

华中科技大学

2023

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:	CS2011 班
学 号:	U202011675
姓 名:	徐锦慧
电 话:	18071680638
邮 件:	2939361916@qq.com
完成日期:	2023-12-17



目 录

1	课程实验概述	1
1.1	实验目的	1
1.2	实验要求	1
2	PA1: 最简单的计算机	3
2.1	方案设计	3
2.2	结果分析	10
2.3	实验必答题	12
2.4	心得体会	14
3	PA2: 冯诺依曼计算机系统	16
3.1	方案设计	16
3.2	结果分析	27
3.3	实验必答题	32
3.4	心得体会	33
4	PA3: 批处理系统	35
4.1	方案设计	35
4.2	结果分析	40
4.3	实验必答题	44
4.4	心得体会	45
	参考文献	46

1 课程实验概述

1.1 实验目的

计算机系统能力是指能自觉运用系统观,理解计算机系统的整体性、关联性、层次性、动态性和开放性,并用系统化方法,掌握计算机软硬件协同工作及相互作用机制的能力。系统能力包括系统分析能力、系统设计能力和系统验证及应用能力三个方面,三个方面相辅相成,共同构成计算机专业本科毕业生的基本能力和专业素养。

《系统能力培养综合实践之指令模拟器》就是为了培养计算机专业学生的系统能力而设置的。课程以软件实践为主,内容循序渐进、由浅入深,使学生能够快速入门,且帮助学生通过实践对理论、技术和方法进行巩固和理解,但是又具有高阶性、趣味性和挑战度,不断激发学生的兴趣和创造性;课程强化系统观,能够强化并检验学生的系统化综合能力;课程结合工程应用,能帮助学生理解计算机系统从底层硬件到高层软件的全套技术,使得学生对计算机系统各层次的技术有更加深刻的认知。

1.2 实验要求

1) 安装与配置开发工具链

基于主流版本的虚拟机平台与开源操作系统 Linux,安装基本的代码编辑、编译、调试、版本管理软件与重要插件,如 vim、gcc、gdb、git,并掌握这些软件的基本使用方法,为下一步的系统级开发奠定坚实的基础。

2) 构建基础编译环境

通过 git 工具下载模拟器软件,并根据文档完成基础开发环境的配置。通过此过程充分理解 Linux 系统中环境变量的功能与重要性,及其在 Makefile 中的常见使用方式。

3) 打造基础调试工具

开发一些必要的调试基础设施,如表达式求值功能、监视点、比较器,为解决开发过程中可能遇到的复杂软件缺陷做好准备。

4) 理解模拟器的基本原理

掌握使用一段程序模拟一条机器指令的基本方法，并能够在系统给出的基本框架下，结合指令集相关文档，实现 X86、MIPS 或 RISC-V 等众多主流指令集中的一个。

5) 构造基本运行时环境

理解运行时环境的基本概念及其在系统中的重要作用，并通过阅读文档，在系统给出的基本框架下，完成主要库函数的功能开发。

6) 构建基本输入输出环境。

理解计算机系统与输入输出设备交互的基本原理与方式方法，并依据文档要求，在系统中添加对于键盘、时钟、图形控制器等常见输入输出设备的支持。在完成上述目标后，系统即能具备较好的可交互性与可展示性。

7) 理解系统调用的基本原理与实现流程

了解计算机系统权限分级的原因与现状，理解自陷指令的功能与基本流程，理解系统调用的作用，并依据文档要求实现基本的系统调用。

8) 理解文件系统的基本工作原理

了解文件系统的主要功能，依据文档要求，实现一个简化的文件系统，支撑对复杂应用所包含的独立数据文件的访问，从而使得模拟器能够支持复杂应用。

9) 理解多任务系统的基本工作原理

了解多进程/任务并行的基本原理，了解虚存的基本原理及其对于多任务并行的重要性，了解分时功能的必要性，实现一个支持上述功能的简易多任务系统。

2 PA1：最简单的计算机

2.1 方案设计

PA 的框架代码由 4 个子项目构成: NEMU, Nexus-AM, Nanos-lite 和 Navy-apps, 它们共同组成了一个完整的计算机系统。而 PA1 只与 NEMU 子项目有关, 它主要由 4 个模块构成: monitor、CPU、memory 和设备。

NEMU 是一个用来执行其它客户程序的程序,可以随时了解客户程序执行的所有信息。为了提高调试的效率,同时也作为熟悉框架代码的练习,我们需要在 monitor 中实现一个简易调试器, 命令具体的格式和功能如表 2.1 所示:

表 2.1 调试器功能表

序号	命令	格式	示例	说明
1	帮助	help	help	打印命令的帮助信息
2	继续运行	c	c	继续运行被暂停的程序
3	退出	q	q	退出 NEMU
4	单步执行	si [N]	si 10	让程序单步执行 N 条指令后暂停
5	打印程序状态	info SUBCMD	info r	打印寄存器、监视点状态
6	表达式求值	p EXPR	p \$eax+1	求出表达式 EXPR 的值
7	扫描内存	x N EXPR	x 10 \$esp	输出以 EXPR 为起始内存地址的 4N 个字节
8	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
9	删除监视点	d N	d 2	删除序号为 N 的监视点

其中, 1、2、3 号功能系统已实现, 下面分别详细说明功能 4~9 的实现方案。

2.1.1 单步执行

单步执行命令的格式为 `si [N]`, 即让程序执行 N 条指令后暂停, 当 N 没有给出时, 默认为 1。

我们首先在 `cmd_table[]` 数组中添加 `cmd_si` 命令, 使其可以在 `help` 命令中被输出, 代码如下:

```
cmd_table [] = {  
    .....  
    { "si", "Pause after single-stepping N instructions, si [N]", cmd_si },  
}
```

接着在 `ui.c` 文件中实现 `cmd_si()` 函数。通过 `strtok()` 函数识别命令中的参数，并判断其是否为空。例如在 “`si n`” 的命令中识别出 `si` 和 `n`，从而得知这是一条单步执行 `n` 条指令的命令。再调用 `cpu_exec()` 函数并传入参数 `n`，让 CPU 向前执行 `n` 条指令。具体代码如下：

```
static int cmd_si(char *args) {
    /* extract the first argument */
    char *arg = strtok(NULL, " ");
    int n = 1; // default n = 1
    if (arg != NULL) {
        n = atoi(arg);
    }

    cpu_exec(n); // execute n instructions
    return 0;
}
```

2.1.2 打印程序状态

打印寄存器的命令为 `info SUBCMD`，即打印寄存器或监视点的信息。

同上，我们首先在 `cmd_table[]` 数组中添加 `cmd_info` 命令：

```
{ "info", "Print register status, info SUBCMD", cmd_info },
```

接着在 `ui.c` 文件中实现 `cmd_info()` 函数。通过 `strtok()` 函数识别命令中的参数，如果为 “`r`”，则调用 `isa_reg_display()` 输出寄存器信息；如果为 “`w`” 则调用 `wp_display()` 输出监视点信息；否则输出相应的报错信息。

`isa_reg_display()` 在 `nemu/src/isa/riscv32/reg.c` 文件中实现，用于打印寄存器的名字和存储的数值，代码如下：

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

`wp_display()` 在 `nemu/src/monitor/debug/watchpoint.c` 文件中实现，用于打印所有监视点的序号、参数和数值，代码如下：

```
void wp_display() {
    printf("NUM\tEXPR\tVAL\n");
    WP* wp = head;
    while(wp) {
        printf("%u\tt %s\tt %u\n", wp->NO, wp->exp, wp->val);
        wp = wp->next;
    }
}
```

2.1.3 表达式求值

表达式求值命令的格式为 `p EXPR`，即求出表达式 `EXPR` 的值。

首先进行词法分析。利用编译原理的相关知识补充表达式匹配规则，并在枚举类型中加上 `TK_DERE` 和 `TK_NEGA`，以解决含有解引用和负号的表达式。其中空格串的 `token` 类型是 `TK_NOTYPE`，不参加求值过程。具体匹配规则如下：

<code>{"\\+", '+'},</code>	<code>// plus</code>
<code>{"-", '-'},</code>	<code>// minus</code>
<code>{"*", '*'},</code>	<code>// multiply</code>
<code>{"/", '/'},</code>	<code>// divide</code>
<code>{"\\(", '('},</code>	<code>// left parenthesis</code>
<code>{"\\)", ')'},</code>	<code>// right parenthesis</code>
<code>{"(0x 0X)[0-9a-fA-F]+"</code>	<code>// hexadecimal integer</code>
<code>TK_HEXINT},</code>	
<code>{"[1-9][0-9]* 0",</code>	<code>// decimal integer</code>
<code>TK_DECINT},</code>	
<code>{" ", TK_NOTYPE},</code>	<code>// spaces</code>
<code>{"==", TK_EQ},</code>	<code>// equal</code>
<code>{"!=", TK_UEQ},</code>	<code>// unequal</code>
<code>{"&&", TK_LOGICAND},</code>	<code>// logic and</code>

给出正则表达式后，就可以在 `make_token()` 函数中识别 `token`。用 `position` 变量来指示当前处理到的位置，并按顺序尝试不同的规则来匹配当前字符串。若识别成功就用 `Log()` 输出配对信息，同时将匹配类型记录到 `tokens` 中。如果 `token` 类型是数字或寄存器名，还需要记录其值并判断缓冲区是否溢出。若尝试了所有规则都无法成功，则打印匹配失败的位置信息并退出。关键代码如下：

```
static bool make_token(char *e) {
    .....
    while (e[position] != '\0') {
        /* Try all rules one by one. */
        for (i = 0; i < NR_REGEX; i++) {
            if (regexec(&re[i], e + position, 1, &pmatch, 0) == 0 && pmatch.rm_so == 0) {
                char *substr_start = e + position;
                int substr_len = pmatch.rm_eo;
                Log(".....");
                position += substr_len;
                assert(nr_token < MAX_TOKSIZE); // prevent array subscript overflow
                tokens[nr_token].type = rules[i].token_type;
                if (rules[i].token_type == TK_DECINT || ..... ) {
                    assert(substr_len < 32); // check if buffer overflows
                    strncpy(tokens[nr_token].str, substr_start, substr_len);
                    tokens[nr_token].str[substr_len] = '\0';
                }
            }
            .....
        }
    }
}
```

接着在 `eval()` 函数中对表达式进行递归求值，用 BNF 的思想给出算数表达式

的归纳定义：

```
<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>
.....
```

为了在 token 表达式中指示一个子表达式，我们用整数 p 和 q 来指示这个子表达式的开始和结束位置。若 $p > q$ ，则标记错误信息并返回；若 $p = q$ ，说明表达式只有一个 token 且类型应为数字或寄存器名，如果类型符合返回 token 值，否则报错；若表达式被一对匹配的括号包围，就通过 $\text{eval}(p + 1, q - 1, \text{success})$ 继续递归。如果以上条件都不符合，则说明表达式需要分裂成操作符 op 和两个子表达式 $val1$ 、 $val2$ ，递归求出 $val1$ 、 $val2$ 后返回最终结果 $val1 \text{ op } val2$ 。 $\text{eval}()$ 函数的流程图如图 2.1 所示。

