华中科技大学

2025

系统能力培养　课程实验报告

题　　目：	riscv32 指令模拟器设计
专　　业：	计算机科学与技术
班　　级：	CS2209 班
学　　号：	U202210755
姓　　名：	章宏宇
电　　话：	13395057812
邮　　件：	1318392380@qq.com
完成日期：	2026-01-15

计算机科学与技术学院

目 录

1 课程实验概述 1

1.1 课设目的 1
1.2 实验环境 1

2 实验方案设计 2

2.1 PA1 – 开天辟地的篇章：最简单的计算机 2
2.1.1 PA1.1: 实现单步执行, 打印寄存器状态, 扫描内存 2
2.1.2 PA1.2: 实现算术表达式求值 3
2.1.3 PA1.3: 实现所有要求, 提交完整的实验报告 5
2.1.4 必答题 6
2.2 PA2 – 简单复杂的机器：冯•诺伊曼计算机系统 9
2.2.1 PA2.1: 在 NEMU 中运行第一个 C 程序 dummy 9
2.2.2 PA2.2: 实现更多的指令, 在 NEMU 中运行所有 cputest 9
2.2.3 PA2.3: 运行打字小游戏, 提交完整的实验报告 11
2.2.4 必答题 13
2.3 PA3 - 穿越时空的旅程: 批处理系统 15
2.3.1 PA3.1: 实现自陷操作_yield()及其过程 15
2.3.2 PA3.2: 实现用户程序加载和系统调用, 支撑 TRM 程序运行 16
2.3.3 PA3.3: 运行仙剑奇侠传展示批处理系统, 提交完整实验报告 16
2.3.4 必答题 17

3 实验结果与结果分析 19

3.1 PA1 – 开天辟地的篇章: 最简单的计算机 19
3.1.1 PA1.1: 实现单步执行, 打印寄存器状态, 扫描内存 19
3.1.2 PA1.2: 实现算术表达式求值 20
3.1.3 PA1.3: 实现所有要求, 提交完整的实验报告 21
3.2 PA2 - 简单复杂的机器: 冯诺依曼计算机系统 22
3.2.1 PA2.1: 在 NEMU 中运行第一个 C 程序 dummy 22
3.2.2 PA2.2: 实现更多的指令, 在 NEMU 中运行所有 cputest 22
3.2.3 PA2.3: 运行打字小游戏, 提交完整的实验报告 24
3.3 PA3 - 穿越时空的旅程: 批处理系统 29
3.3.1 PA3.1: 实现自陷操作_yield()及其过程 29
3.3.2 PA3.2: 实现用户程序加载和系统调用, 支撑 TRM 程序运行 29
3.3.3 PA3.3: 运行仙剑奇侠传展示批处理系统, 提交完整实验报告 29

4 总结与课程建议 32

4.1 实验总结 32
4.2 课程建议 32

参考文献 34

1 课程实验概述

1.1 课设目的

理解"程序如何在计算机上运行"的根本途径是从"零"开始实现一个完整的计算机系统. 华中科技大学计算机科学与技术系系统能力指令模拟器课程的小型项目 (Programming Assignment, PA)将提出 x86/mips32/riscv32 架构相应的教学版子集, 指导学生实现一个经过简化但功能完备的 x86/mips32/riscv32 模拟器 NEMU(NJU EMUlator), 最终在 NEMU 上运行游戏"仙剑奇侠传", 来让学生探究"程序在计算机上运行"的基本原理. NEMU 受到了 QEMU 的启发, 并去除了大量与课程内容差异较大的部分. PA 包括一个准备实验(配置实验环境)以及 5 部分连贯的实验内容:

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

1.2 实验环境

- CPU 架构: riscv32
- 操作系统: GNU/Linux
- 编译器: GCC
- 编程语言: C 语言

2 实验方案设计

2.1 PA1 – 开天辟地的篇章：最简单的计算机

PA1 的目标是实现一个简单的 monitor，能在 nemu 环境中起到跟 GDB 类似的作用，方便后续的 debug。PA 主要实现了以下功能：

命令	格式	使用举例	说明
帮助(1)	`help`	`help`	打印命令的帮助信息
继续运行(1)	`c`	`c`	继续运行被暂停的程序
退出(1)	`q`	`q`	退出NEMU
单步执行	`si [N]`	`si 10`	让程序单步执行 `N` 条指令后暂停执行，当 `N` 没有给出时，缺省为 `1`
打印程序状态	`info SUBCMD`	`info r` `info w`	打印寄存器状态 打印监视点信息
表达式求值	`p EXPR`	`p $eax + 1`	求出表达式 `EXPR` 的值，`EXPR` 支持的运算请见调试中的表达式求值小节
扫描内存(2)	`x N EXPR`	`x 10 $esp`	求出表达式 `EXPR` 的值，将结果作为起始内存地址，以十六进制形式输出连续的 `N` 个4字节
设置监视点	`w EXPR`	`w *0x2000`	当表达式 `EXPR` 的值发生变化时，暂停程序执行
删除监视点	`d N`	`d 2`	删除序号为 `N` 的监视点

图 2.1.1 简易调试器命令格式和功能

2.1.1 PA1.1: 实现单步执行，打印寄存器状态，扫描内存

首先是单步执行，根据 doc 的提示，我们仿照 ui_mainloop() 等函数中的 strtok() 用法，将 args 解析为多个字符串，然后利用 atoi() 函数将字符串转为 int 型即可。代码十分简洁，如下：

```
1. static int cmd_si(char *args) {
2.   char *arg = strtok(NULL, " ");
3.   int n;
4.   if (arg == NULL) {
5.     n = 1;
6.   } else {
7.     n = atoi(arg);
8.   }
9.   cpu_exec(n);
10.  return 0;
11. }
```

代码 2.1.1 cmd_si 函数实现

其次是打印寄存器状态，这也很简单，我们只需要沿着 ui.c => reg.c 的路径，将调用栈补全即可，最后修改 isa_reg_display() 函数，将 32 个寄存器全部输出：

```
1. void isa_reg_display() {
2.   int i;
3.   for (i = 0; i < 32; i++) {
4.     printf("%s\t0x%08x\t%d\n", reg_name(i, 4), reg_l(i), reg_l(i));
5.   }
6. }
```

代码 2.1.2 isa_reg_display **函数实现**

最后的扫描内存，我们放到算术表达式求值后来实现，因为需要大量调用 expr() 函数。

2.1.2 PA1.2: 实现算术表达式求值

这一部分较为复杂，但还好 doc 里有详尽的指导，我们只需按部就班即可。

首先先添加所需的 enum 类，我这里还未实现负数，故只添加了如下几种：

```
1. enum {
2.   TK_NOTYPE = 256,
3.   TK_EQ,
4.   /* TODO: Add more token types */
5.   TK_NEQ,
6.   TK_AND,
7.   TK_OR,
8.   TK_NUM,
9.   TK_HEX,
10.  TK_REG,
11.  TK_DEREF,
12. };
```

