

华中科技大学

2024

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:	CS2010 班
学 号:	U202015606
姓 名:	曾梓桂
电 话:	15342200250
邮 件:	3050057422@qq. com
完成日期:	2024-01-16



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	PA1 – 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	2
2.3	扫描内存	3
2.4	表达式求值	4
2.5	设置监视点	4
2.6	实验结果展示	6
3	PA2 – 简单复杂的机器：冯诺依曼计算机系统	7
3.1	运行第一个 C 程序 dummy、	7
3.2	实现指令，成功通过 cputest	9
3.3	IO 指令及功能的实现	10
3.3.1	串口	10
3.3.2	时钟	11
3.3.3	键盘	12
3.3.4	VGA	13
3.3.5	运行打字游戏	14
4	PA3 – 穿越时空的旅程：批处理系统	15
4.1	实现自陷机制 yield	15
4.1.1	实现新指令	15
4.1.2	重新组织 _Context 结构体	16
4.1.3	实现正确的事件分发	17
4.2	实现系统调用	17
4.3	堆区管理	19
4.4	实现文件系统及其 IO 操作	21
4.4.1	实现完整文件系统	21
4.4.2	把设备输入抽象成文件	21
4.5	把 VGA 抽象为文件	22
5	总结	24
	参考文献	25

1 课程实验概述

1.1 课设目的

计算机系统能力的培养是每位计算机专业学生的核心需求，它涉及到对计算机软硬件的综合理解和应用。为了满足这一需求，系统能力综合培养课程旨在提供一个全面的系统设计实践机会。这个课程的核心是要求深入理解计算机系统的运作机制，并培养在实际操作中运用这些知识的能力。

我选择 RISC-V32 指令模拟器设计作为本次系统能力综合培养的课题。这个选题的主要目标是从底层理解计算机指令集架构，并掌握如何设计和实现一个完整的计算机系统。

为了能够更直观地理解这一原理，课程设计将通过实现一个简化的指令模拟器 NEMU 来完成。NEMU 虽然经过简化，但功能完备，能够模拟指令指令集的执行。在完成 NEMU 后，将进一步在 NEMU 上运行软件，以实际运行程序的方式来探究“程序在计算机上运行”的基本原理。

1.2 课设任务

本课程设计的总体目标是构建一个简化的 RISC-V 模拟器。需要基于已提供的模拟器代码框架，实现 RISC-V 执行代码的解释与执行。该模拟器要求具备以下功能：

完整的输入输出设备支持，确保模拟器与外部环境的交互能力。

异常流处理机制，使模拟器能够妥善处理异常情况，保证程序的正常运行。

精简操作系统的集成，包括文件系统、虚存管理和进程分时调度等功能，以提供更为接近真实计算机环境的模拟体验。

在本次任务中，需要做到深入探究“程序在计算机上运行”的机理。不仅要提高对计算机软硬件协同工作机制的掌握，还要进一步加深对计算机分层系统栈的理解。本次课程设计主要包含下列实验内容。

1. 图灵机与简易调试器
2. 冯诺依曼计算机系统
3. 批处理系统
4. 分时多任务

1.3 实验环境

虚拟机平台：Oracle VM VirtualBox

CPU：单核 CPU

操作系统：Ubuntu 20.04 Desktop

内存大小：1024MB

指令集：RISC-V 32

编译/调试环境：GNU MCU Eclipse RISC-V Embedded GCC v8.1.0-2-20181019

2 PA1 - 开天辟地的篇章: 最简单的计算机

PA0 在本节中，我们将重点实现一个简易的调试器。该调试器已经具备了一些基本功能，包括帮助、继续执行和退出。接下来，我们需要自行开发单步执行、打印程序状态、表达式求值、扫描内存、设置监视点以及删除监视点等功能。

通过这些功能的集成，在后续开发过程中，可以对在模拟器中运行的程序进行调试，及时发现并修正潜在的问题。这个调试器是深入了解和掌握计算机系统运行机制的重要工具。

2.1 单步执行

单步执行的功能是允许用户逐条指令地执行程序，有助于观察程序的执行流程和状态变化。从参数 `args` 中获得单步执行的步数，如果参数为空，则默认只执行一步

```
static int cmd_si(char *args) {
    char* arg=strtok(NULL, " ");
    int num=1;
    if(!(args==NULL)){
        num=atoi(arg);
    }
    cpu_exec(num);
    return 0;
}
```

图表 2-1 单步执行命令代码

2.2 打印程序状态

目前我们实现的 `info` 只需要处理 `r` 参数，也就是只需要打印寄存器状态。实现时调用已经声明的函数 `isa_reg_display()`，在里面直接参照寄存器的结构体使用 `printf` 进行输出就行。

```

static int cmd_info(char *args) {
    char* op = strtok(NULL, " ");
    if (op == NULL) {
        return 0;
    }
    if (strcmp(op, "r") == 0) {
        isa_reg_display();
    }
}

//Mips32
void isa_reg_display() {
    for (int i = 0; i < 8; i++)
    {
        printf("reg%s : %d\n", regsl[i], cpu.gpr[i]);
    }
}

//Riscv32
void isa_reg_display() {
    for (int i = 0; i < 32; i++)
    {
        printf("reg%s : %d\n", regsl[i], cpu.gpr[i]);
    }
}

//X86
void isa_reg_display() {
    for (int i = 0; i < 8; i++)
    {
        printf("reg%s : %d\n", regsl[i], cpu.gpr[i]);
    }
}

```

图表 2-2 打印程序状态

2.3 扫描内存

扫描内存的实现也不难，对命令进行解析之后，先求出表达式的值。但你还没有实现表达式求值的功能，现在可以先实现一个简单的版本：规定表达式 **EXPR** 中只能是一个十六进制数。

暂时不必纠缠于表达式求值的细节。解析出待扫描内存的起始地址之后，就可以使用循环将指定长度的内存数据通过十六进制打印出来。

```

static int cmd_x(char *args) {
    char* n = strtok(NULL, " ");
    char* baseaddr = strtok(NULL, " ");
    int len = 0;
    paddr_t addr = 0;
    sscanf(n, "%d", &len);
    sscanf(baseaddr, "%x", &addr);
    for (int i = 0; i < len; i++)
    {
        printf("%x\n", vaddr_read(addr, 4));
        addr = addr + 4;
    }
    return 0;
}

```

图表 2-3 扫描内存指令代码

2.4 表达式求值

为算术表达式中的各种 token 类型添加规则，需要注意 C 语言字符串中转义字符的存在和正则表达式中元字符的功能。在成功识别出 token 后，将 token 的信息依次记录到 tokens 数组中。

要注意不能忽视了结合性的方向性，否则会导致了在负号里面出现了错误，因为负号的结合律是从右向左的，所以扫描的时候应该是从左向右进行扫描。

在进行计算之前，需要对解引用符号 ‘*’ 的进行额外处理，以区别乘号。当 ‘*’ 前面一个符号已经是运算操作符或者当它们处于第一个运算符时，需要将其更换为解引用类型。以此类推，负数的区分也可通过以这样处理来实现。

完成字符串解析后，我们开始在符号列表上执行表达式求值。为简化描述，这里我们仅考虑合法的表达式。

首先，处理括号：扫描符号列表找到左括号，记下其位置。随后再次扫描，找到与之匹配的右括号。这之间的子符号列表代表一个子表达式，我们递归求值。

去掉括号后，我们得到一个纯数字与运算符的表达式。使用编译原理的相关算法，我们建立符号栈，并交替取出数字和运算符。对于数字，可能有多余的正负号，需统计 ‘*’ 和 ‘-’ 的数量。