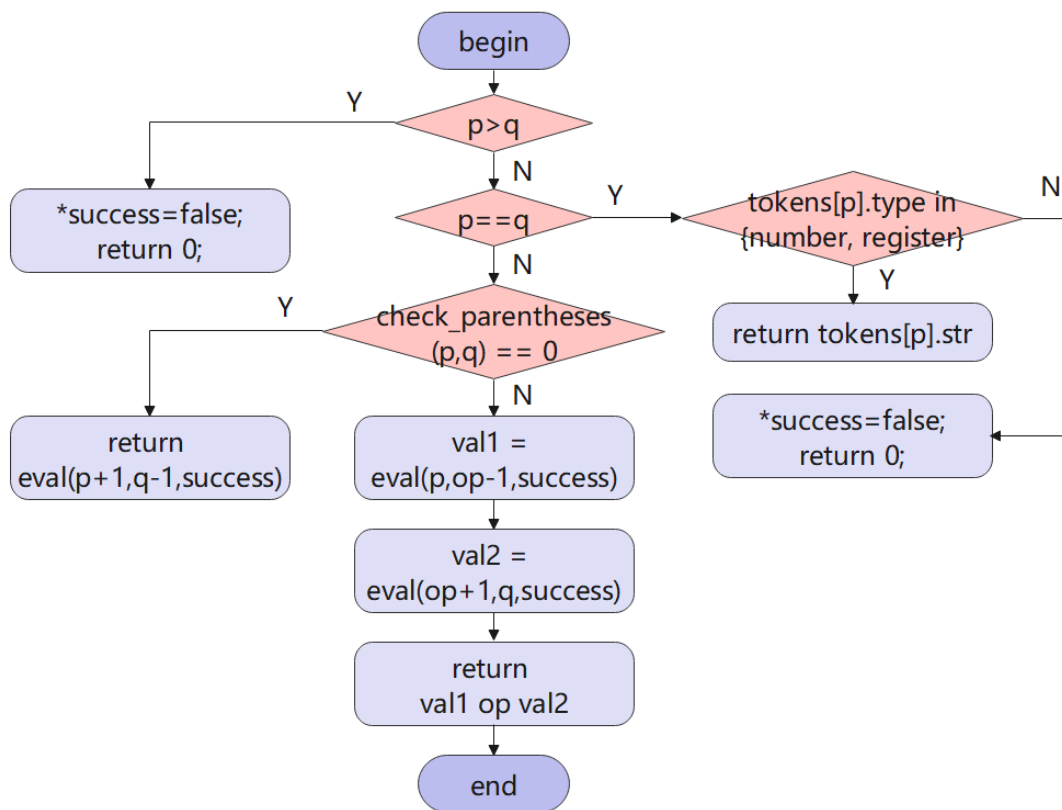


图 2.1 $\text{eval}()$ 函数流程图

其中 $\text{check_parentheses}()$ 函数用于判断表达式是否被一对匹配的括号包围，同时检查表达式中的左右括号是否匹配。如果不匹配，则说明表达式不符合语法，直接输出报错信息并返回即可。 $\text{check_parentheses}()$ 返回的情况共有 3 种：①表达式中左右括号不匹配；②表达式中左右括号匹配，但表达式不是由 “(” 和 “)”

包围着的；③表达式既被一对括号包围，又满足左右括号匹配。

最后，在 `expr()` 中整合以上函数，并补充 `cmd_help` 和 `cmd_p` 的系统调用，完成对表达式的求值。由于表达式中可能含有解引用和负号，我们需要在 `expr()` 中判断并标记。如果 `tokens[i]` 的类型为 “*” 或 “-”，且其前一个字符为 {+, -, *, /, { } 中的任意一个，则说明 “*” 不是乘号而是解引用，“-” 不是减号而是负号，从而正确计算出表达式。`expr()` 的代码如下：

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

    // check if expression contains dereference or negative symbol
    for(int i = 0; i < nr_token; i++) {
        if(tokens[i].type=='*' && (i==0 || get_priority(i-1)<5))
            tokens[i].type = TK_DERE;
        else if(tokens[i].type=='-' && (i==0 || get_priority(i-1)<5))
            tokens[i].type = TK_NEGA;
    }

    return eval(0, nr_token-1, success);
}
```

2.1.4 扫描内存

扫描内存的命令为 `x N EXPR`，即求出表达式 `EXPR` 的值，并将结果作为起始内存，以十六进制的形式输出连续的 `4N` 个字节。

同上，我们首先在 `cmd_table[]` 数组中添加 `cmd_x` 命令：

```
{ "x", "Output 4N bytes with EXPR as the starting address, x N EXPR", cmd_x },
```

接着在 `ui.c` 文件中实现 `cmd_x()` 函数。通过 `strtok()` 函数识别命令中的表达式 `e` 和字节数 `n`，接着用 2.1.3 中实现的 `expr()` 求出表达式的值，最后输出 `4n` 个字节即可。代码如下：

```
static int cmd_x(char *args) {
    /* extract the first argument */
    char *N = strtok(NULL, " "); char *e = strtok(NULL, "\n");
    if(N==NULL || e==NULL){ printf("Error expression!\n");return 0; }
    int n = atoi(N); bool success;
    paddr_t base_addr = expr(e, &success);
    if(!success){ printf("Error expression!\n");return 0; }

    for(int i=0; i<n; i++, base_addr+=4){
        if(i%4 == 0){
            if(i != 0) printf("\n");
            printf("%0#x\t:", base_addr);
        }
    }
}
```

```

    }
    printf("%#x\t", paddr_read(base_addr, 4));
}
printf("\n");
return 0;
}

```

2.1.5 设置监视点

监视点的功能主要是监视表达式的值，命令的格式为 `w EXPR`，当表达式 `EXPR` 的值变化时，需要暂停程序并输出相关信息。

为了发挥监视点的功能，我们需要扩展 2.1.3 中表达式求值的实现，即补充十六进制、寄存器、指针解引用的正则匹配规则：

```

<expr> ::= <decimal-number>
| <hexadecimal-number>      # 以"0x"开头
| <reg_name>                 # 以"$"开头
| "*" <expr>                 # 指针解引用
.....

```

由于调试器需要满足用户设置多个监视点的需求，我们需要使用链表组织监视点的信息。监视点的结构体如下：

```

typedef struct watchpoint {
    int NO;
    struct watchpoint *next;
    char exp[65535];
    uint32_t val;
} WP;

```

其中 `NO` 表示监视点的序号，`next` 表示指向下一个监视点的指针，`exp` 表示监视的表达式，`val` 表示监视点当前的值。在 `nemu/src/monitor/debug/watchpoint.c` 文件中，我们用 `wp_pool` 存储所有的监视点，`head` 指针用于组织正在使用中的监视点，`free` 指针用于组织空闲的监视点，并通过 `init_wp_pool()` 对以上结构进行初始化。

为了实现创建监视点的功能，我们需要补充 `new_wp()` 函数。若 `free` 链表中有空闲的监视点，则从中取出第一个插入 `head` 链表并返回该监视点，否则调用 `assert(0)` 终止程序。

为了检查监视点的变化情况，我们需要补充 `check_wp()` 函数。首先遍历 `head` 链表，用 `expr()` 求出当前监视点的新值，并与现有值进行比较。若表达式的值变化，则需要输出相关信息，并更新监视点的值。`check_wp()` 的代码如下：

```

bool check_wp() {
    WP* wp = head; bool flag = false;
    while(wp) {
        bool success;
        uint32_t new_val = expr(wp->exp, &success);
        assert(success);
        if(new_val != wp->val){
            printf("Watchpoint NO.%u has changed:\n", wp->NO);
            printf("expression = %s, old value = %u, new value = %u\n", wp->exp,
wp->val, new_val);
            wp->val = new_val, flag = true;
        }
        wp = wp->next;
    }
    return flag;
}

```

完成以上函数后，就可以实现对监视点的管理。当用户给出一个待监视表达式时，系统就会通过 `new_wp()` 申请一个空闲的监视点结构，并将表达式记录下来。每当 `cpu_exec()` 执行完一条指令，就调用 `check_wp()` 来比较它们的值有没有发生变化，若发生变化，程序就会因触发监视点而暂停，通过将 `nemu_state.state` 变量设为 `NEMU_STOP` 来实现。最后输出相关提示，并返回到 `ui_mainloop()` 循环中等待用户的命令。

2.1.6 删除监视点

与 2.1.5 类似，删除监视点的命令为 `d N`，即取消监视序号为 `N` 的监视点，由函数 `free_wp()` 实现。若链表 `head` 为空，说明没有需要释放的监视点，调用 `assert(0)` 并退出；否则遍历 `head`，找到序号为 `N` 的监视点，将其从 `head` 归还到 `free_` 链表中。注意由于链表的特殊性，需要额外考虑选中监视点为 `head` 的头结点的情况。代码如下：

```

void free_wp(uint32_t num) {
    if(head == NULL) { assert(0); return 0; }

    WP *wp = NULL, *tmp = head;
    if(head->NO != num){
        while(tmp->next->NO != num) {
            if(tmp->next == NULL){
                printf("Cannot get NO.%u watchpoint\n", num); return 0;
            }
            tmp = tmp->next;
        }
        tmp = tmp->next;
    }
}

```

```

}
wp = tmp; tmp = tmp->next; wp->next = free_; free_ = wp;
return 0;
}

```

2.2 结果分析

2.2.1 单步执行

实验结果如图 2.2 所示。输入 `help` 命令，系统能输出 `si` 命令的相应解释；输入 `si`，程序自动向前执行一条指令；输入 `si 1` 和 `si 3`，系统也能执行相应的步数。

```

Welcome to riscv32-NEMU!
For help, type "help"
(nemu) help
help - Display informations about all supported commands
c - Continue the execution of the program
q - Exit NEMU
si - Pause after single-stepping N instructions, si [N]
(nemu) si
80100000: b7 02 00 80          lui  0x80000,t0
(nemu) si 1
80100004: 23 a0 02 00          sw   0(t0),$0
(nemu) si 3
80100008: 03 a5 02 00          lw   0(t0),a0
8010000c: 6b 00 00 00          nemu trap
nemu: HIT GOOD TRAP at pc = 0x8010000c

[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 4

```

图 2.2 单步执行结果

2.2.2 打印程序状态

实验结果如图 2.3 所示。输入 `help` 命令，系统能输出 `info` 命令的相应解释；输入错误参数时，系统能打印相应的错误提示；输入 `info r`，系统也能正确打印寄存器的信息。

```

(nemu) info
Please input subcmd, 'r' or 'w'
(nemu) si 2
80100000: b7 02 00 80          lui  0x80000,t0
80100004: 23 a0 02 00          sw   0(t0),$0
(nemu) info r
$0      0          0
ra      0          0
sp      0          0
gp      0          0
tp      0          0
t0      0x80000000  -2147483648

```

图 2.3 打印寄存器结果

2.2.3 表达式求值

最初进行表达式求值的检验时，程序出现了如图 2.4 所示的编译错误。经检查发现，是“(”的正则式匹配部分有误，需要改用“\\(”的格式匹配。

```
+ CC src/isa/riscv32/init.c
+ LD build/riscv32-qemu-so
make[1]: Leaving directory '/home/hust/ics2019/nemu/tools/qemu-diff'
./build/riscv32-nemu -l ./build/nemu-log.txt -d /home/hust/ics2019/nemu/tools/qemu-diff/build/riscv32-qemu-so
[src/monitor/monitor.c,36,load_img] No image is given. Use the default build-in image.
[src/memory/memory.c,16,register_pmem] Add 'pmem' at [0x80000000, 0x87ffffff]
regex compilation failed: Unmatched ( or \(
(
riscv32-nemu: src/monitor/debug/expr.c:57: init_regex: Assertion `0' failed.
make: *** [Makefile:77: run] Aborted (core dumped)
```

图 2.4 表达式求值结果 1

修改词法分析部分后，程序能通过编译，但是在执行如图 2.5 所示的命令时出错，经检查发现，在存储表达式的数字时出现地址越界。

```
(nemu) p (5+4*3/2-1)
[src/monitor/debug/expr.c,84,make_token] match rules[4] = "(" at position 0 with len 1: (
[src/monitor/debug/expr.c,84,make_token] match rules[6] = "[1-9][0-9]*|0" at position 1 with len 1: 5
[src/monitor/debug/expr.c,84,make_token] match rules[0] = "+" at position 2 with len 1: +
[src/monitor/debug/expr.c,84,make_token] match rules[6] = "[1-9][0-9]*|0" at position 3 with len 1: 4
[src/monitor/debug/expr.c,84,make_token] match rules[2] = "*" at position 4 with len 1: *
[src/monitor/debug/expr.c,84,make_token] match rules[6] = "[1-9][0-9]*|0" at position 5 with len 1: 3
[src/monitor/debug/expr.c,84,make_token] match rules[3] = "/" at position 6 with len 1: /
[src/monitor/debug/expr.c,84,make_token] match rules[6] = "[1-9][0-9]*|0" at position 7 with len 1: 2
[src/monitor/debug/expr.c,84,make_token] match rules[1] = "-" at position 8 with len 1: -
[src/monitor/debug/expr.c,84,make_token] match rules[6] = "[1-9][0-9]*|0" at position 9 with len 1: 1
[src/monitor/debug/expr.c,84,make_token] match rules[5] = ")" at position 10 with len 1: )
address (0x00000003) is out of bound {???} [0x00000000, 0x00000000] at pc = 0x80100000
riscv32-nemu: src/device/io/map.c:20: check_bound: Assertion `map != ((void *)0) && addr <= map->high && addr >= map->low' failed.
make: *** [Makefile:77: run] Aborted (core dumped)
```

图 2.5 表达式求值结果 2

修改以上错误后，程序能正确执行，并且能够处理表达式中的空格和负号等情况，如图 2.6 所示。