代码 2.1.3 token_type **实现**

之后需要填充所有 token 的正则表达式，这里需要尤其注意的是，越往前的 reg 优先级越高，所以需要把范围最广的（如数字）放在后面，不然就会出现 0x10000 被识别为 num 而不是 hex 的情况出现。我的正则表达式如下：

```
1. {"0[xX][0-9a-fA-F]+", TK_HEX}, // hexadecimal numbers
2. {"\\$[a-zA-Z][a-zA-Z0-9]*", TK_REG}, // registers
3. {" +", TK_NOTYPE},    // spaces
4. {"\\+", '+'},         // plus
5. {"-", '-'},          // minus
6. {"\\*", '*'},         // multiply
7. {"/", '/'},           // divide
8. {"\\(", '('},         // left parenthesis
9. {"\\)", ')'},         // right parenthesis
10. {"==", TK_EQ},        // equal
11. {"!=", TK_NEQ},       // not equal
12. {"&&", TK_AND},       // and
13. {"\\|\\|", TK_OR},    // or
```

```
14. {"[0-9]+", TK_NUM},    // numbers
```

代码 2.1.4 正则表达式

之后需要补全 make_token() 函数，主要是对不同的 token_type 要进行不同的处理，数字等需要保留字符串的 token 就需要 copy 下来。

```
 1. switch (rules[i].token_type) {
 2.   case TK_NOTYPE:
 3.     break;
 4.   case TK_NUM:
 5.   case TK_HEX:
 6.   case TK_REG:
 7.     tokens[nr_token].type = rules[i].token_type;
 8.     strncpy(tokens[nr_token].str, substr_start, substr_len);
 9.     tokens[nr_token].str[substr_len] = '\0';
10.     nr_token++;
11.     break;
12.   case TK_EQ:
13.   case TK_NEQ:
14.   case TK_AND:
15.   case TK_OR:
16.   default:
17.     tokens[nr_token].type = rules[i].token_type;
18.     nr_token++;
19.     break;
20.   // default: TODO();
21. }
```

代码 2.1.5 token 结构体初始化

至于算法部分，按照 doc 的思路，我们每次提取出没有被括号包含的优先级最低的运算符，并执行分治算法，将左右两边的值分别计算出来，代码就不贴了。

在做到后面监视点的时候，会提及一个特殊的 type：解引用。我们需要自己实现将 “*” 转为解引用的逻辑：

```
 1. for (int i = 0; i < nr_token; i ++) {
 2.   if (tokens[i].type == '*') {
 3.     if (i == 0 || (tokens[i - 1].type != TK_NUM && tokens[i - 1].type != TK_HEX &&
 4.                    tokens[i - 1].type != TK_REG && tokens[i - 1].type != ')')) {
 5.       tokens[i].type = TK_DEREF;
 6.     }
 7.   }
 8.   // printf("token[%d]: type=%d, str=%s\n", i, tokens[i].type, tokens[i].str);
 9. }
```

代码 2.1.6 星号转解引用

之后为了测试 eval 的正确性，doc 还要求我们实现单元测试，随机生成一些合法的算术表达式，并计算出结果，统一放到 input 文件里。然后修改 is_batch_mode，触发 nemu 的 batch 模式，批量测试 eval 函数。

在实现的过程中也是遇到了许多 bug，主要还是如何避免除 0 的情况。我是在 gen-exp.c 中也实现了一个分治计算器，来解析生成的表达式是否会除 0，就是代码过于冗长，这里就不贴了。

2.1.3 PA1.3: 实现所有要求，提交完整的实验报告

至此，还剩下最后一个“硬骨头”——监视点。

主要分两部分：链表逻辑和功能函数。链表逻辑十分简单，我们只需要实现 wp 的申请和释放即可：

```
1. WP* new_wp() {
2.   Assert(free_ != NULL, "No free watchpoints available.");
3.   WP *wp = free_;
4.   free_ = free_->next;
5.   wp->next = head;
6.   head = wp;
7.   return wp;
8. }
9.
10. void free_wp(WP *wp) {
11.   if (wp == NULL) return;
12.   if (head == wp) {
13.     head = wp->next;
14.   } else {
15.     WP *p = head;
16.     while (p && p->next != wp) {
17.       p = p->next;
18.     }
19.     if (p == NULL) return;
20.     p->next = wp->next;
21.   }
22.   wp->next = free_;
23.   free_ = wp;
24.   wp->expr[0] = '\0';
25.   wp->last_value = 0;
26. }
```

代码 2.1.7 链表代码

功能函数的实现比较特殊，我们得回到监视点的功能：在每次 cpu_exec() 执行结束后计算 exp 的值是否有发生改变。那么我们首先得实现 check_wp() 函数，来计算当前状态下的 exp 的值和保存在结构体内的 last_value 是否一致：

```
1. bool check_wp() {
2.   WP *p = head;
3.   bool triggered = false;
4.
5.   while (p != NULL) {
6.     bool success = true;
```

```
    uint32_t current_value = expr(p->expr, &success);
    if (!success) {
      printf("Failed to evaluate expression for watchpoint %d: %s\n", p->NO,
p->expr);
      p = p->next;
      continue;
    }

    if (current_value != p->last_value) {
      printf("Watchpoint %d triggered: %s\n", p->NO, p->expr);
      printf("Old value = %u\n", p->last_value);
      printf("New value = %u\n", current_value);
      p->last_value = current_value;
      triggered = true;
    }

    p = p->next;
  }

  return triggered;
}
```

代码 2.1.8 check_wp() 代码

然后我们得找到 cpu_exec() 函数，在 DEBUG 内调用 check_wp()，并将 nemu_state.state 设为 NEMU_STOP 来达到停机的效果。

d 命令更是简单了，我们只需要增加一个 display_wp() 函数，显示所有的 watchpoint 即可：

```
void display_wp() {
  if (head == NULL) {
    printf("No watchpoints.\n");
    return;
  }
  printf("Num\tType\t\tWhat\n");
  WP *p = head;
  while (p) {
    printf("%d\twatchpoint\t%s\t%u\n", p->NO, p->expr, p->last_value);
    p = p->next;
  }
}
```

代码 2.1.9 display_wp() 代码

2.1.4 必答题

1. 送分题：我选择的 ISA 是_____.

   A：Riscv32

2.理解基础设施：我们通过一些简单的计算来体会简易调试器的作用. 首先作以下假设:

(1)假设你需要编译 500 次 NEMU 才能完成 PA.
(2)假设这 500 次编译当中, 有 90%的次数是用于调试.
(3)假设你没有实现简易调试器, 只能通过 GDB 对运行在 NEMU 上的客户程序进行调试. 在每一次调试中, 由于 GDB 不能直接观测客户程序, 你需要花费 30 秒的时间来从 GDB 中获取并分析一个信息.
(4)假设你需要获取并分析 20 个信息才能排除一个 bug.
那么这个学期下来, 你将会在调试上花费多少时间?
A: t1 = 500 * 90% * 30 * 20 = 270000s = 4500m = 75h

由于简易调试器可以直接观测客户程序, 假设通过简易调试器只需要花费 10 秒的时间从中获取并分析相同的信息. 那么这个学期下来, 简易调试器可以帮助你节省多少调试的时间?
A：t2 = 500 * 90% * 10 * 20 = 90000s，dt = t1 – t2 = 180000s = 50h

事实上, 这些数字也许还是有点乐观, 例如就算使用 GDB 来直接调试客户程序, 这些数字假设你能通过 10 分钟的时间排除一个 bug. 如果实际上你需要在调试过程中获取并分析更多的信息, 简易调试器这一基础设施能带来的好处就更大.

