KEYDOWN_MASK 和键盘码获得按键按下和松开的状态。这部分的代码可以从 nemu/src/device/input.c 中参考。

```
uint32_t keyboard_code = inl(KBD_ADDR);
kbd->keydown = keyboard_code & KEYDOWN_MASK ? 1 : 0;
kbd->keycode = keyboard_code & ~KEYDOWN_MASK;
```

图 2-13

运行结果：

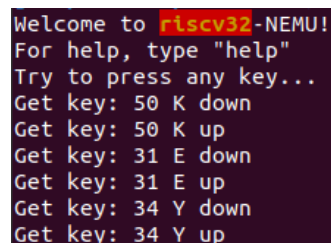


图 2-14

3. VGA

在 `__am_video_read` 函数的 `_DEVREG_VIDEO_INFO` case 中，通过 `inl(SCREEN_ADDR)` 就可以获得屏幕的宽和高信息，分别为高十六位和低十六位。

```
case _DEVREG_VIDEO_INFO: {
    _DEV_VIDEO_INFO_t *info = (_DEV_VIDEO_INFO_t *)buf;
    uint32_t screen_info = inl(SCREEN_ADDR);
    info->width = screen_info >> 16;
    info->height = screen_info & 0xffff;
    return sizeof(_DEV_VIDEO_INFO_t);
}
```

图 2-15

在 `__am_video_write` 函数的 `_DEVREG_VIDEO_FBCTL` case 中，通过结构体指针可以获得像素信息 `pixels` 和矩形的坐标和长宽。再通过提供的 `screen_width` 和 `screen_height` 函数获取屏幕的宽和高，然后逐行将 `pixels` 信息拷贝到 video memory 的 MMIO 空间。

```
uint32_t *pixels = ctl->pixels;
int x = ctl->x, y = ctl->y, w = ctl->w, h = ctl->h;
int W = screen_width(), H = screen_height();
int copy_bytes = sizeof(uint32_t) * (w < W - x ? w : W - x);
uint32_t *vmem = (uint32_t *) (uintptr_t) FB_ADDR;
for (int i = 0; i < h && y + i < H; i++, pixels += w)
    memcpy(&vmem[(y + i) * W + x], pixels, copy_bytes);
```

图 2-16

运行然后发现还有一个 `vga_io_handler` 函数没有实现，写完之后要更新屏幕。

运行结果：

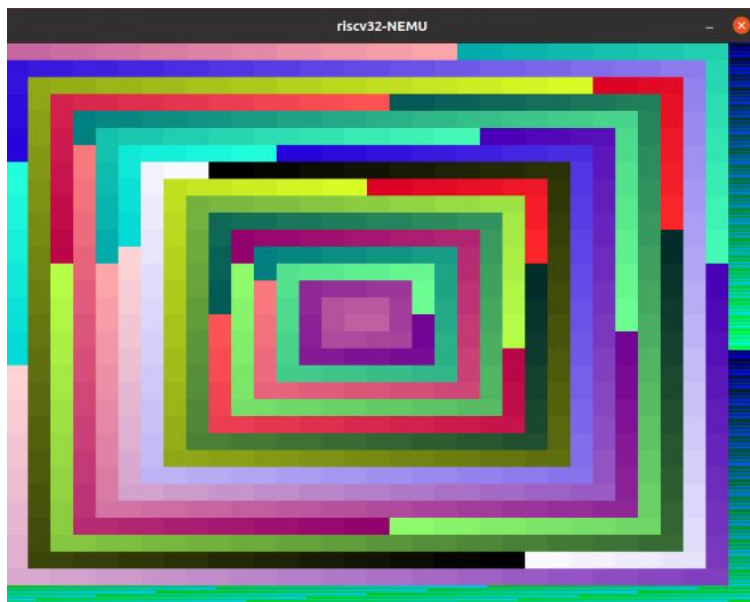


图 2-17

注意：PA1 中使用了 `debug` 功能，PA2 不要忘记在 `common.h` 中把 `DEBUG`

的定义注释掉关掉，不然程序会非常非常慢。

4. 打字游戏

运行结果：



图 2-18

红白机：



图 2-19

2.3 PA3 – 穿越时空的旅程：批处理系统

PA3.1: 实现自陷操作_yield() 及其过程

先阅读文档。在 RISC-V32 中实现系统指令，需要以下三个寄存器

- sepc寄存器 - 存放触发异常的PC
- sstatus寄存器 - 存放处理器的状态
- scause寄存器 - 存放触发异常的原因

把它们加到译码结构体中

```
struct ISADecodeInfo {  
    Instr instr;  
    uint32_t sepc;  
    uint32_t sstatus;  
    uint32_t scause;  
    uint32_t stvec;  
};
```