考虑运算符的优先级：乘除、加减、逻辑运算。我们三次处理符号栈：每次处理掉当前最高优先级的运算符。例如，乘法时，将栈顶的两个数字相乘并替换。

完成上述步骤后，得到表达式的值。但若表达式错误，例如出现无法继续执行的符号，则将 success 赋值为 0 并退出。

2.5 设置监视点

监视点功能包括创建、删除、输出和检查。为了实现这些功能，我们使用两个单链表来保存监视点信息，一个用于已使用的监视点，另一个用于可用的监视点。每个监视点包含编号、监视表达式字符串以及上次计算的结果。

创建监视点时，我们从可用监视点链表中取出一个空监视点，并将其插入到已使用链表的头部。若没有可用监视点，则给出错误提示。

删除监视点时，我们遍历已使用链表，找到要删除的监视点，将其从链表中移除并放回可用链表。若未找到该监视点，则不做任何操作。

输出监视点时，我们遍历已使用链表，逐个输出监视点的编号和表达式。

最后，检查监视点在每次执行完指令后进行。在 `exec_once` 函数中，完成原有操作后，调用监视点检查函数。该函数遍历所有已使用监视点，对每个监视表达式进行求值。若表达式的值发生变化，则更新该监视点并返回。

```
bool check_wp() {
    WP * p = head;
    bool b = true;
    bool *success = &b;
    int flag = 1;
    while (p != NULL) {
        int nv = expr(p->expr, success);
        if (nv != p->val) {
            flag = 0;
            p->val = nv;
        }
        p = p->next;
    }
    if (flag == 0)
        return false;
    else
        return true;
}
```

图表 2-4 检查监视点代码

```
static int cmd_w(char *args) {
    char exp[MAX_EXPR_LEN];
    strcpy(exp, args);
    bool b = true;
    bool *success = &b;
    int val = expr(exp, success);
    if(b == false) {
        printf("expr is invalid.\n");
        return -1;
    }
    WP *wp = new_wp();
    strcpy(wp->expr, exp);
    wp->val = expr(wp->expr, success);
    return 0;
}

static int cmd_d(char *args) {
    int n;

    sscanf(args, "%d", &n);

    if (free_wp(n)) {
        printf("Watchpoint deleted.\n");
        return 1;
    }

    return 0;
}
```

图表 2-5 创建和删除监视点

2.6 实验结果展示

一开始使用 x86 实现，后续换用 riscv32

```
hust@hust-desktop:~/Documents/pa/ics2019/nemu$ make ISA=x86 run
Building x86-nemu
+ CC src/monitor/debug/ui.c
+ LD build/x86-nemu
make -C /home/hust/Documents/pa/ics2019/nemu/tools/qemu-diff
make[1]: Entering directory '/home/hust/Documents/pa/ics2019/nemu/tools/qemu-diff'
make[1]: Nothing to be done for 'app'.
make[1]: Leaving directory '/home/hust/Documents/pa/ics2019/nemu/tools/qemu-diff'
./build/x86-nemu -l ./build/nemu-log.txt -d /home/hust/Documents/pa/ics2019/nemu/tools/qemu-diff/build/x86-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 [0x00000000, 0x07ffffff]
[src/monitor/monitor.c,20,welcome] Debug: ON
[src/monitor/monitor.c,21,welcome] If debug mode is on, A log file will be generated to record every instruction NEMU executes. This
may lead to a large log file. If it is not necessary, you can turn it off in include/common.h.
[src/monitor/monitor.c,28,welcome] Build time: 17:10:46, Jan 12 2024
Welcome to x86-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 - Run the program step by step
info - Print the status fo the program
x - Scan the memory
p - Calculate expr.
w - Set a watchpoint
d - Delete a watchpoint
(nemu) w $eax
(nemu) w $eax+$ebx
(nemu) w $eax==*200000
(nemu) w 0++
need a number at position 2
0++
^
expr is invalid.
(nemu) info w
No. expr
00 $eax
01 $eax+$ebx
02 $eax==*200000
(nemu) d 2
Watchpoint deleted.
(nemu) info w
No. expr
00 $eax
01 $eax+$ebx
(nemu)
```

图表 2-6 实验结果演示 1 x86

```
may lead to a large log file. If it is not necessary, you can turn it off in include/common.h.
[src/monitor/monitor.c,28,welcome] Build time: 17:10:46, Jan 12 2024
Welcome to x86-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 - Run the program step by step
info - Print the status fo the program
x - Scan the memory
p - Calculate expr.
w - Set a watchpoint
d - Delete a watchpoint
(nemu) w $eax
(nemu) w $eax+$ebx
(nemu) w $eax==*200000
(nemu) w 0++
need a number at position 2
0++
^
need a number at position 2
0++
^
(nemu) info w
No. expr
00 $eax
01 $eax+$ebx
02 $eax==*200000
03 0++
(nemu) q
hust@hust-desktop:~/Documents/pa/ics2019/nemu$ make ISA=x86 run
Building x86-nemu
+ CC src/monitor/debug/ui.c
+ LD build/x86-nemu
make -C /home/hust/Documents/pa/ics2019/nemu/tools/qemu-diff
make[1]: Entering directory '/home/hust/Documents/pa/ics2019/nemu/tools/qemu-diff'
make[1]: Nothing to be done for 'app'.
make[1]: Leaving directory '/home/hust/Documents/pa/ics2019/nemu/tools/qemu-diff'
./build/x86-nemu -l ./build/nemu-log.txt -d /home/hust/Documents/pa/ics2019/nemu/tools/qemu-diff/build/x86-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 [0x00000000, 0x07ffffff]
[src/monitor/monitor.c,20,welcome] Debug: ON
[src/monitor/monitor.c,21,welcome] If debug mode is on, A log file will be generated to record every instruction NEMU executes. This
may lead to a large log file. If it is not necessary, you can turn it off in include/common.h.
[src/monitor/monitor.c,28,welcome] Build time: 17:10:46, Jan 12 2024
Welcome to x86-NEMU!
For help, type "help"
(nemu) w $eax
(nemu) w $eax+$ebx
(nemu) w $eax==*200000
(nemu) w 0++
need a number at position 2
0++
^
expr is invalid.
```

图表 2-7 实验结果演示 2 riscv32

3 PA2 - 简单复杂的机器: 冯诺依曼计算机系统

3.1 运行第一个 C 程序 dummy、

直接运行，自然是失败。make ARCH=\$ISA-nemu ALL=dummy run

```
Welcome to NEMU!
For help, type "help"
Invalid opcode(PC = 0x0010000a): e8 09 00 00 00 90 f3 0f ...

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

If it is the first case, see
386 Manual
For more details.

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

nemu: A000F at pc = 0x0010000a
[src/monitor/cpu-exec.c,29,monitor_statistics] total guest instructions = 3
dummy
host@host-desktop:~/Documents/pa/ics2019/nexus-am/tests/cputest$
```