3.查阅手册：理解了科学查阅手册的方法之后, 请你尝试在你选择的 ISA 手册中查阅以下问题所在的位置, 把需要阅读的范围写到你的实验报告里面:
(1)x86
1.EFLAGS 寄存器中的 CF 位是什么意思?
2.ModR/M 字节是什么?
3.mov 指令的具体格式是怎么样的?
(2)mips32
1.mips32 有哪几种指令格式?
2.CP0 寄存器是什么?
3.若除法指令的除数为 0, 结果会怎样?
(3)riscv32
1.riscv32 有哪几种指令格式?
A：6 种指令格式。RISC-V-Reader-Chinese-v2p12017 的第 26 页，图 2.2
2.LUI 指令的行为是什么?
A：将符号位扩展的 20 位立即数 immediate 左移 12 位，并将低 12 位置零，写入 x[rd]中。 压缩形式：c.lui rd, imm。RISC-V-Reader-Chinese-v2p12017 的第 153 页。
3.mstatus 寄存器的结构是怎么样的?
A：见下图

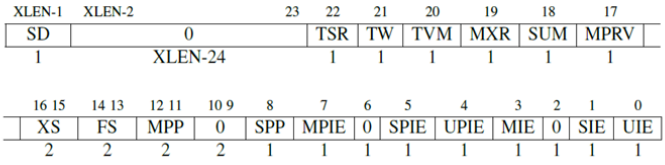


图 10.4：mstatus 控制状态寄存器。在仅有机器模式且没有 F 和 V 扩展的简单处理中，有效的域只有全局中断使能、MIE 和 MPIE（它在异常发生后保存 MIE 的旧值）。RV32 的 XLEN 时 32，RV64 是 40。（来自[Waterman and Asanovic 2017]中的表 3.6；有关其他域的说明请参见该文档的第 3.1 节。）

图 2.1.2 mstatus

4.shell 命令：完成 PA1 的内容之后, nemu/目录下的所有.c 和.h 和文件总共有多少行代码? 你是使用什么命令得到这个结果的? 和框架代码相比, 你在 PA1 中编写了多少行代码? (Hint: 目前 pa1 分支中记录的正好是做 PA1 之前的状态, 思考一下应该如何回到"过去"?) 你可以把这条命令写入 Makefile 中, 随着实验进度的推进, 你可以很方便地统计工程的代码行数, 例如敲入 make count 就会自动运行统计代码行数的命令. 再来个难一点的, 除去空行之外, nemu/目录下的所有.c 和.h 文件总共有多少行代码?

A：用 find 命令筛选出所有的.c 和.h 文件，然后用 xargs 转换为参数传递给 word，word 命令再统计行数。代码如下，统计结果为 5616 行。

```
1. find nemu -name "*.[ch]" | xargs wc -l
```

代码 2.1.10 统计行数代码

因为现在还没有 merge，所以可以先切换到 master 分支（或者 pa0）。统计结果为 4968 行。可得我新增了 5616 – 4968 = 648 行

要排除空行的话，我们需要用正则表达式反选，如下

```
1. find nemu -name "*.[ch]" | xargs cat | grep -v "^$" | wc -l
```

5.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?

A：-Wall 是“Warning All”的意思，-Werror 是“Warning to Error”的意思。

用-Wall 可以开启编译器几乎所有的警告信息。虽然名字叫 "All"，但它实际上开启的是通过长期实践被认为最有用的那部分警告集合

用-Werror 将所有的警告当作错误处理。一旦编译过程中出现任何警告，编译器就会停止编译并报错，无法生成可执行文件。

2.2 PA2 – 简单复杂的机器：冯・诺伊曼计算机系统

这一阶段需要我们实现 riscv32 指令集并实现 IOE 来支持各种 device。

2.2.1 PA2.1: 在 NEMU 中运行第一个 C 程序 dummy

PA2.1 比较简单，但偏工程，细节多。原理跟组原实验一样，都是需要我们实现 decode、exec。我们只需要理解代码流程，仿照已经提供的 lw、st 指令扩展更多指令即可。

首先查看反汇编代码，发现部分指令没有实现：auipc、addi、jal 和 jalr。这里以 auipc 为例，我们首先需要找到 decode 的入口，即在 opcode_table 里面按照 opcode 添加对应的宏：

```
1. static OpcodeEntry opcode_table [32] = {
2.   /* b00000 */ IDEX(ld, load),    EMPTY,             EMPTY,           EMPTY,
3.   /* b00100 */ IDEX(I, op_imm),   IDEX(U, auipc),    EMPTY,           EMPTY,
4.   /* b01000 */ IDEX(st, store),   EMPTY,             EMPTY,           EMPTY,
5.   /* b01100 */ IDEX(R, op_r),     IDEX(U, lui),      EMPTY,           EMPTY,
6.   /* b10000 */ EMPTY,            EMPTY,             EMPTY,           EMPTY,
7.   /* b10100 */ EMPTY,            EMPTY,             EMPTY,           EMPTY,
8.   /* b11000 */ IDEX(B, op_b),     IDEX(I, jalr),     EX(nemu_trap),    IDEX(J, jal),
9.   /* b11100 */ EMPTY,            EMPTY,             EMPTY,    EMPTY,
10. };
```

代码 2.2.1 pa2 完成后的 opcode_table

这里添加 IDEX(U, auipc) 的含义就是先执行 decode_U() 函数，然后再执行 exec_auipc() 函数。然后我们再在 compute.c 里面添加所需的 exec_auipc() 函数：

```
1. make_EHelper(auipc) {
2.   // result = pc + imm - 4
3.   rtl_addi(&s0, &decinfo.seq_pc, id_src->imm);
4.   rtl_subi(&s0, &s0, 4);
5.   // rd = result
6.   rtl_sr(id_dest->reg, &s0, 4);
7.
8.   print_asm_template2(auipc);
9. }
```

代码 2.2.2 exec_auipc() 代码

这里也是仿照 exec_ld 应用 rtl 宏来实现寄存器操作。

目的是通过 dummy 的话，那我们只需要重复如上操作即可（但也会为之后的测试埋下许多坑。。。）

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

PA2.2 相比于 PA2.1，只需要注意完成以下几个部分即可：

1.添加更多类型指令的 decode 代码，如 R 型、B 型指令。在实现解码逻辑时，得格外注意变量类型是有符号还是无符号。PA2 实现的指令中，所有的立即数都是有符号类型，这也体现在 Instr 结构体的定义中：

```
struct {
  uint32_t pad1      : 7;
  uint32_t imm4_0    : 5;
  uint32_t pad2      :13;
  int32_t  simm11_5  : 7;
};
```

代码 2.2.3 Instr 结构体

可以看到最高位的立即数都是 int32_t 类型。所以我们在 decode 时**务必小心符号**。以 B-type 为例：

```
make_DHelper(B) {
  uint32_t imm12 = (decinfo.isa.instr.simm12 << 12) | (decinfo.isa.instr.imm10_5 << 5)
| (decinfo.isa.instr.imm4_1 << 1) | (decinfo.isa.instr.imm11 << 11);
  int32_t simm = (int32_t)(imm12 << 19) >> 19;
  decode_op_r(id_src, decinfo.isa.instr.rs1, true);
  decode_op_r(id_src2, decinfo.isa.instr.rs2, true);
  decode_op_i(id_dest, simm, true);
  print_Dop(id_dest->str, OP_STR_SIZE, "%d", id_dest->imm);
}
```

代码 2.2.4 exec_B() 代码

为了保险起见（才不是搞不明白有符号和无符号的隐式类型转换），我手动加上了符号转换。

2.译指的三级结构。R-type 和部分 I-type 指令都存在三级结构：opcode、funct3、funct7。所以在实现新指令时，不能只考虑 hit 的情况，得提前预留出空间给未实现指令 miss，不然就会像我一样调试半天，发现是有一个指令未实现。