```
(nemu) p (5 + 4 * 3 / 2 - 1)
[src/monitor/debug/expr.c,84,make_token] match rules[4] = "(" at position 0 with len 1: (
[src/monitor/debug/expr.c,84,make_token] match rules[6] = "[1-9][0-9]*|0" at position 1 with len 1: 5
[src/monitor/debug/expr.c,84,make_token] match rules[8] = " +" at position 2 with len 1:
[src/monitor/debug/expr.c,84,make_token] match rules[0] = "+" at position 3 with len 1: +
[src/monitor/debug/expr.c,84,make_token] match rules[8] = " +" at position 4 with len 1:
[src/monitor/debug/expr.c,84,make_token] match rules[6] = "[1-9][0-9]*|0" at position 5 with len 1: 4
[src/monitor/debug/expr.c,84,make_token] match rules[8] = " +" at position 6 with len 1:
[src/monitor/debug/expr.c,84,make_token] match rules[2] = "*" at position 7 with len 1: *
[src/monitor/debug/expr.c,84,make_token] match rules[8] = " +" at position 8 with len 1:
[src/monitor/debug/expr.c,84,make_token] match rules[6] = "[1-9][0-9]*|0" at position 9 with len 1: 3
[src/monitor/debug/expr.c,84,make_token] match rules[8] = " +" at position 10 with len 1:
[src/monitor/debug/expr.c,84,make_token] match rules[3] = "/" at position 11 with len 1: /
[src/monitor/debug/expr.c,84,make_token] match rules[8] = " +" at position 12 with len 1:
[src/monitor/debug/expr.c,84,make_token] match rules[6] = "[1-9][0-9]*|0" at position 13 with len 1: 2
[src/monitor/debug/expr.c,84,make_token] match rules[8] = " +" at position 14 with len 1:
[src/monitor/debug/expr.c,84,make_token] match rules[1] = "-" at position 15 with len 1: -
[src/monitor/debug/expr.c,84,make_token] match rules[8] = " +" at position 16 with len 1:
[src/monitor/debug/expr.c,84,make_token] match rules[6] = "[1-9][0-9]*|0" at position 17 with len 1: 1
[src/monitor/debug/expr.c,84,make_token] match rules[5] = ")" at position 18 with len 1: )
result = 10
```

图 2.6 表达式求值结果 3

2.2.4 扫描内存

实验结果如图 2.7 所示。输入 `x 4 0x80100000` 命令，系统能正确识别 token 并计算表达式 `0x80100000` 的值，输出以它为起始地址的连续 16 个字节。

```

Welcome to riscv32-NEMU!
For help, type "help"
(nemu) si 1
80100000: b7 02 00 80                                lui 0x80000,t0
(nemu) x 4 0x80100000
[src/monitor/debug/expr.c,84,make_token] match rules[6] = "(0x|0X)[0-9a-fA-F]+" at position 0 with len 10: 0x80100000
0x80100000 :0x800002b7 0x2a023 0x2a503 0x6b
(nemu) █

```

图 2.7 扫描内存结果

2.2.5 设置监视点

输入 `w $t0` 命令，程序最初出现如图 2.8 的报错。经过检查发现，是由于 `cmd_table[]` 中没有写入对 `cmd_w` 的调用。

```

Welcome to riscv32-NEMU!
For help, type "help"
(nemu) w $t0
make: *** [Makefile:77: run] Segmentation fault (core dumped)

```

图 2.8 设置监视点结果 1

修改后再次运行 `w $t0` 命令，程序能够正确解析表达式 `$t0`，并设置监视点。输入 `c` 命令继续运行，程序能够识别监视点值的变化并输出。继续输入 `info w` 命令，程序能够打印出监视点的相关信息。如图 2.9 所示。

```

Welcome to riscv32-NEMU!
For help, type "help"
(nemu) w $t0
[src/monitor/debug/expr.c,83,make_token] match rules[12] = "\\$[a-zA-Z_][a-zA-Z0-9_]*" at position 0 with len 3: $t0
New watchpoint: NO.0, expression = $t0
(nemu) c
[src/monitor/debug/expr.c,83,make_token] match rules[12] = "\\$[a-zA-Z_][a-zA-Z0-9_]*" at position 0 with len 3: $t0
Watchpoint NO.0 has changed:
expression = $t0, old value = 0, new value = 2147483648
(nemu) info w
NUM      EXPR      VAL
0        $t0      2147483648

```

图 2.9 设置监视点结果 2

2.2.6 删除监视点

实验结果如图 2.10 所示。首先用 `w $t0` 命令设置一个监视点，其序号为 0，再输命令 `d 0`，能够成功删除。

```

Welcome to riscv32-NEMU!
For help, type "help"
(nemu) w $t0
[src/monitor/debug/expr.c,83,make_token] match rules[12] = "\\$[a-zA-Z_][a-zA-Z0-9_]*" at position 0 with len 3: $t0
New watchpoint: NO.0, expression = $t0
(nemu) c
[src/monitor/debug/expr.c,83,make_token] match rules[12] = "\\$[a-zA-Z_][a-zA-Z0-9_]*" at position 0 with len 3: $t0
Watchpoint NO.0 has changed:
expression = $t0, old value = 0, new value = 2147483648
(nemu) d 0
Watchpoint NO.0 has been deleted.

```

图 2.10 删除监视点结果

2.3 实验必答题

1. ISA

问：选择的 ISA 是哪一个？

答：我选择的 ISA 是 riscv32。

2. 理解基础设施

问：假设你需要编译 500 次 NEMU 才能完成 PA，有 90% 的次数用于调试。如果没有实现简易调试器，只能通过 GDB 对 NEMU 上的客户程序调试。每次调试中，需要花 30 秒从 GDB 中获取并分析一个信息。你需要分析 20 个信息才能排除一个 bug。那么这个学期，你将会在调试上花费多少时间？由于简易调试器可以直接观测客户程序，只需要花费 10 秒从中获取并分析相同的信息。那么这个学期，简易调试器可以帮助你节省多少调试的时间？

答：总调试时间 = 编译次数 × 调试比例 × (获取和分析一个信息所需时间) × (解决一个 bug 需要的信息数)。

使用 GDB 时，总调试时间 = $500 \times 0.9 \times 30 \text{ 秒} \times 20 = 27000 \text{ 秒}$ 。使用简易调试器时，总调试时间 = $500 \times 0.9 \times 10 \text{ 秒} \times 20 = 9000 \text{ 秒}$ ，相当于节省 18000 秒调试的时间。由此可见，如果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。

3. 查阅手册

问：理解了科学查阅手册的方法之后，请尝试在选择的 ISA 手册中查阅以下问题所在的位置。①riscv32 有哪几种指令格式？②LUI 指令的行为是什么？③mstatus 寄存器的结构是怎么样的？

答：① 信息在《The RISC-V Reader》文档的 25 页。riscv32 有 6 种基本指令格式，分别是用于寄存器-寄存器操作的 R 类型指令，用于短立即数和访存 load 操作的 I 型指令，用于访存 store 操作的 S 型指令，用于条件跳转操作的 B 类型指令，用于长立即数的 U 型指令和用于无条件跳转的 J 型指令。① 信息在《The RISC-V Reader》的 129 页。②LUI 指令的扩展形式为 `lui rd, imm`，用于高位立即数加载。③信息在《The RISC-V Reader》的 103 页。mstatus 寄存器保存全局中断使能和许多其他的状态，具体结构如图 2.11 所示。

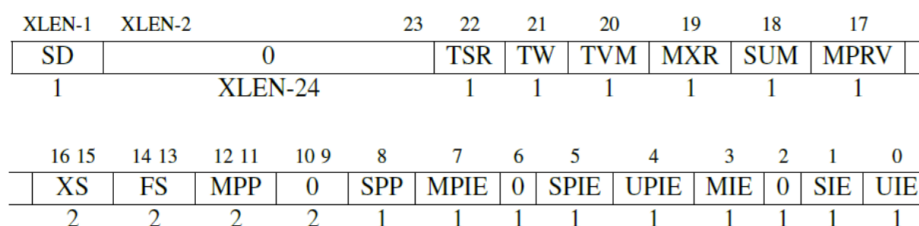


图 2.11 mstatus 寄存器结构

4. shell 命令

问：完成 PA1 的内容之后，nemu/目录下的所有.c 和.h 文件总共有多少代码？你是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？

答：在 PA1 分支使用如下命令来获取 nemu/ 目录下的所有 .c 和 .h 文件的不包含注释和空行在内的代码行数：

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

完成 PA1 之后，nemu/目录下的.c 和.h 文件共有 5371 行代码；再切换到 PA0 分支，共有 4968 行代码，即在 PA1 中编写了 403 行。

5. man 命令

问：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：①-Wall 选项会使编译器对代码中的潜在问题提出警告，包括但不限于未使用的变量、类型不匹配、未初始化的变量等。通过启用 -Wall，开发者可以更容易地发现代码中的问题，提高代码质量和可维护性。②-Werror 选项将警告信息视为错误，即如果编译器发出任何警告，它将导致编译过程失败。将警告视为错误的主要目的是强制开发者解决代码中的警告，以确保代码的质量和稳定性。③启用-Wall 和 -Werror 可以提高代码质量和可维护性，有助于避免潜在的运行时问题和错误。

2.4 心得体会

通过本次实验，我完成了单步执行、打印程序状态、表达式求值等功能，再次复习了编译原理、操作系统等方面的知识，从而进一步巩固所学的计算机知识。通过实现 NEMU，我也更深的理解了 CPU 的原理和调试器的工作内容。

在实验过程中，我们需要结合提示文字理解 NEMU 的框架代码，这也锻炼了我接手一个陌生项目的的能力，能够分阶段分重点的阅读代码，我也体会到了理解代码框架是一个螺旋上升的过程，不必因为看不懂某些细节而感到沮丧。通过“如何阅读手册”章节，我学会了通过目录等信息快速定位目标问题，逐步细化搜索范围，这种筛选信息的能力在今后的学习工作中也十分重要。

同样，在实验中我也遇到了一些问题，例如正则式匹配错误、地址越界等，在分析报错信息、和同学沟通交流后成功解决了。这也锻炼了我的调试、分析和解决问题的能力。

3 PA2：冯诺依曼计算机系统

3.1 方案设计

PA2 涉及 NEMU, Nexus-AM 子项目, 主要考察 CPU 和输入输出设备的知识。具体由 3 个子任务构成:

- 1) 根据 CPU 取指-译码-执行的指令周期, 在 NEMU 中运行第一个 C 程序 dummy。
- 2) 实现更多的指令, 在 NEMU 中运行所有的 cputest。
- 3) 补充输入输出设备的相关代码, 运行打字小游戏。

以下分别介绍这 3 部分的实现。

3.1.1 实现 dummy 程序

首先在 nexus-am/tests/cputest/目录下输入以下命令编译运行 dummy.c 程序:

```
make ARCH=riscv32-nemu ALL=dummy run
```

得到对应的反汇编文件 dummy-riscv32-nemu.txt, 如图 3.1 所示:

```
7  80100000 <_start>:
8  80100000:  00000413          li  s0,0
9  80100004:  00009117      auipc  sp,0x9
10 80100008:  ffc10113      addi   sp,sp,-4 # 80109000 <_end>
11 8010000c:  00c000ef      jal   ra,80100018 <_trm_init>
12
13 Disassembly of section .text.startup:
14
15 80100010 <main>:
16 80100010:  00000513          li  a0,0
17 80100014:  00008067          ret
```

图 3.1 反汇编结果

为了成功运行 dummy.c 程序, 需要在 NEMU 中补充该文件中的所有汇编指令: addi, auipc, j, jal, jalr, li, lui, mv, ret, sw。而由文档可知, 系统引入了对译码、执行和操作数宽度的解耦实现和 RTL, 因此添加一条新指令只需以下两步: ① 在 opcode_tabl 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。② 用 RTL 实现正确的执行辅助函数, 使用时需要遵守小型调用约定。阅读代码可以发现, j 和 lui 在系统中均已实现, 只需要完成 addi, auipc, jal, jalr, li, mv, sw, ret 指令。以下分别介绍他们的实现过程。

- 1) addi/ li/ mv

查阅 RISC-V 手册可知, addi/ li/ mv 指令均为 I 型指令, 且 2-5 位都是 00100,

在 `opcode_table` 中处于同一位置，因此合并在一起实现。根据取指-译码-执行的指令周期，首先在 `decode.c` 中实现他们的译码函数，代码如下：