图 3-1

然后根据系统指令的操作码更新 opcode_table, 并实现 ecall 的译码和执行辅助函数。这与 PA2 的过程相似。执行函数根据 funct3 来区分指令。

开始实现自陷操作。根据文档的提是一步步来。先定义宏 HAS_CTE。

然后完成 raise_intr 函数来模拟异常相应机制：

riscv32触发异常后硬件的响应过程如下：

1. 将当前PC值保存到sepc寄存器
2. 在scause寄存器中设置异常号
3. 从stvec寄存器中取出异常入口地址
4. 跳转到异常入口地址

```
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;  
    decinfo.isa.scause = NO;  
    decinfo.jmp_pc = decinfo.isa.stvec;  
    rtl_j(decinfo.jmp_pc);  
}
```

图 3-2

重新组织_Context结构体。阅读 trap.S, 调整结构体成员的顺序为 gpr, cause, status, epc 和 as。

在__am_irq_handle 的 case 中识别出自陷。这部分的枚举常量已经在 am.h 中定义好了。最后在 do_event 的 case 里增加_EVENT_YIELD。

```
switch (c->cause) {
    case -1: // Self trap
        ev.event = _EVENT_YIELD;
        break;
```

图 3-3

```
switch (e.event) {
    case _EVENT_YIELD:
        printf("Self trap!\n");
        break;
```

图 3-4

运行结果：

```
Welcome to riscv32-NEMU!
For help, type "help"
[/home/superbug/Desktop/U202015360/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/superbug/Desktop/U202015360/nanos-lite/src/main.c,15,main] Build time: 06:55:37, Jan 12 2024
[/home/superbug/Desktop/U202015360/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = ,
e = -2146428571 bytes
[/home/superbug/Desktop/U202015360/nanos-lite/src/device.c,35,init_device] Initializing devices...
[/home/superbug/Desktop/U202015360/nanos-lite/src/irq.c,15,init_irq] Initializing interrupt/exception
[/home/superbug/Desktop/U202015360/nanos-lite/src/proc.c,25,init_proc] Initializing processes...
[/home/superbug/Desktop/U202015360/nanos-lite/src/main.c,33,main] Finish initialization
Self trap!
[/home/superbug/Desktop/U202015360/nanos-lite/src/main.c,39,main] system panic: Should not reach here
nemu: HIT BAD TRAP at pc = 0x8010054c
```

图 3-5

PA3. 2: 实现用户程序的加载和系统调用，支撑 TRM 程序的运行

1. 实现 loader

loader 的作用是把程序加载到指定内存位置，根据手册说明，从 elf 的 header 中提取出以下信息：

- 1) e_phoff: header 的偏移
- 2) e_phnum: header 的个数，接下来从 e_phoff 开始连续读 e_phnum 次。
- 3) p_type: 程序类型，值为 PT_LOAD 时加载这个 header 对应的 segment。
- 4) p_offset: header 对应的 segment 的偏移
- 5) p_vaddr: 程序拷贝到的目的地址
- 6) p_filesz: 文件大小
- 7) p_memsz: 内存大小

把 p_offset 开始的 p_filesz 个字节的程序拷贝到从 p_vaddr 开始的内存中，并把 p_vaddr+p_filesz 到 p_vaddr+p_memsz 结束的内存设置为 0。

这段程序偶尔读不到内容，没有找到是什么原因。

```

static uintptr_t loader(PCB *pcb, const char *filename) {
    //TODO();
    Elf_Ehdr Ehdr;
    ramdisk_read(&Ehdr, 0, sizeof(Ehdr));
    for(int i = 0; i < Ehdr.e_phnum; i++){
        Elf_Phdr Phdr;
        ramdisk_read(&Phdr, Ehdr.e_phoff + i*Ehdr.e_phentsize, sizeof(Phdr));
        if(Phdr.p_type == PT_LOAD){
            ramdisk_read((void*)Phdr.p_vaddr, Phdr.p_offset, Phdr.p_filesz);
            memset((void*)(Phdr.p_vaddr+Phdr.p_filesz),0,(Phdr.p_memsz-Phdr.p_filesz));
        }
    }
    return Ehdr.e_entry;
}