图表 3-1 第一次运行 dummy

首先查看反汇编文件，通过 dummy-riscv32-nemu.txt 文件，我们可以得知这部分需要实现的指令有：li, auipc, addi, jal, mv, sw, jalr。

```
/home/host/Documents/pa/ics2019/nexus-am/tests/cputest/build/dummy-riscv32-nemu.elf:

Disassembly of section .text:

80100000 <_start>:
80100000: 00000413          li  s0,0
80100004: 00009117          auipc sp,0x9
80100008: ffc10113          addi sp,sp,-4 # 80109000 <_end>
8010000c: 00c000ef          jal ra,80100018 <_trm_init>

Disassembly of section .text.startup:

80100010 <main>:
80100010: 00000513          li  a0,0
80100014: 00008067          ret

Disassembly of section .text._trm_init:

80100018 <_trm_init>:
80100018: 80000537          lui  a0,0x80000
8010001c: ffc10113          addi sp,sp,-16
80100020: 00050513          mv  a0,a0
80100024: 00112623          sw  ra,12(sp)
80100028: fe9ff0ef          jal ra,80100010 <_etext>
8010002c: 00050513          mv  a0,a0
80100030: 0000006b          j    0x6b
80100034: 0000006f          j    80100034 <_trm_init+0x1c>
```

图表 3-2 dummy 反汇编结果

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即可。

这里需要注意的是，实现 RTL 伪指令的时候，尽可能不使用 dest 之外的寄存器存放中间结果。由于 dest 最后会被写入新值，其旧值肯定要覆盖，自然也可以安全地作为 RTL 伪指令的临时寄存器。

通过查询 riscv 文档进行编写代码。

```
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, branch), IDEX(JALR, jalr), EX(nemu_trap), IDEX(JAL, jal), IDEX(SYSTEM, syst
};
```

图表 3-3 opcode_table

```
static inline make_DopHelper(i) {
    op->type = OP_TYPE_IMM;
    op->imm = val;
    rtl_li(&op->val, op->imm);

    print_Dop(op->str, OP_STR_SIZE, "%d", op->imm);
}

static inline make_DopHelper(r) {
    op->type = OP_TYPE_REG;
    op->reg = val;
    if (load_val) {
        rtl_lr(&op->val, op->reg, 4);
    }

    print_Dop(op->str, OP_STR_SIZE, "%s", reg_name(op->reg, 4));
}
```

图表 3-4 辅助译码函数示例

```
make_EHelper(lui) {
    rtl_sr(id_dest->reg, &id_src->val, 4);

    print_asm_template2(lui);
}

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);
}

make_EHelper(i) {
    switch (decinfo.isa.instr.funct3) {
```

图表 3-5 辅助执行函数示例

将指令全部实现后，重新编译运行 dummy，成功 hit good trap。

需要注意的是，ret 指令是一个伪指令，所以不需要实现，通过操作码 01100111 对比可以发现其实是 jalr 指令。

其中实现的所有辅助函数都需要在 decode.h 和 all-instr.h 中声明。

```

dummy
hust@hust-desktop:~/Documents/pa/ics2019/nexus-am/tests/cputest$ make ARCH=riscv32-nemu ALL=dummy run
Makefile:17: warning: overriding recipe for target 'image'
/home/hust/Documents/pa/ics2019/nexus-am/arch/platform/nemu.mk:20: warning: ignoring old recipe for target 'image'
Makefile:18: warning: overriding recipe for target 'run'
/home/hust/Documents/pa/ics2019/nexus-am/arch/platform/nemu.mk:27: warning: ignoring old recipe for target 'run'
# Building dummy [riscv32-nemu] with AM_HOME [/home/hust/Documents/pa/ics2019/nexus-am]
# Building lib-am [riscv32-nemu]
# Building lib-klb [riscv32-nemu]
# Creating binary image [riscv32-nemu]
+ LD -> build/dummy-riscv32-nemu.elf
+ OBJCOPY -> build/dummy-riscv32-nemu.bin
Building riscv32-nemu
+ CC src/cpu/cpu.c
+ CC src/isa/riscv32/decode.c
+ CC src/isa/riscv32/intr.c
+ CC src/isa/riscv32/exec/muldiv.c
+ CC src/isa/riscv32/exec/control.c
+ CC src/isa/riscv32/exec/ldst.c
+ CC src/isa/riscv32/exec/system.c
+ CC src/isa/riscv32/exec/special.c
+ CC src/isa/riscv32/exec/exec.c
+ CC src/isa/riscv32/exec/compute.c
+ LD build/riscv32-nemu
[src/monitor/monitor.c,48,load_img] The image is /home/hust/Documents/pa/ics2019/nexus-am/tests/cputest/build/dummy-riscv32-nemu.bin
[src/memory/memory.c,16,register_pmem] Add 'pmem' at [0x80000000, 0x87ffffff]
[src/monitor/monitor.c,20,welcome] Debug: ON
[src/monitor/monitor.c,21,welcome] If debug mode is on, A log file will be generated to record every instruction NEMU executes. This
may lead to a large log file. If it is not necessary, you can turn it off in include/common.h.
[src/monitor/monitor.c,28,welcome] Build time: 12:32:58, Jan 13 2024
Welcome to riscv32-NEMU!
For help, type "help"
nemu: HIT G000 TRAP at pc = 0x80100030

[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 13
dummy
hust@hust-desktop:~/Documents/pa/ics2019/nexus-am/tests/cputest$

```

图表 3-6 成功执行 dummy

3.2 实现指令，成功通过 cputest

按照之前实现那几个特定指令的步骤，将剩下所有指令依次实现。在实现的过程中会发现很多指令的功能很相似，很多时候只是操作数的形式不同，在实现时要注意区分。最后也是通过除了 string.c 之外的所有测试。

之所以无法通过 string 测试，是因为 string 还要求自己实现相关的 C 语言库函数，而同样需要库函数的 hello-str 能顺利 pass 是因为未实现的函数默认输出 0，歪打正着地通过了测试。

字符串处理函数也都是比较常见的函数，在实现 sprintf 一类的函数时，可以通过 c 语言的数据结构变长数组 va_list 来获取可变参数。

```

int vsprintf(char *buf, const char *fmt, va_list args) {
    char *str;
    int field_width; /*width of output field*/

    for(str=buf;*fmt;*str++){
        unsigned long num;
        int base = 10;
        int flags = 0;
        int qualifier = -1;
        int precision = -1;
        bool bFmt = true;
        if(*fmt != '%'){
            *str++ = *fmt;
            continue;
        }

        bFmt = true;
        while(bFmt){
            fmt++; /*This also skips first '%'*/
            switch(*fmt){
                case '0': flags |= ZEROPAD; break;
                case '-':
                case '+':
                case ' ':
                case '#':
                    _putc('$'); break;
                default: bFmt = false; break;
            }
        }
    }
}

```

图表 3-7 vsprintf 函数部分实现

```
hust@hust-desktop:~/Documents/pa/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!
hust@hust-desktop:~/Documents/pa/ics2019/nemu$
```

图表 3-8 程序测试

3.3 IO 指令及功能的实现

已经成功运行了 `cputest` 中的各个测试用例，但这些测试用例都只能默默地进行纯粹的计算。既然设备也有寄存器，一种最简单的方法就是把设备的寄存器作为接口，让 CPU 来访问这些寄存器。比如 CPU 可以从/往设备的数据寄存器中读出/写入数据，进行数据的输入输出；可以从设备的状态寄存器中读出设备的状态，询问设备是否忙碌；或者往设备的命令寄存器中写入命令字，来修改设备的状态。

框架代码为映射定义了一个结构体类型 `IOMap` (在 `nemu/include/device/map.h` 中定义)，包括名字，映射的起始地址和结束地址，映射的目标空间，以及一个回调函数。然后在 `nemu/src/device/io/map.c` 实现了映射的管理，包括 I/O 空间的分配及其映射，还有映射的访问接口。