```
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);
}
```

该函数可用于解析所有的 I 类型指令，其中 `id_src` 存储源操作数，`id_src2` 存储立即数，`id_dest` 存储目的操作数。在解析过程中，通过 `decode_op_r` 和 `decode_op_i` 宏，分别解析寄存器和立即数，并将结果存储到对应的操作数结构中。接着，在 `compute.c` 中实现他们的执行函数，代码如下：

```
make_EHelper(I) {
    switch (decinfo.isa.instr.funct3) {
        case 0: // li, mv, 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);
            }
        }
        rtl_sr(id_dest->reg, &id_dest->val, 4);
    }
}
```

该函数可用于执行所有的 I 型指令，并根据 `decinfo.isa.instr.funct3` 字段判断是什么指令。如果为 0 则对应 `addi/li/mv`，直接使用 `rtl_addi()` 函数计算目的操作数的值。再通过 `decinfo.isa.instr.rs1` 字段区分这 3 个指令，分别调用 `print_asm_template()` 打印汇编指令模版。最后通过 `rtl_sr()` 将目的操作数的值写回目的寄存器。

由于这 3 条指令的 2-5 位都是 00100，结合以上译码函数和执行函数，需要将 `opcode_table` 的第 4 为改为 `IDEX(I, I)`。

2) auipc

`auipc` 为 U 型指令，其译码函数 `make_DHelper(U)` 已实现，只需要补充执行函数。`auipc` 主要用于生成一个相对于当前 PC 的大立即数，在执行中先通过 `rtl_add()` 将立即数与当前程序计数器 `cpu.pc` 相加，然后将结果写入目的寄存器 `id_dest->reg`。最后通过 `print_asm_template2` 打印汇编指令模板。代码如下：

```

make_EHelper(auipc) {
    rtl_add(&id_dest->val, &cpu.pc, &id_src->val);
    rtl_sr(id_dest->reg, &id_dest->val, 4);
    print_asm_template2(auipc);
}

```

auipc 的 2-5 位为 00101，结合译码和执行函数，需要将 opcode_table 的第 5 为改为 IDEX(U, auipc)。

3) jal

jal 为 J 型指令，用于无条件跳转。其译码函数如下：

```

make_DHelper(J) {
    s0 = (decinfo.isa.instr.simm20<<20) + (decinfo.isa.instr.imm19_12<<12)
        + (decinfo.isa.instr.imm11_1<<11) + (decinfo.isa.instr.imm10_1<<1);
    decode_op_i(id_src, s0, true);
    decode_op_r(id_dest, decinfo.isa.instr.rd, false);
}

```

该函数可以用于解析所有 J 型指令。在解析过程中，先计算跳转目标的地址，再用过 decode_op_i()和 decode_op_r()分别设置源、目的操作数。jal 具体的执行逻辑为把下一条指令的地址(pc+4),然后把 pc 设置为当前值加上符号位扩展的 offset，执行函数代码在 control.c 文件中实现：

```

make_EHelper(jal) {
    s0 = cpu.pc + 4;
    rtl_sr(id_dest->reg, &s0, 4); // x[rd]=pc+4
    rtl_add(&decinfo.jump_pc, &cpu.pc, &id_src->val); // pc+=sext(offset)
    rtl_j(decinfo.jump_pc);
    print_asm_template2(jal);
}

```

jal 的 2-5 位为 11011，需要将 opcode_table 的第 27 为改为 IDEX(J, jal)。

4) jalr/ ret

根据 RISC-V 手册，ret 指令和 jalr 指令对应的 opcode 相同，因此只需实现 jalr。jalr 为 I 型指令，其译码函数 make_DHelper(I)已实现，只需在 control.c 中补充执行函数。jalr 执行过程和 jal 类似，但需要把 pc 设置为 x[rs1]+sign-extend(offset)，因此只需改写 rtl_add()函数的调用，代码如下：

```

rtl_add(&decinfo.jump_pc, &id_src->val, &id_src2->val); // pc=(x[rs1]+sext(offset))&~1

```

jalr 的 2-5 位为 11001，需要将 opcode_table 的第 25 为改为 IDEX(I, jalr)。

5) sw

sw 为 S 型指令，其译码和执行函数系统均已实现，只需根据对应的宽度修改对应的 store_table：

```
static OpcodeEntry store_table [8] = {
    EXW(st, 1), EXW(st, 2), EXW(st, 4), EMPTY, EMPTY, EMPTY, EMPTY,
    EMPTY
};
```

最后,在 decode.h 和 all-instr.h 中申明实现的所有辅助函数,即可通过 dummy.c 的测试。

3.1.2 实现所有 cputest

此阶段需要实现更多的汇编指令, 来通过 nexus-am/tests/cputest/目录下的所有测试用例。其中, string.c 和 hello-str.c 使用了未实现的系统库函数, 需要自己编写, 并且 TODO 处的部分函数也需要补充实现。因此本阶段的实现分为这 3 部分介绍: 实现库函数、实现 TODO 处辅助函数、实现所有指令。

1. 实现所有库函数

1) 字符串库函数

字符串相关的库函数在 nexus-am/libs/klib/src/string.c 文件中, 包含 strlen(), strcpy(), strncpy(), strcat(), strcmp(), strncmp(), memset(), memcpy(), memcmp()。由于这些函数简单常见, 此处的具体实现略。

2) vsprintf()

vsprintf()是格式化函数, 用于根据提供的格式字符串 fmt 和可变参数 ap 进行字符串格式化。它遍历格式字符串, 通过调用相应的辅助函数处理每个格式说明符, 如 “%d”, “%x”, “%u”, “%s”, “%c”。格式化的字符被写入由 out 指向的内存位置, 最终以空字符结尾并返回写入的字符数。函数的流程图如图 3.2 所示。

其中, my_putchar()是一个辅助函数, 用于 “%c” 格式, 即将单个字符 c 写入由 **str 指向的内存位置, 然后递增 *str 指针。代码如下:

```
void my_putchar(char **str, char c) {
    if (str) { **str = c; ++(*str); }
}
```

my_puts()用于 “%s” 格式, 即将以 null 结尾的字符串 s 写入由 **str 指向的内存位置。它通过迭代字符串中的每个字符, 使用 my_putchar()逐个写入。代码如下:

```
void custom_puts(char **str, const char *s) {
    while (*s) { my_putchar(str, *s++); }
}
```

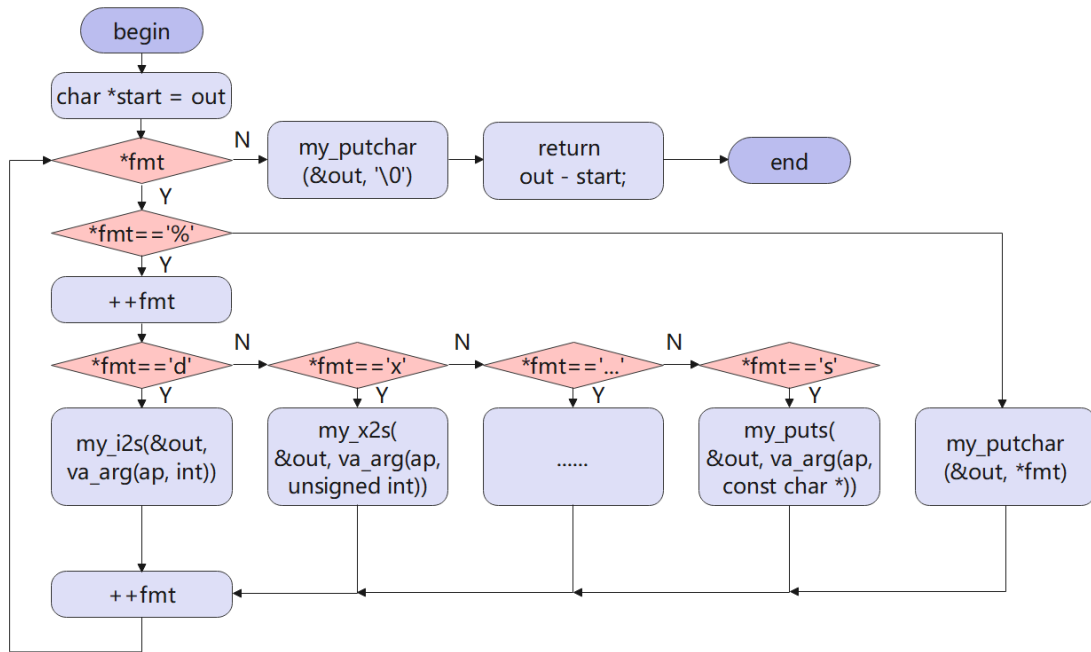


图 3.2 vsprintf()流程图

my_i2s()用于“%d”和“%u”格式，即将整数 num 转换为其 ASCII 表示，并写入由**str 指向的内存位置。它首先处理符号，递归地将数字除以 10 直到小于 10，再将各位数字写入内存位置。代码如下：

```

void my_i2s(char **str, int num) {
    if (num < 0) { my_putchar(str, '-'); num = -num; }
    if (num >= 10) { my_i2s(str, num / 10); }
    my_putchar(str, '0' + (num % 10));
}

```

my_x2s()用于“%x”格式，即将无符号整数 num 转换为其十六进制表示，并写入由**str 指向的内存位置。它使用一个固定数组 hex_digits 存储十六进制字符，并递归地将数字除以 16。代码如下：

```

void my_x2s(char **str, unsigned int num) {
    const char hex_digits[] = "0123456789ABCDEF";
    if (num >= 16) { my_x2s(str, num / 16); }
    my_putchar(str, hex_digits[num % 16]);
}

```

3) printf()函数

结合 C 库 stdarg.h 中提供的宏，可以用上面实现的 vsprintf()完成 printf()。具体代码如下：

```

int printf(const char *fmt, ...) {
    va_list args;
    va_start(args, fmt);
    char buffer[1000];

```

```

    int out_len = vsprintf(buffer, fmt, args);
    for(int i=0; buffer[i] != '\0'; i++) {
        _putc(buffer[i]);
    }
    va_end(args);
    return out_len;
}

```

以上代码中，首先创建一个变量参数列表 `args`，调用 `va_start()` 初始化参数列表，指向第一个可变参数。接着通过 `vsprintf()` 将格式化后的字符串存储到缓冲区 `buffer` 中，并获得格式化后的字符串长度 `out_len`。再调用 `_putc()` 逐个字符输出到标准输出，最后通过 `va_end()` 清理参数列表。

4) `snprintf()` 函数

和 `printf()` 类似，借助 `stdarg.h` 中的宏和 `vsprintf()` 可以轻松实现 `snprintf()`。具体过程略。

2. 实现 TODO 处辅助函数

为了通过所有 `cputest`，还需要实现 `nemu/include/rtl/rtl.h` 中 RTL 伪指令相关的辅助函数。

1) `rtl_not()`

函数接受两个参数 `dest` 和 `src1`，它们分别是指向 RTL 目的和源操作数寄存器的指针。函数的作用是将 `src1` 中的值按位取反，并将结果存储到 `dest` 中。实现过程较简单，此处略。

2) `rtl_sext()`

此函数用于对有符号整数进行符号扩展，将源整数的高位符号位复制扩展到目标整数的高位。实现时首先将 `src1` 转换为 `int32_t` 类型的临时变量 `temp`，再根据要扩展的字节数，分别进行不同的符号扩展。代码如下：

```

static inline void rtl_sext(rtlreg_t* dest, const rtlreg_t* src1, int width) {
    // dest <- signext(src1[(width * 8 - 1) .. 0])
    int32_t temp = *src1;
    switch(width) {
        case 4: *dest = *src1; break;
        case 3: temp = temp << 8; *dest = temp >> 8; break;
        case 2: temp = temp << 16; *dest = temp >> 16; break;
        case 1: temp = temp << 24; *dest = temp >> 24; break;
        default: assert(0);
    }
}

```

3) `rtl_msb()`

这个函数主要是获取有符号整数的最高有效位的值。对于一个 `width` 字节的

整数，其最高有效位即为整数的符号位。具体实现过程略。

4) rtl_mux()

这个函数实现了一个三目运算符的功能，根据条件 `cond` 选择将 `src1` 或 `src2` 的值存入目的操作数。具体实现过程略。

3. 实现所有指令

补充好以上的库函数和其他辅助函数后，在 `nemu/` 目录运行以下命令，即可得到 `cputest` 中所有测试文件的反汇编结果：