```

图 3-6

为什么需要将[VirtAddr + FileSiz, VirtAddr + MemSiz]对应的物理区间清零？

因为这里包括了.data，不清零在运行过程中访问数据段可能会出问题

运行结果：

```

Welcome to riscv32-NEMU!
For help, type "help"
[/home/superbug/Desktop/U202015360/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/superbug/Desktop/U202015360/nanos-lite/src/main.c,15,main] Build time: 07:07:08, Jan 12 2024
[/home/superbug/Desktop/U202015360/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = ,
e = -2146427467 bytes
[/home/superbug/Desktop/U202015360/nanos-lite/src/device.c,35,init_device] Initializing devices...
[/home/superbug/Desktop/U202015360/nanos-lite/src/irq.c,18,init_irq] Initializing interrupt/exceptio
[/home/superbug/Desktop/U202015360/nanos-lite/src/proc.c,25,init_proc] Initializing processes...
[/home/superbug/Desktop/U202015360/nanos-lite/src/loader.c,30,naive_uload] Jump to entry = 830000c8
Self trap!
nemu: HIT GOOD TRAP at pc = 0x80100848

```

图 3-7

2. 系统调用

先看 system.h 中定义的系统调用枚举常量

```

enum {
    SYS_exit,
    SYS_yield,
    SYS_open,
    SYS_read,
    SYS_write,
    SYS_kill,
    SYS_getpid,
    SYS_close,
    SYS_lseek,
    SYS_brk,
    SYS_fstat,
    SYS_time,
    SYS_signal,
    SYS_execve,
    SYS_fork,
    SYS_link,
    SYS_unlink,
    SYS_wait,
    SYS_times,
    SYS_gettimeofday
};

```

图 3-8

要能运行测试程序，我们需要完成 exit, yield, open, read, write, close, lseek, brk, execve。

像上一个任务一样，更新__am_irq_handle 和 do_event 函数。然后在 system.c

完成系统调用函数，在 nano.c 中更新用户函数。其中 sbrk 函数比较复杂，根据文档里的指导和 man 指令就可以了。对程序的结束位置做变化。实现如下

```
void *_sbrk(intptr_t increment) {
    extern intptr_t _end;
    static intptr_t program_break = (intptr_t)&_end;
    int old = program_break;
    if(!_syscall_(SYS_brk, program_break + increment, 0, 0)){
        program_break += increment;
        return (void *)old;
    }
    return (void *)-1;
}
```

图 3-9

注意：不要忘记 GPR 的定义需要修改！调了很久最后发现是这个没改。
这里也是看书，找到寄存器的函数调用规范。

寄存器	接口名称	描述	在调用中是否保留?
Register	ABI Name	Description	Preserved across call?
x0	zero	Hard-wired zero 硬编码 0	—
x1	ra	Return address 返回地址	No
x2	sp	Stack pointer 栈指针	Yes
x3	gp	Global pointer 全局指针	—
x4	tp	Thread pointer 线程指针	—
x5	t0	Temporary/alternate link register 临时寄存器	No /备用链接寄存器
x6-7	t1-2	Temporaries 临时寄存器	No
x8	s0/fp	Saved register/frame pointer 保存寄存器	Yes /帧指针
x9	s1	Saved register 保存寄存器	Yes
x10-11	a0-1	Function arguments/return values 函数参数	No /返回值
x12-17	a2-7	Function arguments 函数参数	No
x18-27	s2-11	Saved registers 保存寄存器	Yes
x28-31	t3-6	Temporaries 临时寄存器	No
f0-7	ft0-7	FP temporaries 浮点临时寄存器	No
f8-9	fs0-1	FP saved registers 浮点保存寄存器	Yes
f10-11	fa0-1	FP arguments/return values 浮点参数/返回值	No
f12-17	fa2-7	FP arguments 浮点参数	No
f18-27	fs2-11	FP saved registers 浮点保存寄存器	Yes
f28-31	ft8-11	FP temporaries 浮点临时寄存器	No

图 3-10

第一个 GPR 已经帮我们定义好了，其他 4 个 GPR 分别定义为 3 个函数参数和返回值。

```
#define GPR1 gpr[17]
#define GPR2 gpr[10]
#define GPR3 gpr[11]
#define GPR4 gpr[12]
#define GPRx gpr[10]
```