那为了实现清晰易懂的结构化译指，我给部分指令实现了三级译指，以 R-type 为例：

```
static OpcodeEntry op_r_table [8] = {
  /* b000 */ EX(op_r_0),
  /* b001 */ EX(op_r_1),
  /* b010 */ EX(op_r_2),
  /* b011 */ EX(op_r_3),
  /* b100 */ EX(op_r_4),
  /* b101 */ EX(op_r_5),
  /* b110 */ EX(op_r_6),
  /* b111 */ EX(op_r_7),
};
```

代码 2.2.5 or_r_table 代码

每个 funct3 到对应的第三级译指，比如 op_r_0

```
static OpcodeEntry op_r_0_table [3] = {
  EX(add),
  EX(sub),
  EX(mul),
};
```

```
6.
7. static make_EHelper(op_r_0) {
8.   switch (decinfo.isa.instr.funct7) {
9.     case 0b0000000: idex(pc, &op_r_0_table[0]); break;
10.    case 0b0100000: idex(pc, &op_r_0_table[1]); break;
11.    case 0b0000001: idex(pc, &op_r_0_table[2]); break;
12.    default: exec_inv(pc); break;
13.  }
14. }
```

代码 2.2.6 op_r_0 处理逻辑

注意这里我们最好用 switch 语句，这样的话就可以用 default 来预留空间给未实现的指令了。

3.klib 实现。需要注意的是，klib 中有许多的运行时环境没有实现，而且仅仅返回 0，在运行时基本感觉不出来（就是你 strlen，调了我 1h 才发现）。所以我们最好手动加上 assert(0) 来提示自己。

4.diff-test 实现。这里我遇到了莫名其妙的 bug：在启动 qemu 时，发现报了以下错误：

```
1. qemu-system-riscv32: Failed to set FD nonblocking: Bad file descriptor
2. qemu-system-riscv32: could not connect serial device to character backend 'mon:stdio'
```

代码 2.2.7 qemu 错误

经过了漫长的 google，场外求援，翻 github 等等尝试，我差点放弃了，直到用 gemini 来帮我一点点读代码，修改代码，终于找到了问题：

```
1. //execlp(ISA_QEMU_BIN, ISA_QEMU_BIN, ISA_QEMU_ARGS "-S", "-s", "-nographic", NULL);
2. execlp(ISA_QEMU_BIN, ISA_QEMU_BIN, ISA_QEMU_ARGS "-S", "-s", "-nographic", "-serial",
"none", "-monitor", "none", NULL);
```

代码 2.2.8 diff-test.c 修改部分

也不清楚到底是 VirtualBox 的问题还是 ics2019 的问题，至少现在能跑了。

2.2.3 PA2.3: 运行打字小游戏，提交完整的实验报告

这一部分主要修改的是 device 相关，只需要我们实现时钟、键盘和显示逻辑。

时钟部分很简单，按照 doc 的提示，我们找到 RTC_ADDR，用提供的 inl 函数读取 4 个字节，即为当前的时间。之后再减去 boot_time 即为开机时间。

```
1. case _DEVREG_TIMER_UPTIME: {
2.   _DEV_TIMER_UPTIME_t *uptime = (_DEV_TIMER_UPTIME_t *)buf;
3.   uptime->hi = 0;
4.   uptime->lo = inl(RTC_ADDR) - boot_time;
5.   return sizeof(_DEV_TIMER_UPTIME_t);
6. }
```

代码 2.2.9 时钟 uptime 逻辑

键盘更是简单，因为 am 的解耦，硬件部分与 am 无关，即开/关信号和 keycode 都已经保存到 KBD_ADDR 那里，我们只需要读取再保存即可。

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

代码 2.2.10 键盘 read 逻辑

VGA 部分说实话很复杂。我之前在硬件综合训练就是实现 VGA 逻辑的，但这里的 TODO 部分太简略了，在 am 层面，我们只需要实现 info 的 read 和 framebuff 的 write 即可。

```c
case _DEVREG_VIDEO_FBCTL: {
  _DEV_VIDEO_FBCTL_t *ctl = (_DEV_VIDEO_FBCTL_t *)buf;
  int x = ctl->x;
  int y = ctl->y;
  int w = ctl->w;
  int h = ctl->h;
  int W = inl(SCREEN_ADDR) >> 16;
  int H = inl(SCREEN_ADDR) & 0xffff;
  if (x < 0 || y < 0 || x + w > W || y + h > H) {
    return 0;
  }
  uint32_t addr = FB_ADDR + (y * W + x) * sizeof(uint32_t);
  uint32_t *pixels = ctl->pixels;
  for (int i = 0; i < h; i++) {
    for (int j = 0; j < w; j++) {
      outl(addr + (i * W + j) * sizeof(uint32_t), pixels[i * w + j]);
    }
  }
  if (ctl->sync) {
    outl(SYNC_ADDR, 0);
  }
  return size;
}
```

代码 2.2.11 fb write 代码

这里我用两个 for 循环来写入 fb，我看好像可以用 memset 来提高性能，这部分就之后再完善。

之后在硬件(nemu)层面，PA 还留了个 TODO：实现一个 handler 来触发 redraw：

```
1. static void vga_io_handler(uint32_t offset, int len, bool is_write) {
2.   if (is_write && (offset == (SYNC_PORT - SCREEN_PORT) || offset == (SYNC_MMIO -
SCREEN_MMIO))) {
3.     update_screen();
4.   }
5. }
```

代码 2.2.12 vga handler

到此 PA2 的所有 task 都已完成，接下来就是轻松快乐的测试时间。

2.2.4 必答题

1.RTFSC 请整理一条指令在 NEMU 中的执行过程. (我们其实已经在 PA2.1 阶段提到过这道题了)

A：首先 isa_exec 调用 instr_fetch 取出当前指令，查找 opcode 表得到对应的 decode 和 exec 函数，decode 是在 decode.c 中根据指令类型(B、I 等)分别实现的，exec 是在 all_instr.h 中注册的每个指令对应的执行逻辑。当然有的指令需要分级 decode。

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

A：单独去掉 static 和单独去掉 inline，虽然会报编译 warning，但是都可以正常运行；只有将两者全部去掉，就会出现编译错误。

原因是单独用 static 会让每个包含该.h 文件的.c 文件都生成一份该函数的私有副本，不会出现编译错误，但是会比不用 static 体积膨胀；单独用 inline 则会尽可能将函数内联，但若是遇到无法内联的情况（递归）照样出错。

证明方法：查看符号表。对于单独去掉 inline 的情况，会看到符号变成了局部符号（小写 t 或被重命名以区分）。

3.编译与链接

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

A：理论上是等于所有包含这个 common.h 的.c 文件的数量。但实际测试的时候会发现比预期多两个，通过全局搜索可以发现是因为有函数名字叫 __am_platform_dummy()，故会多两个，实际应该是 35 个。用 nm 跑下面的代码即可：

```
1. nm build/riscv32-nemu | grep dummy | wc -l
```

代码 2.2.13 nm **分析** wc **统计** dummy **数量**

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

A：跟上一问的结果一样，还是 35 个。阅读代码可得，common.h 中手动 include 了 debug.h，这样相当于 common.h 中有两个一样的 dummy 声明。又因为两个 dummy 都没有初始化，故被视为 Tentative Definition，被合并为一个了。又因为全局搜索，发现 debug.h 并没有被其他的 include 了，故最终还是 35 个。