```
bash runall.sh ISA=riscv32
```

进而得到需要实现的所有汇编指令。根据指令在 `opcode_table` 中的位置，我们将系统还未实现的合并为 3 组，如表 3.1 所示。（`load` 和 `store` 相关的指令将在后续讨论）

表 3.1 `opcode_table` 中的指令分组

x	opcode_table[x]	译码/执行函数	包含指令
00100	IDEX(I, I)	make_DHelper(I) make_EHelper(I)	I-type: slli, seqz, sltiu, not, xori, srai, srli, andi
01100	IDEX(R, R)	make_DHelper(I) make_EHelper(R)	R-type: add, sll, slt, sltu, xor, srl, or, and, mul, mulh, mulhu, div, divu, rem, sub, neg, sra
11000	IDEX(B, B)	make_DHelper(B) make_EHelper(B)	B-type: beq, beqz, bne, bnez, blt, bgtz, bltz, bge, ble, blez, bltu, bgeu, bleu

以下分别介绍这些指令的具体实现。

1) I-type 指令

I-type 指令包括 `slli`, `seqz`, `sltiu`, `not`, `xori`, `srai`, `srli`, `andi`。其译码函数为 `make_DHelper(I)`，在 3.1.1 中已实现；执行函数为 `make_EHelper(I)`，在 3.1.1 中完成了 `addi`, `li`, `mv` 的部分。由于这些指令的 `decinfo.isa.instr.funct3` 大都不同，实现时现根据这一字段区分每个指令，并通过 RTL 伪指令分别处理。若有 `funct3` 相同的，则再根据 `funct7` 等字段区分。部分代码如下：

```
make_EHelper(I) {
    switch (decinfo.isa.instr.funct3) {
        .....
        case 1: // slli
            rtl_shli(&id_dest->val, &id_src->val, decinfo.isa.instr.simm11_0);
            print_asm_template3(slli);
            break;
        .....
        case 5:
            if(decinfo.isa.instr.funct7) { // srai
```



```

        rtl_sari(&id_dest->val, &id_src->val, decinfo.isa.instr.rs2);
        print_asm_template3(srai);
    } else {
        // srli
        rtl_shri(&id_dest->val, &id_src->val, decinfo.isa.instr.rs2);
        print_asm_template3(srli);
    }
    .....
    rtl_sr(id_dest->reg, &id_dest->val, 4);
}

```

2) R-type 指令

R-type 指令的译码函数为 `make_DHelper(R)`，与 I-type 的类似，依次识别 2 个源操作数和 1 个目的操作数即可，具体代码略。

R-type 指令的执行函数为 `make_EHelper(R)`。为了更准确地区分不同的指令，函数先将 `funct7` 左移 3 位，与 `funct3` 相加，得到 `funct73`。再通过该字段的值执行相应的运算，如加法、减法、与、或、异或等等。每个 case 处理一种指令，并调用相应的 RTL 操作实现指令功能，调用 `print_asm_template3()` 打印汇编指令的模板信息。最后通过 `rtl_sr()` 将计算结果写回目标寄存器。部分代码如下：

```

make_EHelper(R) {
    int funct73 = (((unsigned)decinfo.isa.instr.funct7)<<3) +
        ((unsigned)decinfo.isa.instr.funct3);
    switch(funct73) {
        case 0b00000000000: // add
            rtl_add(&id_dest->val, &id_src->val, &id_src2->val);
            print_asm_template3(add);
            break;
        case 0b00000000001: // sll
            rtl_shl(&id_dest->val, &id_src->val, &id_src2->val);
            print_asm_template3(sll);
            break;
        .....
        default:
            assert(0);
    }
    rtl_sr(id_dest->reg, &id_dest->val, 4);
}

```

3) B-type 指令

B-type 的译码函数为 `make_DHelper(B)`。首先通过按位左移和按位或操作将各个字段组合成目标地址，并存储到临时寄存器 `s0` 中。接着用 `decode_op_r()` 解码指令的两个源操作数 `rs1` 和 `rs2`。最后用 `rtl_add()` 将 `s0` 和当前指令的地址 `cpu.pc` 相加，得到最终的分支目标地址，并将结果存储在 `decinfo.jump_pc` 中。这个地址

在后续的执行阶段用于实现条件分支跳转。代码如下：

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

B-type 的执行函数为 `make_DHelper(B)`。根据 `nemu/include/rtl/relop.h` 中的关系运算枚举类型，这里使用 `branch_table` 数组来确定分支的条件。执行过程中，先调用 `rtl_jrelop()`，根据传入的 `funct3`、源和目的操作数来判断是否满足条件，从而实现条件跳转。接着根据 `branch_table[decinfo.isa.instr.funct3]` 的值区分并不同指令，打印相应的汇编指令模板，用于调试和输出。具体代码如下：

```
static uint32_t branch_table[8] = { RELOP_EQ, RELOP_NE, RELOP_FALSE,
RELOP_FALSE, RELOP_LT, RELOP_GE, RELOP_LTU, RELOP_GEU };

make_EHelper(B) {
    rtl_jrelop(branch_table[decinfo.isa.instr.funct3], &id_src->val, &id_src2->val,
decinfo.jump_pc);
    switch(branch_table[decinfo.isa.instr.funct3]) {
        case RELOP_EQ: print_asm_template3(beq);break;        // beq, beqz
        case RELOP_NE: print_asm_template3(bne);break;        // bne, bnez
        .....
        default:      assert("Illegal branch opcode!");break;
    }
}
```

4) load 相关指令

load 相关的指令有 `lw`, `lh`, `lhu`, `lb`, `lbu`。其中 `lw`, `lhu`, `lbu` 的译码和执行函数均已实现，只需补充 `lh` 和 `lb` 的执行函数，在 `witch` 语句中根据指令宽度和 `decinfo.isa.instr.funct3` 判断并操作即可。代码如下：

```
make_EHelper(ld) {
    rtl_lm(&s0, &id_src->addr, decinfo.width);
    switch (decinfo.width) {
        case 4: print_asm_template2(lw); break;        // lw
        case 2:                                     // lh, lhu
            if(decinfo.isa.instr.funct3 == 1) {rtl_sext(&s0, &s0, 2); print_asm_template2(lh);}
            else { print_asm_template2(lhu); }
            break;
        case 1:                                     // lb, lbu
            if (decinfo.isa.instr.funct3 == 0) {rtl_sext(&s0, &s0, 1); print_asm_template2(lb);}
            else { print_asm_template2(lbu); }
            break;
        default: assert(0);
    }
```

```

    }
    rtl_sr(id_dest->reg, &s0, 4);
}

```

5) store 相关指令

store 相关指令有 sw, sh, sb, 系统均已实现。

3.1.3 输入输出

此阶段主要包含串口、时钟、键盘、VGA 设备的输入输出实现，涉及大量的 API 调用。以下分别介绍各部分的实现。

1) 串口

系统在 nexus-am/am/src/nemu-common/trm.c 中已经提供了串口的功能，只需要在 nemu/include/common.h 中定义宏 HAS_IOE，即可编译运行 hello 程序。

2) 时钟

首先完善 __am_timer_init(), 通过通过读取 RTC 地址来获取系统的启动时间，完成初始化。代码如下：

```

static uint32_t boot_time;
void __am_timer_init() {
    boot_time = inl(RTC_ADDR);
}

```

接着在 __am_timer_read() 中实现宏 _DEVREG_TIMER_UPTIME。函数先从 RTC 地址读取当前时间，减去系统启动时间 boot_time，得到系统运行的时间，并更新 _DEV_TIMER_UPTIME_t 结构的高位和低位。代码如下：

```

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);

```

3) 键盘

为了实现 readkey test 测试，需要在 nexus-am/am/src/nemu-common/nemu-input.c 中实现 _DEVREG_INPUT_KBD 的功能：通过读取键盘寄存器，解析键盘状态和键码。函数首先将从键盘地址 KBD_ADDR 读取的键盘状态和键码合并到一个 32 位的值中。通过与 KEYDOWN_MASK 进行按位与运算，得到键盘是否按下，并记录到 kbd->keydown 中；通过与 KEYDOWN_MASK 进行取反运算，得到键盘的键码，并记录到 kbd->keycode 中。其中，宏 KEYDOWN_MASK 是

一个用于检测键盘按键是否按下的掩码，值为 0x8000。具体代码如下：

```
case _DEVREG_INPUT_KBD:
    _DEV_INPUT_KBD_t *kbd = (_DEV_INPUT_KBD_t *)buf;
    uint32_t keyboard_code = inl(KBD_ADDR);
    kbd->keydown = keyboard_code & KEYDOWN_MASK ? 1 : 0;
    kbd->keycode = keyboard_code & ~KEYDOWN_MASK;
    return sizeof(_DEV_INPUT_KBD_t);
```

4) VGA

VGA 可以用于显示颜色像素，是最常用的输出设备。为了实现绘制图像的功能，需要在 nemu-video.c 中补充宏 _DEVREG_VIDEO_FBCTL 的代码。

函数先将传入的 buf 强制类型转换为 _DEV_VIDEO_FBCTL_t 结构，方便后续的操作。接着提取结构中的各个字段，如像素数据 pixels、起始坐标 x、y、绘制宽度 w、高度 h、屏幕宽度 W 和高度 H。接着计算需要拷贝的字节数 copy_bytes，避免越界访问。获取显存的起始地址，并将 pixels 中的图像数据拷贝到显存中对应的位置。最后判断在图像绘制完成后是否需要同步，返回 size。代码如下：

```
case _DEVREG_VIDEO_FBCTL:
    _DEV_VIDEO_FBCTL_t *ctl = (_DEV_VIDEO_FBCTL_t *)buf;
    uint32_t *pixels = ctl->pixels;
    int x = ctl->x, y = ctl->y, w = ctl->w, h = ctl->h;
    int W = screen_width();
    int 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++) {
        memcpy(&vmem[(y + i) * W + x], pixels, copy_bytes);
        pixels += w;
    }
    if (ctl->sync) { outl(SYNC_ADDR, 0); }
    return size;
```

3.2 结果分析

3.2.1 实现 dummy 程序

实验结果如图 3.3 所示。输入 `make` 命令，系统能部分正确编译，但在运行中一直卡住，不能正确运行至出现“HIT GOOD TRAP”。如图 3.3 所示。

```
Welcome to riscv32-NEMU!  
For help, type "help"  
^Z  
[1]+  Stopped                  make ARCH=riscv32-nemu ALL=dummy run
```

图 3.3 dummy 错误结果

经过检查调试发现，`jal` 的译码程序 `make_DHelper(J)` 中的移位运算缺少括号，导致程序不能正确运行结束。修改后可以成功执行，如图 3.4 所示。

```
Welcome to riscv32-NEMU!  
For help, type "help"  
nemu: HIT GOOD TRAP at pc = 0x80100030  
  
[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 13  
dummy  
hust@hust-desktop:~/ics2019/nexus-am/tests/cputest$
```

图 3.4 dummy 正确结果

3.2.2 实现所有 cputest

由于所有测试文件的指令不尽相同，一起测试时难以发现具体是哪些指令出现问题。因此该阶段先列出所有需要测试的文件，再各自编译运行每个文件，可快速判断出哪些指令是正确的，哪些指令有问题。

在测试 `if-else.c` 文件时，程序在 `pc = 0x80100128` 处报错，如图 3.5 所示。

```
Welcome to riscv32-NEMU!  
For help, type "help"  
nemu: HIT BAD TRAP at pc = 0x80100128  
  
[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 49  
if-else
```

图 3.5 if-else 错误结果

```
Welcome to riscv32-NEMU!  
For help, type "help"  
nemu: HIT GOOD TRAP at pc = 0x80100148  
  
[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 290  
if-else
```

图 3.6 if-else 正确结果

经过调试发现，实现 `slt` 指令的执行函数时缺少 `signed` 导致运算出错，正确修改后可以得到如图 3.6 所示的正确结果。修正后的代码如下：

```
case 0b0000000010: // slt
    id_dest->val = (signed) id_src->val < (signed)id_src2->val;
    print_asm_template3(slt);
```

成功后继续测试剩下的 `cputest`。在编译运行 `load-store.c` 文件时，程序出现如图 3.7 所示的报错。

```
Welcome to riscv32-NEMU!
For help, type "help"
please implement me
riscv32-nemu: ./include/rtl/rtl.h:140: rtl_sext: Assertion `0' failed.
make[2]: *** [Makefile:77: run] Aborted (core dumped)
make[1]: *** [/home/hust/ics2019/nexus-am/am/arch/platform/nemu.mk:27: run] Error 2
make: [Makefile:13: Makefile.load-store] Error 2 (ignored)
load-store
```

图 3.7 load-store 错误结果 1

由“please implement me”可知，程序是由于有其他函数未实现而报错。由“`rtl_sext: Assertion `0' failed`”可知，还需完成 `rtl_sext()` 等函数。完成后发现指令依然报错，如图 3.8 所示。