3.3.1 串口

定义宏 `HAS_IOE`，运行 `h` 程序。`make ARCH=riscv32-nemu mainargs=h run`

```

/ffff]
[src/device/io/port-io.c,16,add_pio_map] Add port-io map 'keyboard' at [0x00000060, 0x00000063]
[src/device/io/mmio.c,14,add_mmio_map] Add mmio map 'keyboard' at [0xa1000060, 0xa1000063]
[src/monitor/monitor.c,20,welcome] Debug: ON
[src/monitor/monitor.c,21,welcome] If debug mode is on, A log file will be generated to record every instruction NEMU executes. This may lead to a large log file. If it is not necessary, you can turn it off in include/common.h.
[src/monitor/monitor.c,28,welcome] Build time: 11:23:10, Jan 15 2024
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

```

图表 3-9 串口实现

3.3.2 时钟

在 nemu-timer.c 中实现_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。

```

size_t __am_timer_read(uintptr_t reg, void *buf, size_t size) {
    switch (reg) {
        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);
        }
        case _DEVREG_TIMER_DATE: {
            _DEV_TIMER_DATE_t *rtc = (_DEV_TIMER_DATE_t *)buf;
            rtc->second = 0;
            rtc->minute = 0;
            rtc->hour = 0;
            rtc->day = 0;
            rtc->month = 0;
            rtc->year = 2000;
            return sizeof(_DEV_TIMER_DATE_t);
        }
    }
    return 0;
}

```

图表 3-10 实现时钟

```

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).
2000-0-0 00:00:00 GMT (13 seconds).
2000-0-0 00:00:00 GMT (14 seconds).
2000-0-0 00:00:00 GMT (15 seconds).
2000-0-0 00:00:00 GMT (16 seconds).
2000-0-0 00:00:00 GMT (17 seconds).
2000-0-0 00:00:00 GMT (18 seconds).
2000-0-0 00:00:00 GMT (19 seconds).
2000-0-0 00:00:00 GMT (20 seconds).
2000-0-0 00:00:00 GMT (21 seconds).
2000-0-0 00:00:00 GMT (22 seconds).
2000-0-0 00:00:00 GMT (23 seconds).
2000-0-0 00:00:00 GMT (24 seconds).
2000-0-0 00:00:00 GMT (25 seconds).
2000-0-0 00:00:00 GMT (26 seconds).
2000-0-0 00:00:00 GMT (27 seconds).

```

图表 3-11 成功实现时钟

3.3.3 键盘

通过 `inl(KBD_ADDR)` 从 MMIO 中获取键盘码，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。

```

[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 14634
make[1]: Leaving directory '/home/hust/Documents/pa/ics2019/nemu'
hust@hust-desktop:~/Documents/pa/ics2019/nexus-am/tests/antest$ make ARCH=riscv32-nemu mainargs=k run
# Building antest [riscv32-nemu] with AM_HOME {/home/hust/Documents/pa/ics2019/nexus-am}
# Building lib-am [riscv32-nemu]
# Building lib-klib [riscv32-nemu]
# Creating binary image [riscv32-nemu]
+ LD -> build/antest-riscv32-nemu.elf
+ OBJCOPY -> build/antest-riscv32-nemu.bin
make -C /home/hust/Documents/pa/ics2019/nemu ISA=riscv32 run ARGS="-b -a k -l /home/hust/Documents/pa/ics2019/nexus-am/build/nemu-log.txt /home/hust/Documents/pa/ics2019/nexus-am/tests/antest/build/antest-riscv32-nemu.bin"
make[1]: Entering directory '/home/hust/Documents/pa/ics2019/nemu'
Building riscv32-nemu
make -C /home/hust/Documents/pa/ics2019/nemu/tools/qemu-diff
make[2]: Entering directory '/home/hust/Documents/pa/ics2019/nemu/tools/qemu-diff'
make[2]: Nothing to be done for 'app'.
make[2]: Leaving directory '/home/hust/Documents/pa/ics2019/nemu/tools/qemu-diff'
./build/riscv32-nemu -b -a k -l /home/hust/Documents/pa/ics2019/nexus-am/tests/antest/build/nemu-log.txt /home/hust/Documents/pa/ics2019/nexus-am/tests/antest/build/antest-riscv32-nemu.bin -d /home/hust/Documents/pa/ics2019/nemu/tools/qemu-diff/build
so
[src/monitor/monitor.c,48,load_img] The image is /home/hust/Documents/pa/ics2019/nexus-am/tests/antest/build/antest-riscv32-nemu.bin
[src/memory/memory.c,16,register_pmen] Add 'pmem' at [0x80000000, 0x87ffffff]
[src/device/io/port-io.c,16,add_pio_map] Add port-io map 'serial' at [0x000003f8, 0x000003f8]
[src/device/io/mmio.c,14,add_mmio_map] Add mmio map 'serial' at [0xa10003f8, 0xa10003f8]
[src/device/io/port-io.c,16,add_pio_map] Add port-io map 'rtc' at [0x00000048, 0x0000004b]
[src/device/io/mmio.c,14,add_mmio_map] Add mmio map 'rtc' at [0xa1000048, 0xa100004b]
[src/device/io/port-io.c,16,add_pio_map] Add port-io map 'screen' at [0x00000100, 0x00000107]
[src/device/io/mmio.c,14,add_mmio_map] Add mmio map 'screen' at [0xa1000100, 0xa1000107]
[src/device/io/mmio.c,14,add_mmio_map] Add mmio map 'vmem' at [0xa0000000, 0xa007ffff]
[src/device/io/port-io.c,16,add_pio_map] Add port-io map 'keyboard' at [0x00000060, 0x00000063]
[src/device/io/mmio.c,14,add_mmio_map] Add mmio map 'keyboard' at [0xa1000060, 0xa1000063]
[src/monitor/monitor.c,25,welcome] Debug: OFF
Welcome to riscv32-NEMU!
For help, type "help"
Try to press any key...
Get key: 46 F down
Get key: 46 F up
Get key: 45 D down
Get key: 45 D up
Get key: 48 H down
Get key: 48 H up
Get key: 36 I down
Get key: 36 I up
Get key: 18 4 down
Get key: 18 4 up
Get key: 19 5 down
Get key: 19 5 up
Get key: 21 7 down
Get key: 21 7 up
Get key: 55 LSHIFT down
Get key: 55 LSHIFT up
Get key: 29 Q down
Get key: 29 Q up
[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 1557285272
make[1]: Leaving directory '/home/hust/Documents/pa/ics2019/nemu'