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

A：报编译错误（重复定义）了。因为若是初始化，就不是 Tentative Definition 了，而是 Definition，就是经典的重复定义了。

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

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

A：首先定义了许多变量比如 ISA，INCLUDE，提供了可配置的编译选项。app 依赖于 binary，binary 依赖于所有的.o 文件，每个.o 文件都会在包含的.c/.h 文件被修改后重新编译一遍，保证了修改的时效性。最后统一链接。

2.3 PA3 - 穿越时空的旅程: 批处理系统

PA3 需要我们实现异常和系统调用，再实现简单的文件系统操作，这样就能运行仙剑奇侠传。

2.3.1 PA3.1: 实现自陷操作_yield()及其过程

PA3.1 主要是熟悉异常的调用栈。根据组原学习的相关知识，异常等中断操作，无非以下流程：保存上下文，通过异常/中断号找到中断处理程序地址，跳转并执行，返回后恢复上下文。下面我们就逐步理清一个_yield()自陷操作的调用流程。

首先，查看 dummy.c 文件，内部调用了 _syscall_ 函数。在 riscv32 架构下，_syscall_ 函数调用 ecall 指令。ecall 指令调用 raise_intr() 函数，触发 User Call（中断号为 8），保存上下文到 csr 寄存器，并找到对应的中断向量：

```
1. void raise_intr(uint32_t NO, vaddr_t epc) {
2.   /* TODO: Trigger an interrupt/exception with ``NO''.
3.    * That is, use ``NO'' to index the IDT.
4.    */
5.   cpu.sepc = epc;
6.   cpu.scause = NO;
7.   decinfo.jmp_pc = cpu.stvec;
8.   rtl_j(cpu.stvec);
9. }
```

代码 2.3.1 raise_intr 函数

准备好上下文后，我们进入 trap.s 中的 __am_asm_trap 函数，通过阅读汇编代码可得，该函数内部将 gpr 和 csr 都保存到栈上，我们根据代码修改 Context 结构体顺序如下：

```
1. struct _Context {
2.   uintptr_t gpr[32], cause, status, epc;
3.   struct _AddressSpace *as;
4. };
```

代码 2.3.2 _Context 结构体

之后从 __am_asm_trap 跳转到 __am_irp_handle。这里我们通过上下文中的 scause 寄存器的值（PA3 只有 User Call），和 ecall 的第一个参数（GPR1）来决定中断事件的类型

```
1. switch (c->cause) {
2.   case IRQ_UECALL:
3.   switch (c->GPR1) {
4.     case -1: ev.event = _EVENT_YIELD; break;
5.     default: ev.event = _EVENT_SYSCALL; break;
6.   }
7.   break;
8.   default: ev.event = _EVENT_ERROR; break;
9. }
```

代码 2.3.3 中断事件逻辑

之后会跳转到 user_handler 进行进一步的处理，user_handler 就是在_init_cte 中注册的函数 do_event。在 do_event 中，我们根据之前确定的 event 号，跳转到对应的处理函数，比如 _EVENT_SYSCALL，就跳转到 do_syscall 函数：

```
1. static _Context* do_event(_Event e, _Context* c) {
2.   switch (e.event) {
3.     case _EVENT_YIELD:
4.       printf("Handling yield event\n");
5.       c->GPRx = 0;
6.       return c;
7.       break;
8.     case _EVENT_SYSCALL:
9.       return do_syscall(c);
10.    default: panic("Unhandled event ID = %d", e.event);
11.  }
12.  return NULL;
13. }
```

代码 2.3.4 do_event 函数

这里我们更进一步，就会跳转到 do_syscall，在这里我们根据 navy-app 里的测试文件传入的 ecall 号（即 GPR1），来决定是那个系统调用。比如我们传入 SYS_YIELD，就调用 sys_yield() 函数即可。

至此一个完整的 ecall 流程就完成了。当然在实现过程中，会遇到许许多多的 bug，比如 csr 指令实现有问题、ecall 指令保存的不是当前 PC、do_event 里面没有 PC+4。但通过锲而不舍的逐 PC 调试（并修改了好几次 monitor），以上 bug 还是很好解决的。

2.3.2 PA3.2: 实现用户程序加载和系统调用，支撑 TRM 程序运行

PA3.2 需要我们实现 ELF 文件的加载，增加新的系统调用（输出、堆区管理）来支持 TRM 程序的执行。

ELF 文件很简单，毕竟经过了 pke 实验的磨炼，还是很得心应手的。首先先读取 ehdr，通过 ehdr 得知每个 phdr 的 size 和 offset，来逐个 loader 所需的 phdr。这里就不赘述。

输出和堆区管理更是简单，通过 PA3.1 的磨炼，我对于 ecall 的流程了如指掌，只需要在每个环节添加相应的处理逻辑即可。

这里需要注意的是，虽然 PA3 不需要进程切换，但堆区其实是跟着 proc 走的，不应该用全局变量保存。所以按照 pke 的处理方法，我还是在 loader 中添加了对 current 的操作，保存 max_brk。

2.3.3 PA3.3: 运行仙剑奇侠传展示批处理系统，提交完整实验报告

PA3.3 需要我们实现一个简易的文件系统，并实现抽象文件的开关、读写操作，并细化完成设备文件的读写操作。

按照 linux 里一切皆文件的哲学，我们将所有的 IOE 扩展也当作文件操作，从键盘读入、写入屏幕都可以抽象为文件操作。

首先我们需要完成抽象后的 fs_read、fs_write、fs_open 和 fs_close。说实话

pke 的文件系统比 PA 要复杂得多，写完 pke 后再写 PA 真是小菜一碟。

以 fs_read 为例，我们先通过已知的 size 和 offset，限制 len 的长度，然后再调用特定设备的专有 read 函数即可。PA 约定，若是函数指针留空，那直接用 ramdisk_read 即可。

这里不得不吐槽下，PA 的简易文件系统太过简易了。因为封装的 ReadFn 传递的是 offset 的值，而不是指针；但其实应该让每个设备来决定是否 increase，像 stdout 和 events 就约等于拥有无限的 size。

当然这里肯定会有同学说，那我将 size 设置为(0xffff=-1)不就好了吗？这种做法有个问题，当 0xffff-len<offset<0xffff 时，照样会出问题，只是概率小点而已。

完成 fs 抽象后的相关函数，就需要我们完成 device 相关的具体函数。基本思路就是去 nexus-am/libs/klib/src/io.c 里找 Abstract Machine 提供的接口(如 uptime、read_key 和 draw_rect 等)，调用这些接口来实现读写操作。

最后的最后，我们还需要实现一个特殊的系统调用 SYS_EXECVE，我们只需要仿照开机时初始化 proc 的操作，再调用一次 naïve_uload 即可。

至此，PA3 堂堂结束。

2.3.4 必答题

1.理解上下文结构体的前世今生 (见 PA3.1 阶段)

Q：你会在__am_irq_handle()中看到有一个上下文结构指针 c, c 指向的上下文结构究竟在哪里？这个上下文结构又是怎么来的？具体地，这个上下文结构有很多成员，每一个成员究竟在哪里赋值的？$ISA-nemu.h, trap.S, 上述讲义文字，以及你刚刚在 NEMU 中实现的新指令，这四部分内容又有什么联系？

如果你不是脑袋足够灵光，还是不要眼睁睁地盯着代码看了，来用纸笔模拟一下__am_asm_trap()的执行过程吧.