```
Welcome to riscv32-NEMU!
For help, type "help"
nemu: HIT BAD TRAP at pc = 0x80100128

[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 49
if-else
```

图 3.8 load-store 错误结果 2

经过调试发现，在实现 `lbu` 指令的执行函数时，调用 `rtl_sr()` 的位置放错。将其改到函数最后执行，即可得到正确运行结果。如图 3.9 所示。

```
Welcome to riscv32-NEMU!
For help, type "help"
nemu: HIT GOOD TRAP at pc = 0x80100148

[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 290
if-else
```

图 3.9 load-store 正确结果

经过以上修改后，在 `/nemu` 目录下输入命令 `bash runall.sh ISA = riscv32`，可以成功运行 `cputest` 的所有测试文件。如图 3.10 所示。

```

hust@hust-desktop:~/ics2019/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 3.10 所有 cputest 运行结果

3.2.3 输入输出

1) 串口

在 nexus-am/tests/amtest/目录下键入 `make mainargs=h run` 命令，程序能正确向终端输出 10 行信息。如图 3.11 所示。

```

Welcome to riscv32-NEMU!
For help, type "help"
Hello, AM World @ riscv32
Hello, AM World @ riscv32
Hello, AM World @ riscv32
Hello, AM World @ riscv32
Hello, AM World @ riscv32
Hello, AM World @ riscv32
Hello, AM World @ riscv32
Hello, AM World @ riscv32
Hello, AM World @ riscv32
Hello, AM World @ riscv32
nemu: HIT GOOD TRAP at pc = 0x80100e60

```

图 3.11 串口运行结果

2) 时钟

实现宏 `_DEVREG_TIMER_UPTIME` 后，在 riscv32-nemu 中运行 real-time clock test 测试，程序能正确的每隔 1 秒向终端输出信息，如图 3.12 所示。

```

Welcome to riscv32-NEMU!
For help, type "help"
2000-0-0 2d:2d:2d GMT (1 second).
2000-0-0 2d:2d:2d GMT (2 seconds).
2000-0-0 2d:2d:2d GMT (3 seconds).
2000-0-0 2d:2d:2d GMT (4 seconds).
2000-0-0 2d:2d:2d GMT (5 seconds).

```

图 3.12 时钟运行结果

3) 键盘

实现宏 `_DEVREG_INPUT_KBD` 后，在 `riscv32-nemu` 中运行 `readkey test` 测试。在程序弹出的新窗口中按下各种按键，能够输出正确的按键信息，包括按键名、键盘码、按键状态，如图 3.13 所示。

```

Welcome to riscv32-NEMU!
For help, type "help"
Try to press any key...

Get key: 54 RETURN down
Get key: 54 RETURN up
Get key: 41 BACKSLASH down
Get key: 41 BACKSLASH up
Get key: 44 S down
Get key: 44 S up
Get key: 58 C down
Get key: 58 C up

```

图 3.13 键盘运行结果

4) VGA

实现宏 `_DEVREG_VIDEO_FBCTL` 后，在 `riscv32-nemu` 中运行 `display test` 测试。程序出现了如图 3.14 所示的报错。

```

Welcome to riscv32-NEMU!
For help, type "help"
please implement me
riscv32-nemu: src/device/vga.c:33: vga_io_handler: Assertion `0' failed.
make[1]: *** [Makefile:77: run] Aborted (core dumped)
make[1]: Leaving directory '/home/hust/ics2019/nemu'
make: *** [/home/hust/ics2019/nexus-am/am/arch/platform/nemu.mk:27: run] Error 2

```

图 3.14 VGA 错误结果

由“please implement me”可知，程序所需的 `vga_io_handler()` 还未实现。在系统中查找并实现该函数后，程序弹出的新窗口能正确输出相应的动画效果，如图 3.15 所示。

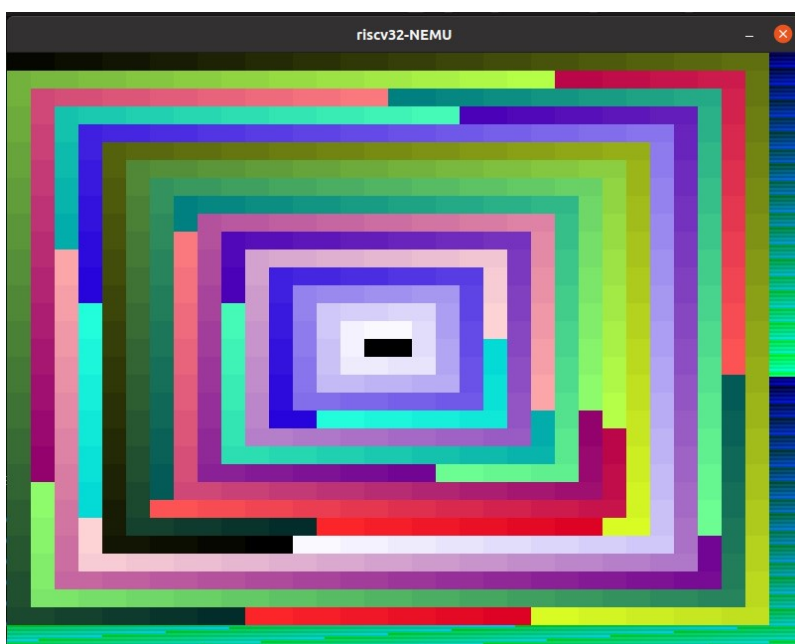


图 3.15 VGA 正确结果

5) 可展示的计算机系统

完整地实现以上 IOE 后，就可以运行系统重的幻灯片播放和打字小游戏了。打字小游戏在 `nexus-am/apps/typing/` 目录下，来源于 2013 年 NJUCS oslab0 的框架代码。为了配合移植，代码的结构做了少量调整，同时对屏幕更新进行了优化，并去掉了浮点数。实现效果如图 3.16 所示。

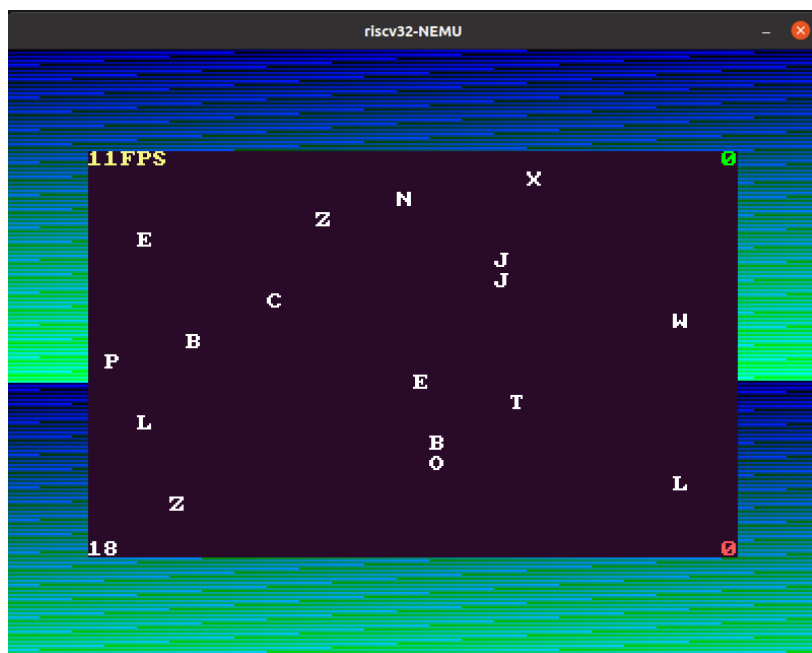


图 3.16 打字小游戏

幻灯片播放在 `nexus-am/apps/slider/` 目录下，正确运行后，程序将每隔 5 秒切

换 images/目录下的图片，效果如图 3.17 所示。

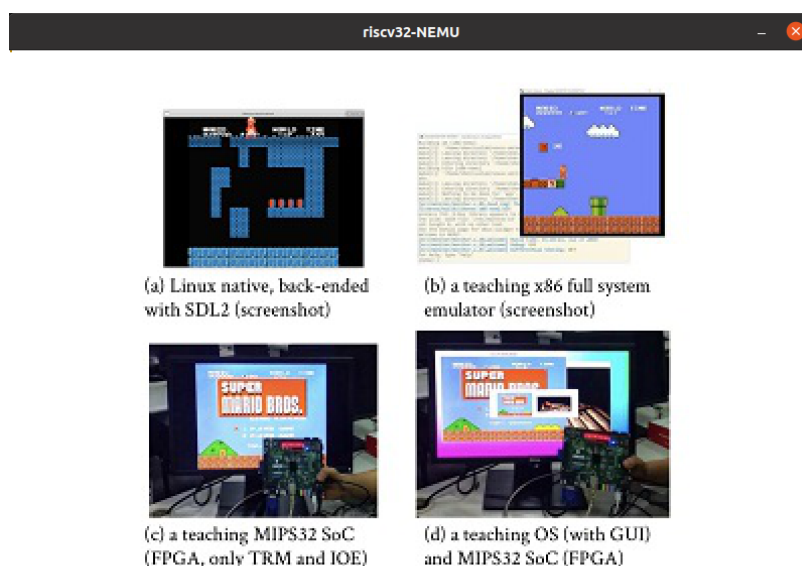


Figure 4. The same LiteNES emulator running on different platforms.

图 3.17 幻灯片播放

3.3 实验必答题

1. RTFSC

问：请整理一条指令在 NEMU 中的执行过程。

答：①取址：从指令存储器中读取下一条指令的地址，将其存储在 PC 中。
②译码：根据从指令寄存器中读取的操作码，解析指令并从寄存器文件中读取操作数。
③执行：根据译码阶段得到的操作数和指令类型进行相应的运算，得到执行结果。
④更新 PC：根据指令类型和执行结果，更新程序计数器（PC），指向下一条待执行的指令。

2. 编译与链接

问 1：在 common.h 中添加 volatile static int dummy，重新编译后的 NEMU 含有多少个 dummy 变量的实体？

答 1：一个。因为这里只在该文件中使用 static 修饰符定义了一次。

问 2：再在 nemu/include/debug.h 中添加 volatile static int dummy，重新编译后的 NEMU 含有多少个 dummy 变量的实体？

答 2：一个。虽然在两个文件中定义了，但这两个都使用了 static 修饰符，而且 debug.h 包含了 common.h，这样就相当于在 debug.h 中定义了两个包含 static

修饰符的 `dummy`，而在同一个文件中使用 `static` 修饰符同一个变量可以写多次定义，但只定义了一次。

问 3：为两处 `dummy` 变量进行初始化 `dummy = 0`，然后重新编译 NEMU。你发现了什么问题？

答 3：重新编译时可能会出现链接错误，提示符号重复定义。在之前的情况下，由于 `dummy` 变量都被定义为 `static`，它们处于各自文件的作用域内，不会被暴露给其他文件。因此，在链接时不会发生冲突。而现在，通过初始化将其放置在数据段中，这使得 `dummy` 的实体变得可见，导致了链接时的冲突。

3. 了解 Makefile

问：请描述在 `nemu/` 目录下，`make` 程序如何组织 `.c` 和 `.h` 文件，最终生成可执行文件。

答：Makefile 中定义了一些变量和规则，如 `$ISA` 表示当前使用的指令集体系结构。这些变量用于定制编译和链接的行为，规则规定了文件之间的依赖关系和如何生成目标文件。当执行 `make` 命令时，`make` 工具会根据规则和依赖关系判断哪些文件需要重新构建，最终生成可执行文件。

3.4 心得体会

在本阶段实验中，我实现了 `cputest` 测试文件所需的所有指令，顺利通过了测试，并完成了串口、键盘、时钟、VGA 等设备的输入输出功能，成功运行了幻灯片放映、打字小游戏。

在实现 `cputest` 的过程中，我深入研究了 RISC-V 指令集手册，理解了指令的编码和执行规则。通过逐一运行 `cputest`，我发现并修复了一些潜在的 bug，并进一步加深了对 CPU 执行流程的理解。这个任务对于提高对 CPU 指令集的熟练度非常有帮助，也让我对计算机体系结构有了更全面的认识。

通过阅读项目的 gitbook，我也养成了一些良好的编程习惯，其中一个就是遵守约定。在 PA2 中，系统定义了 RTL 寄存器和相应的 RTL 指令，基于这些定义，原则上可以编写出任意的 RTL 指令序列并执行。但光靠这些定义，无法避免 RTL 寄存器相互覆盖的错误，此时小型调用约定就可以很好的解决这类问题。