```

图表 3-12 成功实现键盘

3.3.4 VGA

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。实现 `vga_io_handler` 函数，函数中在需要的时候更新一下屏幕就好。

```
#include <am.h>
#include <amdev.h>
#include <nemu.h>

size_t _am_video_read(uintptr_t reg, void *buf, size_t size) {
    switch (reg) {
        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);
        }
    }
    return 0;
}

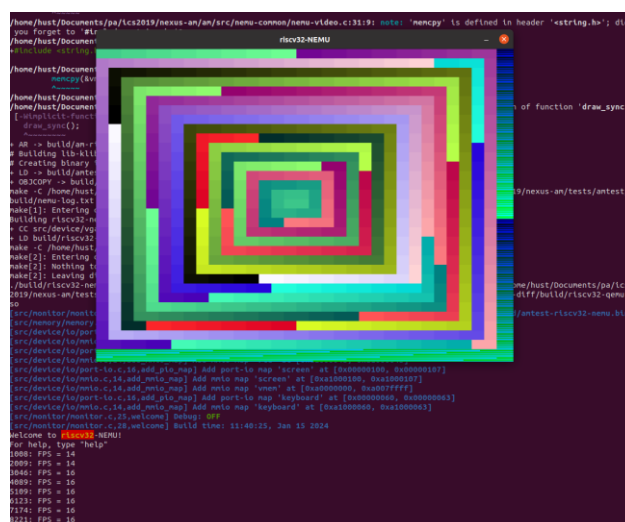
size_t _am_video_write(uintptr_t reg, void *buf, size_t size) {
    switch (reg) {
        case DEVREG_VIDEO_FBCTL: {
            DEV_VIDEO_FBCTL_t *ctl = (DEV_VIDEO_FBCTL_t *)buf;
            uint32_t *pixels = ctl->pixels;
            int x = ctl->x, y = ctl->y, w = ctl->w, h = ctl->h;

            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 *)FB_ADDR; // video
            for (int i = 0; i < h && y + i < H; i++) {
                memcpy(&vmem[(y + i) * W + x], pixels, copy_bytes);
                pixels += w;
            }

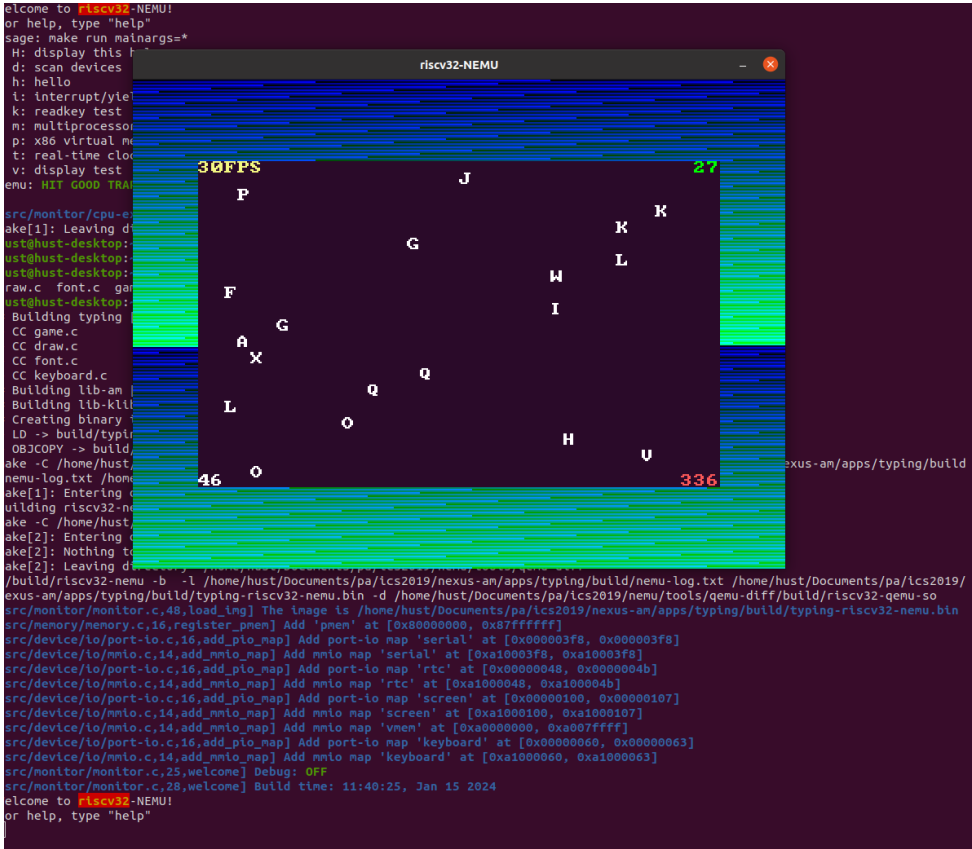
            if (ctl->sync) {
                outl(SYNC_ADDR, 0);
            }
            return size;
        }
    }
    return 0;
}
```

图表 3-13 实现 vga 的 io 功能



图表 3-14 vga 测试通过

3.3.5 运行打字游戏



图表 3-15 打字游戏

4 PA3 - 穿越时空的旅程: 批处理系统

riscv32 提供 `ecall` 指令作为自陷指令, 并提供一个 `stvec` 寄存器来存放异常入口地址. 为了保存程序当前的状态, riscv32 提供了一些特殊的系统寄存器, 叫控制状态寄存器(CSR 寄存器). 在 PA 中, 我们只使用如下 3 个 CSR 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

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

首先是按照 ISA 的约定来设置异常入口地址, 将来切换执行流时才能跳转到正确的异常入口. 需要在 `nanos-lite/include/common.h` 中定义宏 `HAS_CTE`, 这样以后, Nanos-lite 会多进行一项初始化工作: 调用 `init_irq()` 函数, 这最终会调用位于 `nexus-am/am/src/$ISA/nemu/cte.c` 中的 `_cte_init()` 函数. `_cte_init()` 函数会做两件事情, 第一件就是设置异常入口地址, 对于 riscv32 来说, 直接将异常入口地址设置到 `stvec` 寄存器中即可. `_cte_init()` 函数做的第二件事是注册一个事件处理回调函数。

4.1 实现自陷机制 yield

4.1.1 实现新指令

按照 pa2 的方式实现 `ecall` 指令, `sret` 指令和一系列与控制状态寄存器相关的指令, 填写 `opcode_table`, 编写辅助译码函数和辅助执行函数。

这些指令的格式都是一致的。

根据 `funct3` 的不同来区分不同的指令。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。

```

make_EHelper(system){
    Instr instr = decinfo.isa.instr;
    switch(instr.funct3){
        case 0b0:
            if ((instr.val & ~(0x7f)) == 0) {
                raise_intr(reg_l(17), cpu.pc);
            } else if (instr.val == 0x10200073) { //
                decinfo.jmp_pc = decinfo.isa.sepc + 4;
                rtl_j(decinfo.jmp_pc);
            } else {
                assert(!"Undo the system opcode");
            }
            break;
        case 0b001: // csrrw
            s0 = get_csr(instr.csr);
            write_csr(instr.csr, id_src->val);
            rtl_sr(id_dest->reg, &s0, 4);
            print_asm_template3(csrrw);
            break;
        case 0b010: // csrrs
            s0 = get_csr(instr.csr);
            write_csr(instr.csr, s0 | id_src->val);
            rtl_sr(id_dest->reg, &s0, 4);
            print_asm_template3(csrrs);
            break;
        case 0b011: // csrrc
            s0 = get_csr(instr.csr);
            write_csr(instr.csr, s0 & ~id_src->val);
            rtl_sr(id_dest->reg, &s0, 4);
            print_asm_template3(csrrc);
            break;
        case 0b101: // csrrwi
            s0 = get_csr(instr.csr);
            write_csr(instr.csr, id_src->reg);
            rtl_sr(id_dest->reg, &s0, 4);
            print_asm_template3(csrrwi);
            break;
        case 0b110: // csrrsi
            s0 = get_csr(instr.csr);
            write_csr(instr.csr, s0 | id_src->reg);
            rtl_sr(id_dest->reg, &s0, 4);
            print_asm_template3(csrrsi);
            break;
    }
}