A：这个上下文结构指针指向 trap.s 中在栈上保存的上下文。这一问是下一问的子集，就不赘述了。

2.理解穿越时空的旅程 (见 PA3.1 阶段)

Q：从 Nanos-lite 调用_yield()开始，到从_yield()返回的期间，这一趟旅程具体经历了什么？软(AM, Nanos-lite)硬(NEMU)件是如何相互协助来完成这趟旅程的？你需要解释这一过程中的每一处细节，包括涉及的每一行汇编代码/C 代码的行为，尤其是一些比较关键的指令/变量. 事实上，上文的必答题"理解上下文结构体的前世今生"已经涵盖了这趟旅程中的一部分，你可以把它的回答包含进来.

别被"每一行代码"吓到了，这个过程也就大约 50 行代码，要完全理解透彻并不是不可能的. 我们之所以设置这道必答题，是为了强迫你理解清楚这个过程中的每一处细节. 这一理解是如此重要，以至于如果你缺少它，接下来你面对 bug 几乎是束手无策.

A：这个问题我在 PA3.1 中已经模拟了 ecall 的流程，就不赘述了。

3.hello 程序是什么，它从而何来，要到哪里去 (见 PA3.2 阶段)

Q：到此为止，PA 中的所有组件已经全部亮相了，整个计算机系统也开始趋于完整. 你也已经在这个自己创造的计算机系统上跑起了 hello 这个第一个还说

得过去的用户程序 (dummy 是给大家热身用的, 不算), 好消息是, 我们已经距离运行仙剑奇侠传不远了(下一个阶段就是啦).

不过按照 PA 的传统, 光是跑起来还是不够的, 你还要明白它究竟怎么跑起来才行. 于是来回答这道必答题吧:

我们知道 navy-apps/tests/hello/hello.c 只是一个 C 源文件, 它会被编译链接成一个 ELF 文件. 那么, hello 程序一开始在哪里? 它是怎么出现内存中的? 为什么会出现在目前的内存位置? 它的第一条指令在哪里? 究竟是怎么执行到它的第一条指令的? hello 程序在不断地打印字符串, 每一个字符又是经历了什么才会最终出现在终端上?

上面一口气问了很多问题, 我们想说的是, 这其中蕴含着非常多需要你理解的细节. 我们希望你能够认真整理其中涉及的每一行代码, 然后用自己的语言融会贯通地把这个过程的理解描述清楚, 而不是机械地分点回答这几个问题.

同样地, 上一阶段的必答题"理解穿越时空的旅程"也已经涵盖了一部分内容, 你可以把它的回答包含进来, 但需要描述清楚有差异的地方. 另外, C 库中 printf() 到 write()的过程比较繁琐, 而且也不属于 PA 的主线内容, 这一部分不必展开回答. 而且你也已经在 PA2 中实现了自己的 printf()了, 相信你也不难理解字符串格式化的过程. 如果你对 Newlib 的实现感兴趣, 你也可以 RTFSC.

总之, 扣除 C 库中 printf()到 write()转换的部分, 剩下的代码就是你应该理解透彻的了. 于是, 努力去理解每一行代码吧!

A：hello 程序一开始只是在 ramdisk 上的一个静态 elf 文件，通过 loader 将需要载入的 segment 载入内存。

在载入结束后，loader 会返回 hello 的入口地址，就是 ehdr.e_entry，之后在 naïve_uload 中，简单粗暴地调用了这个地址的函数，机器就会从该位置开始运行。

输出的话，从 libc 的 printf 翻译为 write 的部分太过复杂，就不讲了。但是若是将_write 被调用时的 buf 和 len 全部打印出来，会发现其实就跟我们自己实现 printf 一模一样，都是用 va_list，逐个判断，遇到%d 之类的就先转为字符串，然后分段打印。

至于打印的系统调用就不赘述了。

4.仙剑奇侠传究竟如何运行 运行仙剑奇侠传时会播放启动动画, 动画中仙鹤在群山中飞过. 这一动画是通过 navy-apps/apps/pal/src/main.c 中的 PAL_SplashScreen()函数播放的. 阅读这一函数, 可以得知仙鹤的像素信息存放在数据文件 mgo.mkf 中. 请回答以下问题: 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? 换一种 PA 的经典问法: 这个过程究竟经历了些什么?

A：首先是读取像素。流程如下：pal app 先调用 Newlib 的 fread 函数，然后 fread 函数最终调用_read，之后就是重复的系统调用。

然后是显示画面。pal app 调用 SDL_UpdateRect 刷新像素，进入了 NDL 库。NDL 库将请求转换为对/dev/fb 的写入操作，调用_write，之后就是系统调用。最后模拟器调用 draw_sync，让显示器刷新（切换 buf 和 frame），这样就能实现动画了。

3　实验结果与结果分析

3.1　PA1 – 开天辟地的篇章: 最简单的计算机

3.1.1　PA1.1: 实现单步执行, 打印寄存器状态, 扫描内存

首先是单步执行的测试，测试了带参数和不带参数两种情况：

```
Welcome to riscv32-NEMU!
For help, type "help"
(nemu) si
80100000:   b7 02 00 80                                 lui  0x80000,t0
(nemu) si 2
80100004:   23 a0 02 00                                 sw  0(t0),$0
80100008:   03 a5 02 00                                 lw  0(t0),a0
```

图 3.1.1 si 测试截图

然后是打印寄存器状态测试，这里是在运行到0x8010008时的info，很正确：

```
(nemu) info r
$0      0x00000000      0
ra      0x00000000      0
sp      0x00000000      0
gp      0x00000000      0
tp      0x00000000      0
t0      0x80000000      2147483648
t1      0x00000000      0
t2      0x00000000      0
s0      0x00000000      0
s1      0x00000000      0
a0      0x00000000      0
a1      0x00000000      0
a2      0x00000000      0
a3      0x00000000      0
a4      0x00000000      0
a5      0x00000000      0
a6      0x00000000      0
a7      0x00000000      0
s2      0x00000000      0
s3      0x00000000      0
s4      0x00000000      0
s5      0x00000000      0
s6      0x00000000      0
s7      0x00000000      0
s8      0x00000000      0
s9      0x00000000      0
s10     0x00000000      0
s11     0x00000000      0
t3      0x00000000      0
t4      0x00000000      0
t5      0x00000000      0
t6      0x00000000      0
```

图 3.1.2 info r 测试截图

最后是扫描内存测试，这里是按照 doc 的思路测试，读取程序段的内容：

```
(nemu) x 4 0x80100000
arg_n: 4
arg_expr: 0x80100000
[src/monitor/debug/expr.c,91,make_token] match rules[0] = "0[xX][0-9a-fA-F]+" at position 0 with len 10: 0x80100000
Memory from 0x80100000:
0x80100000: 0x800002b7 2147484343
0x80100004: 0x0002a023 172067 #
0x80100008: 0x0002a503 173315
0x8010000c: 0x0000006b 107 k
```