在第 3 阶段中，通过实现串口、时钟等输入输出设备，我更好地理解计算机系统 I/O 设备的工作原理，尤其是键盘输入和屏幕输出。

此外，阅读和理解 NEMU 和 Nexus-AM 的代码结构，也锻炼了我在大型项目中定位和理解代码的能力。刚接触这个项目时，我还不知道如何下手，但通过 RTFSC，能够慢慢明白项目的基本组织结构。在调试的过程中，我对这些模块的理解也逐渐清晰，最终能够游刃有余的在项目中添加新功能。

综上所述，本次 PA2 实验提供了一个全面的实践机会，通过深入的学习和实践，我不仅加深了对计算机系统的理解，还提升了编程和调试的能力。这对未来的计算机科学学习和职业发展都有着积极的影响。

4 PA3：批处理系统

4.1 方案设计

PA3 涉及 Nanos-lite 子项目，是一个为 PA 量身订造的操作系统。通过编写 Nanos-lite 的代码，可以更深刻地认识到操作系统是如何使用硬件提供的机制（即 ISA 和 AM）来支撑程序的运行的，这也符合 PA 的终极目标。具体由 3 个子任务构成：

- 1) 实现自陷操作 `_yield()` 及其过程。
- 2) 实现用户程序的加载和系统调用，支撑 TRM 程序的运行。
- 3) 实现文件系统和批处理系统，运行仙剑奇侠传并展示，提交完整的实验报告。

以下分别介绍这 3 部分的实现。

4.1.1 实现自陷操作

此阶段需要在 Nanos-lite 中触发一次自陷操作。为了实现最简单的操作系统，硬件需要提供一种可以限制入口的执行流切换方式，即自陷指令。riscv32 提供 `ecall` 作为自陷指令，具体步骤为：①设置异常入口地址、②触发自陷操作、③保存上下文、④事件分发、⑤恢复上下文。

- 1) 设置异常入口地址

设置异常入口地址，只需在 `nanos-lite/include/common.h` 中定义宏 `HAS_CTE`。这样系统初始化时便会调用 `init_irq()` 函数，最终通过 `_cte_init()` 来设置异常入口地址。

- 2) 触发自陷操作

为了测试异常入口地址是否设置正确，需要调用 `_yield()` 来触发自陷操作，即需要在 NEMU 中实现 `raise_intr()` 函数来模拟异常响应机制。代码如下：

```
void raise_intr(uint32_t NO, vaddr_t epc) {
    decinfo.isa.sepc = epc;
    decinfo.isa.scause = NO;
    decinfo.jmp_pc = decinfo.isa.stvec;
    rtl_j(decinfo.jmp_pc);
}
```

函数首先设置异常发生时的返回地址 `sepc` 和异常原因 `scause`，将中断处理的入口地址产给目标地址参数 `jmp_pc`，最终通过 `rtl_j()` 执行跳转，进入中断处理程

序。

3) 保存上下文

根据反汇编文件等内容，这阶段需要新增加一些指令：环境调用指令 `ecall`、管理员模式例外返回指令 `sret`、`csrrc` 等控制寄存器相关指令，此处将他们合并在一起实现。新指令的译码函数为 `make_DHelper(SYSTEM)`，与 I-type 指令的类似。执行函数为 `make_EHelper(SYSTEM)`，与 PA2 的实现方法相同：在 `switch` 语句中先根据 `decinfo.isa.instr.funct3` 字段判断出具体指令，再分别处理每个 `case`。部分代码如下：

```
make_EHelper(SYSTEM) {
    switch(decinfo.isa.instr.funct3) {
        case 0b000:          // ecall, sret
            if ((decinfo.isa.instr.val & ~(0x7f)) == 0) {          // ecall
                raise_intr(reg_l(17), cpu.pc);
            } else if (decinfo.isa.instr.val == 0x10200073) {      // sret
                decinfo.jump_pc = decinfo.isa.sepc + 4;
                rtl_j(decinfo.jump_pc);
            }
            break;
        case 0b001:          // csrrw
            s0 = read_csr(decinfo.isa.instr.csr);
            write_csr(decinfo.isa.instr.csr, id_src->val);
            rtl_sr(id_dest->reg, &s0, 4);
            print_asm_template3(csrrw);
            break;
        .....
    }
```

接着重新组织 `_Context` 结构体的成员，使得这些成员的定义顺序和 `nexus-am/am/src/$ISA/nemu/trap.S` 中构造的上下文保持一致：

```
struct _Context {
    uintptr_t gpr[32], cause, status, epc;
    struct _AddressSpace *as;
};
```

然后就可以在 `__am_irq_handle()` 中输出上下文 `c` 的内容，通过简易调试器观察触发自陷时的寄存器状态，从而检查 `_Context` 实现是否正确。

4) 事件分发

`__am_irq_handle()` 会把执行流切换的原因打包成事件，然后调用已经注册的事件处理回调函数，将事件交给 Nanos-lite 来处理。此部分需要在函数中补充编号为 `_EVENT_YIELD` 的自陷事件：

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

接着在 `do_event()` 中识别出自陷事件 `_EVENT_YIELD`，输出相应信息即可。

4.1.2 实现用户程序和系统调用

1) 用户程序

为了把用户程序加载到正确的内存位置和执行用户程序，需要在 Nanos-lite 中实现 loader 的功能。loader() 函数在 nanos-lite/src/loader.c 中定义，代码如下：

```
static uintptr_t loader(PCB *pcb, const char *filename) {
    Elf_Ehdr Ehdr;
    ramdisk_read(&Ehdr, 0, sizeof(Ehdr));
    for (uint16_t 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;
}
```

函数首先通过 `ramdisk_read()` 从虚拟磁盘中读取 ELF 头信息，接着遍历 ELF 文件的所有程序头表，读取程序头信息并判断是否为可加载类型 `PT_LOAD`。如果可加载，则将 ELF 文件中的数据加载到内存，最终返回 ELF 文件的入口地址。

然后在 `init_proc()` 中调用 `naive_oload(NULL, NULL)`，系统便通过实现的 loader 来加载第一个用户程序 `dummy`，并在 Nanos-lite 中触发一个未处理的 1 号事件。

2) 系统调用

操作系统中的系统调用由自陷指令实现，这一操作被打包成事件 `_EVENT_SYSCALL`。和 4.1.1 中的 `_EVENT_YIELD` 自陷事件类似，先在 `__am_irq_handle()` 中把执行流切换的原因打包成事件，交给 Nanos-lite 来处理，再调用 `do_event()` 函数根据事件类型再次进行分发。

Nanos-lite 收到该事件后，会调出系统调用处理函数 `do_syscall()` 进行处理。`do_syscall()` 首先通过宏 `GPR1` 从上下文 `c` 中获取用户进程之前设置好的系统调用参数，通过第一个参数系统调用号进行分发。接着添加系统调用，只需要在分发时添加相应的系统调用号，并编写相应的系统调用处理函数 `sys_xxx()`，最后通过宏 `GPRx` 来设置系统调用的返回值。`do_syscall()` 代码如下：

```

_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;
        .....
    }
    return NULL;
}

```

接着在文件中补充函数 `sys_yield()`、`sys_exit()`、`sys_write()`、`sys_brk()`，就可以实现系统调用 `SYS_yield`、`SYS_exit`、`SYS_write`、`SYS_brk`。

4.1.3 实现文件系统和批处理系统

1) 实现完整的文件系统

为了维护文件到 `ramdisk` 上的映射，完成各种文件操作，需要在 `nanos-lite/src/fs.c` 中实现 `fs_open()`、`fs_read()`、`fs_close()`、`fs_write()`和 `fs_lseek()`等常用的文件相关函数，并在 `Nanos-lite` 和 `Navy-apps` 的 `libos` 中添加相应的系统调用。此部分较简单常见，具体过程略。

在 4.1.2，系统在 `loader()`中直接调用 `ramdisk_read()`来加载用户程序。而当 `ramdisk` 中的文件数量增加之后，此种方式效率会降低。因此，在实现以上文件相关函数后，可以调用他们重新修改 `load()`：

```

static uintptr_t loader(PCB *pcb, const char *filename) {
    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;
}

```

2) 把 IOE 抽象成文件

为了实现一切皆文件的思想，需要扩展之前的文件系统，把串口、设备、VGA 等抽象成文件。

把串口抽象成文件，只需在 `device.c` 中实现 `serial_write()`，然后在文件记录表中设置相应的写函数。代码如下：

```

size_t serial_write(const void *buf, size_t offset, size_t len) {
    for (int i = 0; i < len; i++) { _putc(((char*)buf)[i]); }
    return len;
}

```

把设备输入抽象成文件，需要在 `device.c` 中实现 `events_read()`。函数首先调用 `read_key()` 获取当前的键盘输入，根据不同的键盘输入生成不同的输出信息，然后将这些信息写入 `buf` 中，从而模拟读取键盘输入的操作。代码如下：

```

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;
}

```

把 VGA 显存抽象成文件，首先要在 `device.c` 中实现 `fb_write()` 和 `fbsync_write()`。`fbsync_write()` 用于同步绘制，直接调用 IOE 的相应 API 即可。`fb_write()` 用于写入图形数据，函数首先计算每个像素的坐标，再调用 `draw_sync()` 确保图形绘制同步，最后调用 `draw_rect()` 以矩形的形式绘制 `buf` 中的数据，并返回写入数据的长度。代码如下：

```

size_t fb_write(const void *buf, size_t offset, size_t len) {
    int x = offset / 4 % screen_width();
    int y = offset / 4 / screen_width();
    draw_sync();
    draw_rect((uint32_t*)buf, x, y, len / 4, 1);
    return len;
}

```

3) 实现批处理系统

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，就可以在 `Nanos-lite` 中加载并运行游戏了。

为了展示批处理系统，需要添加系统调用 `SYS_execve` 来实现开机菜单程序，它的作用是结束当前程序的运行，并启动一个指定的程序。参考前面系统调用的实现，只需在 `do_syscall()` 中添加 `SYS_execve` 的 `case`，并调用 `sys_execve()` 处理。代码如下：

```
int sys_execve(const char *fname, char * const argv[], char *const envp[]) {
    printf("%s\n", fname);
    naive_oload(NULL, fname);
}
```

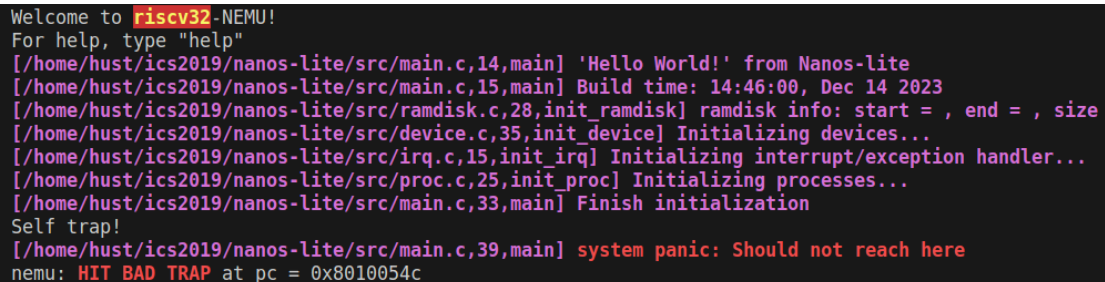
实现开机菜单程序后，还需要修改 `SYS_exit` 的实现，让它调用 `SYS_execve` 来再次运行 `/bin/init`，而不是直接调用 `_halt()` 来结束整个系统的运行。这样在一个用户程序结束的时候，操作系统就会自动再次运行开机菜单程序，来让用户选择一个新的程序来运行。修改后的 `sys_exit()` 如下：

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

4.2 结果分析

4.2.1 实现自陷操作

实现自陷所需的新指令和函数后，实验结果如图 4.1 所示。系统能够显示在 `do_event()` 中输出的信息 “Self trap!”，并触发了 `main()` 函数末尾设置的 `panic()`。



```
Welcome to riscv32-NEMU!
For help, type "help"
[/home/hust/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/ics2019/nanos-lite/src/main.c,15,main] Build time: 14:46:00, Dec 14 2023
[/home/hust/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = , end = , size
[/home/hust/ics2019/nanos-lite/src/device.c,35,init_device] Initializing devices...
[/home/hust/ics2019/nanos-lite/src/irq.c,15,init_irq] Initializing interrupt/exception handler...
[/home/hust/ics2019/nanos-lite/src/proc.c,25,init_proc] Initializing processes...
[/home/hust/ics2019/nanos-lite/src/main.c,33,main] Finish initialization
Self trap!
[/home/hust/ics2019/nanos-lite/src/main.c,39,main] system panic: Should not reach here
nemu: HIT BAD TRAP at pc = 0x8010054c
```