```

图表 4-1 辅助执行函数部分实现

4.1.2 重新组织_Context 结构体

看了 trap.S 的源码之后很容易知道，先压栈的是 32 个通用寄存器，然后是 scause，然后是 sstatus，然后是 sepc。

```

struct _Context {
    uintptr_t gpr[32], cause, status, epc;
    struct _AddressSpace *as;
};
#define GPR1 gpr[17]
#define GPR2 gpr[10]
#define GPR3 gpr[11]
#define GPR4 gpr[12]
#define GPRx gpr[10]

```

图表 4-2 Context 结构体

打印出来，正确。

```

[src/monitor/monitor.c:48,load_img] The image is /home/hust/Documents/pa/ics2019/nanos-lite/build/nanos-lite-riscv32-nemu.bin
[src/memory/memory.c:16,register_pmem] Add 'pmem' at [0x00000000, 0x07ffffff]
[src/device/io/port-io.c:16,add_pio_map] Add port-io map 'serial' at [0x000003f8, 0x000003f8]
[src/device/io/mmio.c:14,add_mmio_map] Add mmio map 'serial' at [0x000003f8, 0x000003f8]
[src/device/io/port-io.c:16,add_pio_map] Add port-io map 'rtc' at [0x00000048, 0x0000004b]
[src/device/io/mmio.c:14,add_mmio_map] Add mmio map 'rtc' at [0x00000048, 0x0000004b]
[src/device/io/port-io.c:16,add_pio_map] Add port-io map 'screen' at [0x00000100, 0x00000107]
[src/device/io/mmio.c:14,add_mmio_map] Add mmio map 'screen' at [0x00000100, 0x00000107]
[src/device/io/mmio.c:14,add_mmio_map] Add mmio map 'vmem' at [0x00000000, 0x007fffff]
[src/device/io/mmio.c:16,add_pio_map] Add port-io map 'keyboard' at [0x00000060, 0x00000063]
[src/device/io/mmio.c:14,add_mmio_map] Add mmio map 'keyboard' at [0x00000060, 0x00000063]
[src/monitor/monitor.c:25,welcome] Debug: OFF
[src/monitor/monitor.c:28,welcome] Build time: 15:20:55, Jan 15 2024
Welcome to riscv32-NEMU!
For help, type "help"
[/home/hust/Documents/pa/ics2019/nanos-lite/src/main.c:14,main] 'Hello World!' from Nanos-lite
[/home/hust/Documents/pa/ics2019/nanos-lite/src/main.c:15,main] Build time: 22:00:01, Jan 15 2024
$$[/home/hust/Documents/pa/ics2019/nanos-lite/src/ramdisk.c:28,init_ramdisk] ramdisk info: start = 2148538925, end = 2148566681, size = 27756 bytes
[/home/hust/Documents/pa/ics2019/nanos-lite/src/device.c:35,init_device] Initializing devices...
[/home/hust/Documents/pa/ics2019/nanos-lite/src/irq.c:12,init_irq] Initializing interrupt/exception handler...
[/home/hust/Documents/pa/ics2019/nanos-lite/src/proc.c:25,init_proc] Initializing processes...
[/home/hust/Documents/pa/ics2019/nanos-lite/src/main.c:33,main] Finish initialization
ffffffff 0 80100ab0
[/home/hust/Documents/pa/ics2019/nanos-lite/src/irq.c:5,do_event] system panic: Unhandled event ID = 5
nemu: HIT BAD TRAP at pc = 0x80100528

[src/monitor/cpu-exec.c:29,monitor_statistic] total guest instructions = 506782
make[1]: Leaving directory '/home/hust/Documents/pa/ics2019/nemu'
hust@hust-desktop:~/Documents/pa/ics2019/nanos-lite$

```

图表 4-3 _Context 结构体正确

4.1.3 实现正确的事件分发

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。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。

重新运行，通过。

```

Welcome to riscv32-NEMU!
For help, type "help"
[/home/hust/Documents/pa/ics2019/nanos-lite/src/main.c:14,main] 'Hello World!' from Nanos-lite
[/home/hust/Documents/pa/ics2019/nanos-lite/src/main.c:15,main] Build time: 22:03:04, Jan 15 2024
$$[/home/hust/Documents/pa/ics2019/nanos-lite/src/ramdisk.c:28,init_ramdisk] ramdisk info: start = 2148539201, end = 2148566957, size = 27756 bytes
[/home/hust/Documents/pa/ics2019/nanos-lite/src/device.c:35,init_device] Initializing devices...
[/home/hust/Documents/pa/ics2019/nanos-lite/src/irq.c:21,init_irq] Initializing interrupt/exception handler...
[/home/hust/Documents/pa/ics2019/nanos-lite/src/proc.c:25,init_proc] Initializing processes...
[/home/hust/Documents/pa/ics2019/nanos-lite/src/main.c:33,main] Finish initialization
ffffffff 0 80100b28
Self trap!
[/home/hust/Documents/pa/ics2019/nanos-lite/src/main.c:39,main] system panic: Should not reach here
nemu: HIT BAD TRAP at pc = 0x801005a0

[src/monitor/cpu-exec.c:29,monitor_statistic] total guest instructions = 507702
make[1]: Leaving directory '/home/hust/Documents/pa/ics2019/nemu'
hust@hust-desktop:~/Documents/pa/ics2019/nanos-lite$

```

图表 4-4 成功实现 yield 自陷操作

4.2 实现系统调用

与前面自陷操作的总体流程是相似的。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* __am_irq_handle(_Context *c) {
    // printf("%x %x %x\n", c->cause, c->status, c->epc);
    _Context *next = c;

    if (user_handler) {
        _Event ev = {0};
        switch (c->cause) {
            case -1: // 自陷异常
                ev.event = _EVENT_YIELD;
                break;
            // 系统调用
            case 0: // SYS_exit
            case 1: // SYS_yield
            case 2: // SYS_open
            case 3: // SYS_read
            case 4: // SYS_write
            case 7: // SYS_close
            case 8: // SYS_lseek
            case 9: // SYS_brk
            case 13: // SYS_execve
                ev.event = _EVENT_SYSCALL;
                break;
            default: ev.event = _EVENT_ERROR; break;
        }

        next = user_handler(ev, c);
        if (next == NULL) {
            next = c;
        }
    }

    return next;
}

```

图表 4-5 添加系统调用

```

_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;
        case SYS_read:
            c->GPRx = sys_read(a[1], (void*)(a[2]), a[3]);
            break;
        case SYS_lseek:
            c->GPRx = sys_lseek(a[1], a[2], a[3]);
            break;
        case SYS_open:
            c->GPRx = sys_open((const char *)a[1], a[2], a[3]);
            break;
        case SYS_close:
            c->GPRx = sys_close(a[1]);
            break;
        case SYS_execve:
            c->GPRx = sys_execve(a[1], a[2], a[3]);
            break;
        default: panic("Unhandled syscall ID = %d", a[0]);
    }

    return NULL;
}

```

图表 4-6 系统调用实现

这里需要特别提到的是，要在文件 `nanos.c` 修改唤起系统调用的函数，通过反汇编才发现这里的函数也需要修改。