图 3.1.3 扫描内存测试截图

可以看出，跟实际指令的内容一样。

3.1.2 PA1.2: 实现算术表达式求值

首先测试了几个不合法的 expr：

```
(nemu) p 1
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "[0-9]+" at position 0 with len 1: 1
1 (0x1)
(nemu) p 1/0
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "[0-9]+" at position 0 with len 1: 1
[src/monitor/debug/expr.c,91,make_token] match rules[6] = "/" at position 1 with len 1: /
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "[0-9]+" at position 2 with len 1: 0
Invalid expression: 1/0
(nemu) p ((1+1)
[src/monitor/debug/expr.c,91,make_token] match rules[7] = "\(" at position 0 with len 1: (
[src/monitor/debug/expr.c,91,make_token] match rules[7] = "\(" at position 1 with len 1: (
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "[0-9]+" at position 2 with len 1: 1
[src/monitor/debug/expr.c,91,make_token] match rules[3] = "\+" at position 3 with len 1: +
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "[0-9]+" at position 4 with len 1: 1
[src/monitor/debug/expr.c,91,make_token] match rules[8] = "\)" at position 5 with len 1: )
Invalid expression: ((1+1)
(nemu) p 1//1
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "[0-9]+" at position 0 with len 1: 1
[src/monitor/debug/expr.c,91,make_token] match rules[6] = "/" at position 1 with len 1: /
[src/monitor/debug/expr.c,91,make_token] match rules[6] = "/" at position 2 with len 1: /
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "[0-9]+" at position 3 with len 1: 1
Invalid expression: 1//1
```

图 3.1.4 invalid expr 测试

然后采用单元测试的思路，用 batch_mode 测试了 10 个 case：

```
Welcome to riscv32-NEMU!
For help, type "help"
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "[0-9]+" at position 0 with len 2: 55
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "[0-9]+" at position 0 with len 2: 74
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "[0-9]+" at position 0 with len 2: 64
[src/monitor/debug/expr.c,91,make_token] match rules[7] = "\(" at position 0 with len 1: (
[src/monitor/debug/expr.c,91,make_token] match rules[7] = "\(" at position 1 with len 1: (
[src/monitor/debug/expr.c,91,make_token] match rules[7] = "\(" at position 2 with len 1: (
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "[0-9]+" at position 3 with len 2: 64
[src/monitor/debug/expr.c,91,make_token] match rules[8] = "\)" at position 5 with len 1: )
[src/monitor/debug/expr.c,91,make_token] match rules[8] = "\)" at position 6 with len 1: )
[src/monitor/debug/expr.c,91,make_token] match rules[8] = "\)" at position 7 with len 1: )
[src/monitor/debug/expr.c,91,make_token] match rules[7] = "\(" at position 0 with len 1: (
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "[0-9]+" at position 1 with len 2: 90
[src/monitor/debug/expr.c,91,make_token] match rules[8] = "\)" at position 3 with len 1: )
[src/monitor/debug/expr.c,91,make_token] match rules[3] = "\+" at position 4 with len 1: +
[src/monitor/debug/expr.c,91,make_token] match rules[7] = "\(" at position 5 with len 1: (
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "[0-9]+" at position 6 with len 2: 10
[src/monitor/debug/expr.c,91,make_token] match rules[8] = "\)" at position 8 with len 1: )
[src/monitor/debug/expr.c,91,make_token] match rules[7] = "\(" at position 0 with len 1: (
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "[0-9]+" at position 1 with len 1: 4
[src/monitor/debug/expr.c,91,make_token] match rules[8] = "\)" at position 2 with len 1: )
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "[0-9]+" at position 0 with len 2: 33
[src/monitor/debug/expr.c,91,make_token] match rules[5] = "\*" at position 2 with len 1: *
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "[0-9]+" at position 3 with len 2: 23
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "[0-9]+" at position 0 with len 2: 31
[src/monitor/debug/expr.c,91,make_token] match rules[7] = "\(" at position 0 with len 1: (
[src/monitor/debug/expr.c,91,make_token] match rules[7] = "\(" at position 1 with len 1: (
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "[0-9]+" at position 2 with len 2: 37
[src/monitor/debug/expr.c,91,make_token] match rules[6] = "/" at position 4 with len 1: /
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "[0-9]+" at position 5 with len 2: 63
[src/monitor/debug/expr.c,91,make_token] match rules[8] = "\)" at position 7 with len 1: )
[src/monitor/debug/expr.c,91,make_token] match rules[8] = "\)" at position 8 with len 1: )
[src/monitor/debug/expr.c,91,make_token] match rules[13] = "[0-9]+" at position 0 with len 2: 55
Batch test summary: 10 / 10 tests passed.
```

图 3.1.5 单元测试截图

最后还测试了解引用的正确性：

```
Welcome to riscv32-NEMU!
For help, type "help"
(nemu) p *0x80100000
[src/monitor/debug/expr.c,91,make_token] match rules[5] = "\*" at position 0 with len 1: *
[src/monitor/debug/expr.c,91,make_token] match rules[0] = "0[xX][0-9a-fA-F]+" at position 1 with len 10: 0x80100000
2147484343 (0x800002b7)
(nemu) p **0x80100000
[src/monitor/debug/expr.c,91,make_token] match rules[5] = "\*" at position 0 with len 1: *
[src/monitor/debug/expr.c,91,make_token] match rules[5] = "\*" at position 1 with len 1: *
[src/monitor/debug/expr.c,91,make_token] match rules[0] = "0[xX][0-9a-fA-F]+" at position 2 with len 10: 0x80100000
2147484343 (0x800002b7)
(nemu) p *0x800002b7
[src/monitor/debug/expr.c,91,make_token] match rules[5] = "\*" at position 0 with len 1: *
[src/monitor/debug/expr.c,91,make_token] match rules[0] = "0[xX][0-9a-fA-F]+" at position 1 with len 10: 0x800002b7
0 (0x0)
```

图 3.1.6 解引用测试截图（bug）

发现出现了 bug，仔细考虑，原来是因为解引用是右结合运算符，最左边的解引用才是 main_op。修改了 main_op 查找逻辑后，测试如下：

```
Welcome to riscv32-NEMU!
For help, type "help"
(nemu) p **0x80100000
[src/monitor/debug/expr.c,91,make_token] match rules[5] = "\*" at position 0 with len 1: *
[src/monitor/debug/expr.c,91,make_token] match rules[5] = "\*" at position 1 with len 1: *
[src/monitor/debug/expr.c,91,make_token] match rules[0] = "0[xX][0-9a-fA-F]+" at position 2 with len 10: 0x80100000
0 (0x0)
```

图 3.1.7 解引用测试截图

也是顺利通过了。

3.1.3 PA1.3: 实现所有要求，提交完整的实验报告

监视点的测试参考了 doc 的思路，通过监视 $pc 的变化来模拟断点：

```
(nemu) w $pc == 0x80100008
[src/monitor/debug/expr.c,91,make_token] match rules[1] = "\$[a-zA-Z][a-zA-Z0-9]*" at position 0 with len 3: $pc
[src/monitor/debug/expr.c,91,make_token] match rules[2] = " +" at position 3 with len 1:
[src/monitor/debug/expr.c,91,make_token] match rules[9] = "==" at position 4 with len 2: ==
[src/monitor/debug/expr.c,91,make_token] match rules[2] = " +" at position 6 with len 1:
[src/monitor/debug/expr.c,91,make_token] match rules[0] = "0[xX][0-9a-fA-F]+" at position 7 with len 10: 0x80100008
Watchpoint 0: $pc == 0x80100008
(nemu) info w
Num     Type           Dec     Hex     What
0       watchpoint     0       0x0     $pc == 0x80100008
(nemu) c
[src/monitor/debug/expr.c,91,make_token] match rules[1] = "\$[a-zA-Z][a-zA-Z0-9]*" at position 0 with len 3: $pc
[src/monitor/debug/expr.c,91,make_token] match rules[2] = " +" at position 3 with len 1:
[src/monitor/debug/expr.c,91,make_token] match rules[9] = "==" at position 4 with len 2: ==
[src/monitor/debug/expr.c,91,make_token] match rules[2] = " +" at position 6 with len 1:
[src/monitor/debug/expr.c,91,make_token] match rules[0] = "0[xX][0-9a-fA-F]+" at position 7 with len 10: 0x80100008
[src/monitor/debug/expr.c,91,make_token] match rules[1] = "\$[a-zA-Z][a-zA-Z0-9]*" at position 0 with len 3: $pc
[src/monitor/debug/expr.c,91,make_token] match rules[2] = " +" at position 3 with len 1:
[src/monitor/debug/expr.c,91,make_token] match rules[9] = "==" at position 4 with len 2: ==
[src/monitor/debug/expr.c,91,make_token] match rules[2] = " +" at position 6 with len 1:
[src/monitor/debug/expr.c,91,make_token] match rules[0] = "0[xX][0-9a-fA-F]+" at position 7 with len 10: 0x80100008
Watchpoint 0 triggered: $pc == 0x80100008
Old value = 0
New value = 1
(nemu) info w
Num     Type           Dec     Hex     What
0       watchpoint     1       0x1     $pc == 0x80100008
(nemu) d 0
Watchpoint 0 deleted.
(nemu) info w
No watchpoints.
```