图 4.1 实现自陷操作结果

4.2.2 实现用户程序和系统调用

在 `init_proc()` 中调用 `naive_ufload(NULL, NULL)` 检测用户程序的实现，结果如图 4.2 所示。执行 `dummy` 程序时在 `Nanos-lite` 中触发了一个未处理的 1 号事件，说明 loader 已经成功加载 `dummy`。

```
Welcome to riscv32-NEMU!
For help, type "help"
[/home/hust/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/ics2019/nanos-lite/src/main.c,15,main] Build time: 14:29:42, Dec 14 2023
[/home/hust/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = , end = , size =
[/home/hust/ics2019/nanos-lite/src/device.c,35,init_device] Initializing devices...
[/home/hust/ics2019/nanos-lite/src/irq.c,12,init_irq] Initializing interrupt/exception handler...
[/home/hust/ics2019/nanos-lite/src/proc.c,25,init_proc] Initializing processes...
[/home/hust/ics2019/nanos-lite/src/main.c,33,main] Finish initialization
[/home/hust/ics2019/nanos-lite/src/irq.c,5,do_event] system panic: Unhandled event ID = 1
nemu: HIT BAD TRAP at pc = 0x80100528
```

图 4.2 实现用户程序结果

实现 `SYS_yield` 和 `SYS_exit` 系统调用后，再次运行 `dummy` 程序，系统能够正确输出“HIT GOOD TRAP”信息，如图 4.3 所示。

```
Welcome to riscv32-NEMU!
For help, type "help"
[/home/hust/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/ics2019/nanos-lite/src/main.c,15,main] Build time: 15:13:32, Dec 14 2023
[/home/hust/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = , end = , size =
[/home/hust/ics2019/nanos-lite/src/device.c,35,init_device] Initializing devices...
[/home/hust/ics2019/nanos-lite/src/irq.c,19,init_irq] Initializing interrupt/exception handler...
[/home/hust/ics2019/nanos-lite/src/proc.c,25,init_proc] Initializing processes...
[/home/hust/ics2019/nanos-lite/src/loader.c,29,naive_ufload] Jump to entry = 830000C8
Self trap!
nemu: HIT GOOD TRAP at pc = 0x801007b0

[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 599970
make[1]: Leaving directory '/home/hust/ics2019/nemu'
```

图 4.3 实现系统调用结果

继续实现系统调用 `SYS_write` 和 `SYS_brk`，然后把 `Nanos-lite` 上运行的用户程序切换成 `hello` 程序，结果如图 4.4 所示。

```
Welcome to riscv32-NEMU!
For help, type "help"
[/home/hust/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/ics2019/nanos-lite/src/main.c,15,main] Build time: 15:23:52, Dec 14 2023
[/home/hust/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = , end = , size =
[/home/hust/ics2019/nanos-lite/src/device.c,35,init_device] Initializing devices...
[/home/hust/ics2019/nanos-lite/src/irq.c,19,init_irq] Initializing interrupt/exception handler...
[/home/hust/ics2019/nanos-lite/src/proc.c,25,init_proc] Initializing processes...
[/home/hust/ics2019/nanos-lite/src/loader.c,29,naive_ufload] Jump to entry = 83000120
Hello World!
riscv32-nemu: src/isa/riscv32/exec/compute.c:48: exec_I: Assertion `0' failed.
make[1]: *** [Makefile:77: run] Aborted (core dumped)
make[1]: Leaving directory '/home/hust/ics2019/nemu'
make: *** [/home/hust/ics2019/nexus-am/am/arch/platform/nemu.mk:27: run] Error 2
```

图 4.4 hello 运行结果 1

查看报错信息发现，系统中有 `I` 指令未实现而导致程序退出。查看反汇编文件并修改后，能够打印正确信息，如图 4.5 所示。

```

Welcome to riscv32-NEMU!
For help, type "help"
[/home/hust/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/ics2019/nanos-lite/src/main.c,15,main] Build time: 15:26:31, Dec 14 2023
[/home/hust/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = , end = , size = -2146428403 bytes
[/home/hust/ics2019/nanos-lite/src/device.c,35,init_device] Initializing devices...
[/home/hust/ics2019/nanos-lite/src/irq.c,19,init_irq] Initializing interrupt/exception handler...
[/home/hust/ics2019/nanos-lite/src/proc.c,25,init_proc] Initializing processes...
[/home/hust/ics2019/nanos-lite/src/loader.c,29,naive_uoload] 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!

```

图 4.5 hello 运行结果 2

4.2.3 实现文件系统和批处理系统

实现文件操作后，运行测试程序/bin/text，结果如图 4.6 所示，程序成功输出“PASS!!!”的信息。

```

Welcome to riscv32-NEMU!
For help, type "help"
[/home/hust/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/ics2019/nanos-lite/src/main.c,15,main] Build time: 10:44:58, Dec 15 2023
[/home/hust/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = , end = , size = -2146423664 bytes
[/home/hust/ics2019/nanos-lite/src/device.c,55,init_device] Initializing devices...
[/home/hust/ics2019/nanos-lite/src/irq.c,19,init_irq] Initializing interrupt/exception handler...
[/home/hust/ics2019/nanos-lite/src/proc.c,27,init_proc] Initializing processes...
[/home/hust/ics2019/nanos-lite/src/loader.c,49,naive_uoload] Jump to entry = 830002F4
PASS!!!
nemu: HIT GOOD TRAP at pc = 0x80100f58

[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 1669167
make[1]: Leaving directory '/home/hust/ics2019/nemu'

```

图 4.6 text 运行结果

把串口和设备输入抽象成文件后，在 Nanos-lite 加中载/bin/events，结果如图 4.7 所示，无法输出时间事件和按键事件的信息。

```

Welcome to riscv32-NEMU!
For help, type "help"
[/home/hust/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/ics2019/nanos-lite/src/main.c,15,main] Build time: 12:48:19, Dec 15 2023
[/home/hust/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = , end = , size = -2146423664 bytes
[/home/hust/ics2019/nanos-lite/src/device.c,55,init_device] Initializing devices...
[/home/hust/ics2019/nanos-lite/src/irq.c,19,init_irq] Initializing interrupt/exception handler...
[/home/hust/ics2019/nanos-lite/src/proc.c,27,init_proc] Initializing processes...
[/home/hust/ics2019/nanos-lite/src/loader.c,50,naive_uoload] Jump to entry = 83000180
Start to receive events...
receive event: receive event: receive event: receive event: receive event: receive event: receive event: receive event: receive event:
vent: receive event: receive event: receive event: receive event: receive event: receive event: receive event: receive event:
eive event: receive event: receive event: receive event: receive event: receive event: receive event: receive event: receive
t: receive event: receive event: receive event: receive event: receive event: receive event: receive event: receive event:

```

图 4.7 events 运行结果 1

经过调试后发现，由于 PA2 中实现的 `sprintf()` 和 `printf()` 有问题，导致此处的输出出错。修改后的正确结果如图 4.8 所示。

```

Start to receive events...
receive event: kd H
receive event: ku H
receive time event for the 1024th time: t 3642
receive event: kd L
receive event: ku L
receive time event for the 2048th time: t 5163
receive event: kd RETURN
receive event: ku RETURN
receive time event for the 3072th time: t 6495
receive time event for the 4096th time: t 7663
receive time event for the 5120th time: t 8927
[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 10270622

```

图 4.8 events 运行结果 2

把 VGA 显存抽象成文件后，在 Nanos-lite 中加载/bin/bmptest，结果如图 4.9 所示，屏幕上成功显示 Project-N 的 logo。

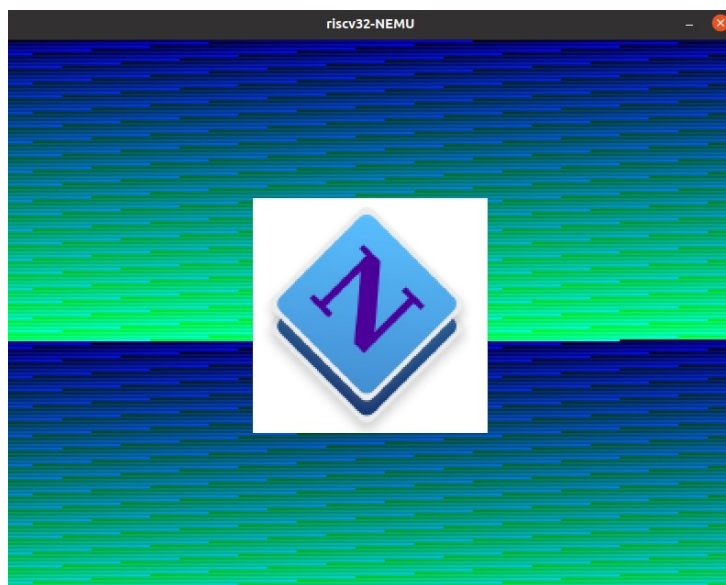


图 4.9 bmptest 运行结果

接着下载仙剑奇侠传的数据文件，并放到 navy-apps/fsimg/share/games/pal/目录下，在 Nanos-lite 中加载并运行/bin/pal，游戏运行效果如图 4.10 所示。



图 4.10 游戏运行结果

最后实现实现开机菜单程序，运行/bin/init 展示批处理系统，结果如图 4.11 所示。



图 4.11 init 运行结果

4.3 实验必答题

问：运行仙剑奇侠传时会播放启动动画，通过 navy-apps/apps/pal/src/main.c 中的 PAL_SplashScreen()函数来播放，像素信息存放在数据文件 mgo.mkf 中。库函数、libos、Nanos-lite、AM、NEMU 是如何相互协助，来帮助仙剑奇侠传的代码从 mgo.mkf 文件中读出仙鹤的像素信息，并且更新到屏幕上的？

答：①读取游戏存档：首先调用 libc 中的 fread()函数，再进行一系列的系统调用，最终传递到_syscall_()进行真正的系统调用。在__am_irq_handle()中，会将其封装成_EVENT_SYSCALL 事件，并转发给 nanos-lite 处理。nanos-lite 在 do_syscall()函数中接收到该事件，根据系统调用号知道是 read 系统调用，从而执行 sys_read()帮助函数，在该函数里会调用真正的文件系统操作函数 fs_read()，并进行真正的文件读取操作。

②更新屏幕：在 fs_write()函数中，判断出要写的文件是/dev/fb 设备文件，接着调用 draw_rect()函数，最终转发给__am_video_write()。在该函数中会执行 out 汇编指令，将数据传送给 VGA 设备中。接收到数据后会保存在定义的显存中，当之后 NDL 库向/dev/fbsync 设备文件中写入时，VGA 设备最终会调用 SDL 库来更新画面。

4.4 心得体会

在本次 PA3 中，我实现了自陷操作 `_yield()`，完成了用户程序的加载和系统调用，并实现了文件系统和批处理系统，最终成功运行了仙剑奇侠传游戏。在完成 Nanos-lite 子项目的过程中，我深刻理解了操作系统是如何与硬件交互，提供支持给用户程序的运行的。

通过实现 `_yield()` 操作，我了解了操作系统中的进程调度机制。这个机制是操作系统负责管理和切换不同进程的重要组成部分，确保每个进程都能有机会执行。同时也更深入理解了自陷的概念和实现方式，尤其是保存上下文、事件分发等阶段的具体过程。

通过实现 `loader()` 等函数，我学会了如何加载用户程序，并将其载入内存执行，重新温习了 ELF 文件格式的解析、加载程序的地址空间等概念。通过实现简单的系统调用，我理解了系统调用的机制，以及如何通过软中断实现用户态和内核态的切换。

通过实现 `fs_open()` 等文件操作函数，实现一个简易的文件系统，我理解了文件系统的组织结构和文件的读写操作，并通过虚拟文件系统，更深刻地体会到“一切皆文件”的思想。

总体而言，通过 PA3 实验，我对操作系统的内部机制有了更加深入的认识，学到了如何与硬件进行交互、管理进程、加载和执行用户程序、实现文件系统等基本概念。

参考文献

- [1] KaiHWang, 王鼎兴. 高等计算机系统结构[M]. 清华大学出版社, 1995.
- [2] 袁春风, 余子濠. 计算机系统基础[M]. 机械工业出版社, 2018.
- [3] 谭志虎. 计算机组成原理[M]. 人民邮电出版社, 2021.