```
intptr_t _syscall (intptr_t type, intptr_t a0, intptr_t a1, intptr_t a2) {
    register intptr_t _gpr1 asm (GPR1) = type;
    register intptr_t _gpr2 asm (GPR2) = a0;
    register intptr_t _gpr3 asm (GPR3) = a1;
    register intptr_t _gpr4 asm (GPR4) = a2;
    register intptr_t ret asm (GPRx);
    asm volatile (SYSCALL : "=r" (ret) : "r"(_gpr1), "r"(_gpr2), "r"(_gpr3), "r"(_gpr4));
    return ret;
}

void _exit(int status) {
    _syscall (SYS_exit, status, 0, 0);
    while (1);
}

int _open(const char *path, int flags, mode_t mode) {
    return _syscall (SYS_open, (intptr_t)path, flags, mode);
}

int _write(int fd, void *buf, size_t count) {
    return _syscall (SYS_write, fd, (intptr_t)buf, count);
}
```

图表 4-7 nanos.c 系统调用修改

```
Welcome to riscv32-NEWU!
For help, type "help"
[/home/hust/Documents/pa/lcs2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/Documents/pa/lcs2019/nanos-lite/src/main.c,15,main] Build time: 11:37:31, Jan 16 2024
SS[/home/hust/Documents/pa/lcs2019/nanos-lite/src/randisk.c,28,init_randisk] randisk info: start = 2148541756, end = 2148571356, size = 29600 bytes
[/home/hust/Documents/pa/lcs2019/nanos-lite/src/device.c,35,init_device] Initializing devices...
[/home/hust/Documents/pa/lcs2019/nanos-lite/src/irq.c,22,init_irq] Initializing interrupt/exception handler...
[/home/hust/Documents/pa/lcs2019/nanos-lite/src/proc.c,25,init_proc] Initializing processes...
[/home/hust/Documents/pa/lcs2019/nanos-lite/src/loader.c,49,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!
Hello World from Navy-apps for the 6th time!
Hello World from Navy-apps for the 7th time!
Hello World from Navy-apps for the 8th time!
Hello World from Navy-apps for the 9th time!
Hello World from Navy-apps for the 10th time!
Hello World from Navy-apps for the 11th time!
Hello World from Navy-apps for the 12th time!
Hello World from Navy-apps for the 13th time!
Hello World from Navy-apps for the 14th time!
Hello World from Navy-apps for the 15th time!
Hello World from Navy-apps for the 16th time!
Hello World from Navy-apps for the 17th time!
Hello World from Navy-apps for the 18th time!
Hello World from Navy-apps for the 19th time!
Hello World from Navy-apps for the 20th time!
Hello World from Navy-apps for the 21th time!
Hello World from Navy-apps for the 22th time!
Hello World from Navy-apps for the 23th time!
```

图表 4-8 系统调用成功实现

4.3 堆区管理

在 Navy-apps 的 Newlib 中，`sbrk()` 最终会调用 `_sbrk()`，它在 `navy-apps/libs/libos/src/nanos.c` 中定义。框架代码让 `_sbrk()` 总是返回 -1，表示堆区调整失败，事实上，用户程序在第一次调用 `printf()` 的时候会尝试通过 `malloc()` 申请一片缓冲区，来存放格式化的内容。若申请失败，就会逐个字符进行输出。如果你在 Nanos-lite 中的 `sys_write()` 中通过 `Log()` 观察其调用情况，你会发现用户程序通过 `printf()` 输出的时候，确实是逐个字符地调用 `write()` 来输出的。

但如果堆区总是不可用，Newlib 中很多库函数的功能将无法使用，因此现在你需要实现 `_sbrk()` 了。为了实现 `_sbrk()` 的功能，我们还需要提供一个用于设置堆

区大小的系统调用。在 GNU/Linux 中，这个系统调用是 `SYS_brk`，它接收一个参数 `addr`，用于指示新的 program break 的位置。`_sbrk()` 通过记录的方式来对用户程序的 program break 位置进行管理，其工作方式如下：

program break 一开始的位置位于 `_end`

被调用时，根据记录的 program break 位置和参数 `increment`，计算出新 program break

通过 `SYS_brk` 系统调用来让操作系统设置新 program break

若 `SYS_brk` 系统调用成功，该系统调用会返回 0，此时更新之前记录的 program break 的位置，并将旧 program break 的位置作为 `_sbrk()` 的返回值返回

若该系统调用失败，`_sbrk()` 会返回 -1。

```
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;
}
```

图表 4-9 支持堆区管理的 loader 函数

修改后文字可以整句输出，说明堆区管理生效。

```
Welcome to rtscv32-NEMU!
For help, type "help"
[/home/hust/Documents/pa/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/Documents/pa/ics2019/nanos-lite/src/main.c,15,main] Build time: 11:46:28, Jan 16 2024
$$[/home/hust/Documents/pa/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = 2148541724, end = 2148571324, size = 29600 bytes
[/home/hust/Documents/pa/ics2019/nanos-lite/src/device.c,35,init_device] Initializing devices...
[/home/hust/Documents/pa/ics2019/nanos-lite/src/irq.c,22,init_irq] Initializing interrupt/exception handler...
[/home/hust/Documents/pa/ics2019/nanos-lite/src/proc.c,25,init_proc] Initializing processes...
[/home/hust/Documents/pa/ics2019/nanos-lite/src/loader.c,49,naive_uload] Jump to entry = 83000120
[/home/hust/Documents/pa/ics2019/nanos-lite/src/syscall.c,27,sys_write] 1
Hello World!
[/home/hust/Documents/pa/ics2019/nanos-lite/src/syscall.c,27,sys_write] 1
Hello World from Navy-apps for the 2th time!
[/home/hust/Documents/pa/ics2019/nanos-lite/src/syscall.c,27,sys_write] 1
Hello World from Navy-apps for the 3th time!
[/home/hust/Documents/pa/ics2019/nanos-lite/src/syscall.c,27,sys_write] 1
Hello World from Navy-apps for the 4th time!
[/home/hust/Documents/pa/ics2019/nanos-lite/src/syscall.c,27,sys_write] 1
```

图表 4-10 实现堆区管理

4.4 实现文件系统及其 IO 操作

4.4.1 实现完整文件系统

实现 `fs_open()`, `fs_read()`, `fs_close()`, `fs_write()` 和 `fs_lseek()` 函数。
先实现文件信息数据结构。

```
typedef struct {
    char *name;
    size_t size;
    size_t disk_offset;
    size_t open_offset;
    ReadFn read;
    WriteFn write;
} Finfo;
```

图表 4-11 Finfo 实验数据结构

在写相关函数时要注意边界问题。读写函数容易引发越界问题。
实现后运行测试程序 `/bin/text`，成功通过。

```
Welcome to riscv32-NEMU!
For help, type "help"
[/home/hust/Documents/pa/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/Documents/pa/ics2019/nanos-lite/src/main.c,15,main] Build time: 12:38:11, Jan 16 2024
$$[/home/hust/Documents/pa/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = 2148542668, end = 2149441586, size = 898918 bytes
[/home/hust/Documents/pa/ics2019/nanos-lite/src/device.c,35,init_device] Initializing devices...
[/home/hust/Documents/pa/ics2019/nanos-lite/src/irq.c,22,init_irq] Initializing interrupt/exception handler...
[/home/hust/Documents/pa/ics2019/nanos-lite/src/proc.c,25,init_proc] Initializing processes...
[/home/hust/Documents/pa/ics2019/nanos-lite/src/loader.c,36,naive_uload] Jump to entry = 830002f4
nemu: HIT GOOD TRAP at pc = 0x80100ce4

[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 1677258
make[1]: Leaving directory '/home/hust/Documents/pa/ics2019/nemu'
hust@hust-desktop:~/Documents/pa/ics2019/nanos-lite$
```

图表 4-12 文件测试通过

4.4.2 把设备输入抽象成文件

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。

`/bin/events`