图 3-11

3. 文件系统

阅读文档，我们需要在 fs.c 中完成这几个文件函数。按照文档来写就行了


```

int fs_open(const char *pathname, int flags, int mode);
size_t fs_read(int fd, void *buf, size_t len);
size_t fs_write(int fd, const void *buf, size_t len);
size_t fs_lseek(int fd, size_t offset, int whence);
int fs_close(int fd);

```

由于文件的大小是固定的，在实现 fs_read(), fs_write()和 fs_lseek()的时候，注意偏移量不要越过文件的边界。

在 fs_write 中，需要对 stdout 和 stderr 做特殊处理，将字符串直接在命令行里输出，不能用 file_table 里的函数，不然会调用 invalid_write，程序会异常退出。

```

size_t fs_write(int fd, void *buf, size_t len){
    //Log("fd = %d write : %s", fd, (char*)buf);
    Finfo fp=file_table[fd];
    if(fp.size>0){
        int free = fp.size - fp.open_offset;
        assert(free >= 0);
    }
    int size;
    if(!strcmp(fp.name,"stdout")||!strcmp(fp.name,"stderr")){
        printf("%s",buf);
    }else if(fp.read){
        size = fp.write(buf, fp.open_offset, len);
        file_table[fd].open_offset += size;
        return size;
    }else{
        size = ramdisk_write(buf, fp.disk_offset + fp.open_offset,min(len,fp.size-1
        file_table[fd].open_offset += size;
        return size;
    }
}

```

图 3-13

然后更新 loader.c: 把 ramdisk_read 换成 fs_lseek 和 fs_read

```

static uintptr_t loader(PCB *pcb, const char *filename) {
    //TODO();
    Elf_Ehdr Ehdr;
    int fd = fs_open(filename, 0, 0);
    fs_lseek(fd, 0, SEEK_SET);
    fs_read(fd, &Ehdr, sizeof(Ehdr));
    for(int i = 0; i < Ehdr.e_phnum; i++){
        Elf_Phdr Phdr;
        fs_lseek(fd, Ehdr.e_phoff + i*Ehdr.e_phentsize, SEEK_SET);
        fs_read(fd, &Phdr, sizeof(Phdr));
        if(Phdr.p_type == PT_LOAD){
            fs_lseek(fd, Phdr.p_offset, SEEK_SET);
            fs_read(fd, (void*)Phdr.p_vaddr, Phdr.p_filesz);
            memset((void*)(Phdr.p_vaddr+Phdr.p_filesz),0,(Phdr.p_memsz-Phdr.p_filesz));
        }
    }
    fs_close(fd);
    return Ehdr.e_entry;
}

```

图 3-14

4. 虚拟文件系统

这个任务中需要把串口、设备和 vgc 都抽象成文件。

先阅读 device.c 代码，函数的名称和参数已经定义，在 file_table 中记录到对应位置。

```
static Finfo file_table[] __attribute__((used)) = {
    {"stdin", 0, 0, 0, invalid_read, invalid_write},
    {"stdout", 0, 0, 0, invalid_read, invalid_write},
    {"stderr", 0, 0, 0, invalid_read, invalid_write},
    {"/dev/events", 0, 0, 0, events_read, invalid_write},
    {"/dev/fb", 0, 0, 0, invalid_read, fb_write},
    {"/dev/fbsync", 0, 0, 0, invalid_read, fbsync_write},
    {"/proc/dispinfo", 0, 0, 0, dispinfo_read, invalid_write},
    {"/dev/tty", 0, 0, 0, invalid_read, serial_write},
#include "files.h"
};
```

图 3-15

然后完成 device.c 中的函数。都是比较简单的函数。其中 event_read 中要注意键盘有不按、按下、松开三种状态

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

图 3-16

init_device 和 init_fs 函数也需要补充。init_fs 中要对 vga 的内存大小进行初始化。对偏移量是否超过文件边界的判断对于 vga 的内存读写就起了作用。

PA3. 3: 运行仙剑奇侠传并展示批处理系统

运行结果：



图 3-17

最后我们要实现批处理系统。在 `sys_execve` 函数中调用 `init`, `sys_exit` 函数中从 `_halt` 改为 `sys_execve`, 就能实现这个循环。

```
int sys_execve(const char *fname, char * const argv[], char *const envp[]) {
    naive_uoload(NULL, fname);
}

void sys_exit(uintptr_t arg){
    sys_execve("/bin/init", NULL, NULL);
}
```