```
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;
}

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;
}
```

图表 4-13 实现将设备输入抽象为文件

成功实现后，运行测试文件。

```
Welcome to riscv32-NEMU!  
For help, type "help"  
[/home/hust/Documents/pa/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite  
[/home/hust/Documents/pa/ics2019/nanos-lite/src/main.c,15,main] Bulld time: 12:44:54, Jan 16 2024  
$$[/home/hust/Documents/pa/ics2019/nanos-lite/src/randisk.c,28,init_randisk] randisk info: start = 2148544104, end = 2149443022, size = 898918 bytes  
[/home/hust/Documents/pa/ics2019/nanos-lite/src/device.c,46,init_device] Initializing devices...  
[/home/hust/Documents/pa/ics2019/nanos-lite/src/irq.c,22,init_irq] Initializing interrupt/exception handler...  
[/home/hust/Documents/pa/ics2019/nanos-lite/src/proc.c,25,init_proc] Initializing processes...  
[/home/hust/Documents/pa/ics2019/nanos-lite/src/loader.c,36,naive_uload] Jump to entry = 83000180  
Start to receive events...  
receive time event for the 1024th time: t 68  
receive time event for the 2048th time: t 120  
receive time event for the 3072th time: t 180  
receive time event for the 4096th time: t 233  
receive time event for the 5120th time: t 289  
receive time event for the 6144th time: t 338  
receive time event for the 7168th time: t 398  
receive event: kd F  
receive time event for the 8192th time: t 461  
receive time event for the 9216th time: t 517  
receive event: kd D  
receive time event for the 10240th time: t 577  
receive event: kd S  
receive time event for the 11264th time: t 619  
receive event: ku F  
receive time event for the 12288th time: t 661  
receive time event for the 13312th time: t 703  
receive time event for the 14336th time: t 742
```

图表 4-14 事件测试通过

4.5 把 VGA 抽象为文件

在 `init_fs()`(在 `nanos-lite/src/fs.c` 中定义)中对文件记录表中 `/dev/fb` 的大小进行初始化。

实现 `fb_write()`(在 `nanos-lite/src/device.c` 中定义), 用于把 `buf` 中的 `len` 字节写到屏幕上 `offset` 处. 你需要先从 `offset` 计算出屏幕上的坐标, 然后调用 IOE 的 `draw_rect()`.

实现 `fbsync_write()`(在 `nanos-lite/src/device.c` 中定义), 直接调用 IOE 的相应 API 即可.

在 `init_device()`(在 `nanos-lite/src/device.c` 中定义)中将 `/proc/dispinfo` 的内容提前写入到字符串 `dispinfo` 中.

实现 `dispinfo_read()`(在 `nanos-lite/src/device.c` 中定义), 用于把字符串 `dispinfo` 中 `offset` 开始的 `len` 字节写到 `buf` 中.

在 VFS 中添加对 `/dev/fb`, `/dev/fbsync` 和 `/proc/dispinfo` 这三个特殊文件的支持.


```
home/hust/Documents/pa/ics2019/nanos-lite/src/syscall.c: In function 'sys_execve':
home/hust/Documents/pa/ics2019/nanos-lite/src/syscall.c:11:1: error: implicit declaration of function [-Wreturn-type]
}
^
AS src/initrd.S
CC src/mm.c
CC src/main.c
CC src/proc.c
/home/hust/Documents/pa/ics2019/nanos-lite/src/syscall.c:11:1: error: implicit declaration of function [-Wreturn-type]
/home/hust/Documents/pa/ics2019/nanos-lite/src/syscall.c:11:1: error: implicit declaration of function [-Wreturn-type]
ion-declaration]
naive_uload(NULL)
CC src/loader.c
CC src/randisk.c
CC src/irq.c
Building lib-am
Building lib-kl
Creating binary
LD -o build/nanos-lite-riscv32-nemu
OBJCOPY -O binary build/nanos-lite-riscv32-nemu
make -C /home/hust/Documents/pa/ics2019/nanos-lite/build/nemu-log.
make[1]: Entering directory '/home/hust/Documents/pa/ics2019/nanos-lite/build/nemu-log.'
Building riscv32-nemu
make -C /home/hust/Documents/pa/ics2019/nanos-lite/build/nemu-log.
make[2]: Entering directory '/home/hust/Documents/pa/ics2019/nanos-lite/build/nemu-log.'
make[2]: Nothing to be done for 'all'.
make[2]: Leaving directory '/home/hust/Documents/pa/ics2019/nanos-lite/build/nemu-log.'
/build/riscv32-nemu
/build/nanos-lite-riscv32-nemu
src/monitor/monitor.c:25, welcome] Debug: OFF
src/monitor/monitor.c:28, welcome] Build time: 15:20:55, Jan 15 2024
Welcome to riscv32-NEMU!
For help, type "help"
/home/hust/Documents/pa/ics2019/nanos-lite/src/main.c:14,main] 'Hello World!' from Nanos-lite
/home/hust/Documents/pa/ics2019/nanos-lite/src/main.c:15,main] Build time: 12:48:16, Jan 16 2024
$[ /home/hust/Documents/pa/ics2019/nanos-lite/src/randisk.c:28,init_randisk] randisk info: start = 2148544380, end = 2149443298, siz
e = 898918 bytes
/home/hust/Documents/pa/ics2019/nanos-lite/src/device.c:55,init_device] Initializing devices...
```

图表 4-15 vga 测试成功

5 总结

在这次课程设计中，我基于已经设计好的模拟器代码框架，成功构建了一个简易调试器。这个调试器具备了执行所有必要 RISC-V 指令的功能。同时，我也测试了 I/O 指令，并以此为基础实现了一个有趣的打字小游戏。更进一步地，我还实现了系统调用和文件系统，确保模拟器能够流畅运行。最终，在模拟器上成功运行了经典游戏《仙剑奇侠传》。

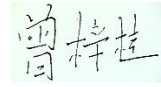
尽管这个课程设计的难度相当大，也耗费了我大量的时间，但整个过程也让我收获颇丰。对于 pa1 阶段实现的简易调试器，其作为热身项目相对简单，没有遇到特别棘手的问题，主要是对代码的熟悉与掌握。然而，到了 pa2 阶段，实现相关指令的任务变得异常艰巨。由于这部分内容涉及到底层实现，调试过程相当抽象和繁琐。尤其是在满屏的宏定义中，阅读和理解代码都变得异常困难。每一步实现都需要细心和耐心，一旦出错就需要长时间的调试。而到了后续的输入输出以及 pa3 阶段，难度相对有所降低。这些部分逻辑更易理解，实现起来也更为直观。文档中的指导也相当详尽。

尽管在做这个课程设计时遇到了许多挑战，但我也从中获得了许多宝贵的经验。例如，我深入了解了 gdb 调试的原理和用法，学会了各种宏定义的技巧，掌握了系统调用的实现方式，甚至用 C 语言模拟了面向对象编程。此外，通过逐条指令调试来查找错误的过程也锻炼了我的细心和耐心。

最后，我非常高兴能在大学的最后一段时间里参与这样一个完整而丰富的 NEMU 项目。这个项目不仅加深了我对计算机分层系统栈的理解，还梳理了大学三年所学的全部理论知识，极大地提升了我的计算机系统能力。

参考文献

电子签名:

A handwritten signature in black ink on a light green rectangular background. The signature consists of three Chinese characters: '曾' (Zeng), '祥' (Xiang), and '桂' (Gui), written in a cursive style.