图 3-18

运行结果:

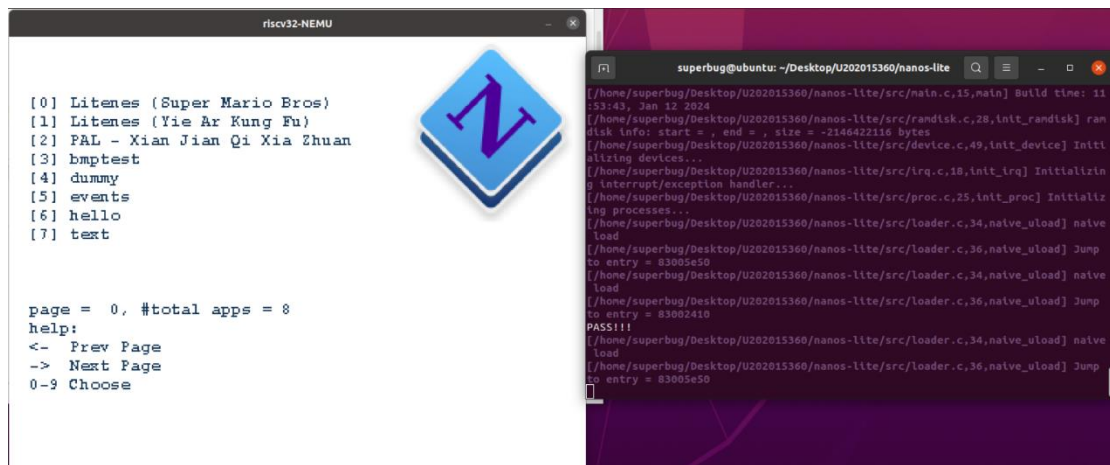


图 3-19

可以看到在开机界面中运行 `text` 后又重新回到了开机界面

3 实验总结

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，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。我觉得本次课程设计的难度很大，需要的时间精力也很多。九月份做完了 PA1，到了一月份才重启，PA1 已经忘得精光，又花了将近两个星期写 PA2 和 PA3，已经感觉筋疲力尽了。

虽然很难，但是文档写的很细致，给了很多提示。耐心看文档，每一个任务都可以慢慢解决。调试的过程中，有很多没注意到的问题在我重新仔细看了文档之后，发现其实都写了，只是我漏看了，刚开始经常调个半天还不如再看一遍文档。刚开始总是急着去实现功能，对文档和代码的不耐烦让我吃了点苦头。

代码的框架很复杂，熟悉代码与文件结构非常重要的，当然调 bug 也非常麻烦。文档中很贴心地给出了文件结构图，但我写完 PA1 都没当回事。到了 PA2，用到的文件结构变得复杂了，我经常翻来覆去找不到自己想找的文件，自己把自己气死。如果要理解一个完整的庞大的机制，对整体和细节的把握是同等重要的。阅读代码感受最深的是宏定义和指针的运用，这两个我认为是 c 语言最精妙的类型。这学期我在实习中看了很多 tmf 的驱动代码，也是运用了相当多的宏定义来简化代码，提高运行效率。

这次的课设综合性很强，内容也很丰富。我在完成的过程中经常觉得“这我之前写过差不多的”。比如 PA1 的表达式求值，我想到大二的程序设计课设，c 语言源程序处理工具中也要实现词法分析和语法分析。PA2 的 vga，我想到嵌入式系统课实现的香橙派触摸屏图像输出。PA3 的系统调用和用户函数，和操作系统课设中的很像。写课设的过程中，我在完成组原课设后又看了 The RISC-V Reader，觉得写得非常优雅。总之，这次课设完成的过程是对大学的知识进行了一次回顾，让我得到了书没白读的安慰。

这次课设进一步加深了我对计算机分层系统栈的理解，梳理了大学 3 年所学的全部理论知识，提升了我的计算机系统能力。同时我也发现原来已经学到了这么多东西，上次成功实现过代码的我这次能写得更好，感到很有成就感。

参考文献

- [1] RISC-V ABIs Specification, Version 0.01, 2021
- [2] DAVID PATTERSON, ANDREW WATERMAN, The RISC-V Reader, 2017
- [3] 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.
- [4] 张晨曦, 王志英. 计算机系统结构. 高等教育出版社, 2008 年