图 3.1.8 监视点测试

3.2 PA2 - 简单复杂的机器: 冯诺依曼计算机系统

3.2.1 PA2.1: 在 NEMU 中运行第一个 C 程序 dummy

```
Welcome to riscv32-NEMU!
For help, type "help"
nemu: HIT GOOD TRAP at pc = 0x80100030

[src/monitor/cpu-exec.c,31,monitor_statistic] total guest instructions = 13
 dummy
```

图 3.2.1 dummy 测试截图

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

首先在 DEBUG 模式下运行 runall.sh

```
mystle@PANJU:~/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!
[       my_add] PASS!
[       my_mul] 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 3.2.2 runall.sh 运行结果

然后我们故意制造点 bug（其实是我之前写错了，这里复现下），将 auipc 写入的值为$pc + imm + 4，然后再运行一遍 runall.sh

```
mystle@PANJU:~/ics2019/nemu$ bash runall.sh ISA=riscv32
compiling NEMU...
Building riscv32-nemu
+ CC src/isa/riscv32/exec/compute.c
+ LD build/riscv32-nemu
NEMU compile OK
compiling testcases...
testcases compile OK
[  add-longlong] PASS!
[           add] PASS!
[           bit] PASS!
[   bubble-sort] PASS!
[           div] PASS!
[         dummy] PASS!
[          fact] PASS!
[           fib] PASS!
[      goldbach] PASS!
[     hello-str] PASS!
[       if-else] PASS!
[     leap-year] PASS!
[    load-store] PASS!
[    matrix-mul] PASS!
[           max] PASS!
[          min3] PASS!
[         mov-c] PASS!
[         movsx] PASS!
[  mul-longlong] PASS!
[        my_add] PASS!
[        my_mul] 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 3.2.3 带 bug 的 runall.sh 结果

可以发现，这个 bug 还是很隐蔽的，正常的 test 很难测试出来。那么这时候就需要我们用究极无敌大杀器——diff-test。让我们开启 DIFF_TEST 试下

```
mystle@PANJU:~/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] FAIL! see add-longlong-log.txt for more information
[           add] FAIL! see add-log.txt for more information
[           bit] FAIL! see bit-log.txt for more information
[   bubble-sort] FAIL! see bubble-sort-log.txt for more information
[           div] FAIL! see div-log.txt for more information
[         dummy] FAIL! see dummy-log.txt for more information
[          fact] FAIL! see fact-log.txt for more information
[           fib] FAIL! see fib-log.txt for more information
[      goldbach] FAIL! see goldbach-log.txt for more information
[     hello-str] FAIL! see hello-str-log.txt for more information
[       if-else] FAIL! see if-else-log.txt for more information
[     leap-year] FAIL! see leap-year-log.txt for more information
[    load-store] FAIL! see load-store-log.txt for more information
[    matrix-mul] FAIL! see matrix-mul-log.txt for more information
[           max] FAIL! see max-log.txt for more information
[          min3] FAIL! see min3-log.txt for more information
[         mov-c] FAIL! see mov-c-log.txt for more information
[         movsx] FAIL! see movsx-log.txt for more information
[  mul-longlong] FAIL! see mul-longlong-log.txt for more information
[        my_add] FAIL! see my_add-log.txt for more information
[        my_mul] FAIL! see my_mul-log.txt for more information
[        pascal] FAIL! see pascal-log.txt for more information
[         prime] FAIL! see prime-log.txt for more information
[    quick-sort] FAIL! see quick-sort-log.txt for more information
[     recursion] FAIL! see recursion-log.txt for more information
[   select-sort] FAIL! see select-sort-log.txt for more information
[         shift] FAIL! see shift-log.txt for more information
[   shuixianhua] FAIL! see shuixianhua-log.txt for more information
[        string] FAIL! see string-log.txt for more information
[  sub-longlong] FAIL! see sub-longlong-log.txt for more information
[           sum] FAIL! see sum-log.txt for more information
[        switch] FAIL! see switch-log.txt for more information
[ to-lower-case] FAIL! see to-lower-case-log.txt for more information
[       unalign] FAIL! see unalign-log.txt for more information
[        wanshu] FAIL! see wanshu-log.txt for more information
```

图 3.2.4 带 bug 的 diff-test runall.sh 结果

可以发现，效果立竿见影，全部 fail。那么我们单独测试最简单的 dummy

```
Welcome to riscv32-NEMU!
For help, type "help"
Differential testing failed at PC = 0x80100004
Register 'sp' mismatch: ref = 0x80109004, dut = 0x80109008
$0      0x00000000      0
ra      0x00000000      0
sp      0x80109008      2148569096
gp      0x00000000      0
tp      0x00000000      0
t0      0x00000000      0
t1      0x00000000      0
t2      0x00000000      0
s0      0x00000000      0
s1      0x00000000      0
a0      0x00000000      0
a1      0x00000000      0
a2      0x00000000      0
a3      0x00000000      0
a4      0x00000000      0
a5      0x00000000      0
a6      0x00000000      0
a7      0x00000000      0
s2      0x00000000      0
s3      0x00000000      0
s4      0x00000000      0
s5      0x00000000      0
s6      0x00000000      0
s7      0x00000000      0
s8      0x00000000      0
s9      0x00000000      0
s10     0x00000000      0
s11     0x00000000      0
t3      0x00000000      0
t4      0x00000000      0
t5      0x00000000      0
t6      0x00000000      0
nemu: ABORT at pc = 0x80100004

[src/monitor/cpu-exec.c,31,monitor_statistic] total guest instructions = 2
qemu-system-riscv32: terminating on signal 15 from pid 38385 ()
 dummy
```

阅读报错信息，发现是 sp 的值比期望值多了 4，刚好就是我们制造的 bug。真得感谢 diff-test，太伟大了。

3.2.3 PA2.3: 运行打字小游戏，提交完整的实验报告

首先测试串口，十分正常

```
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

[src/monitor/cpu-exec.c,31,monitor_statistic] total guest instructions = 2188
qemu-system-riscv32: terminating on signal 15 from pid 38903 ()
make[1]: Leaving directory '/home/mystle/ics2019/nemu'
```

图 3.2.5 串口测试截图

然后测试时钟，也没有问题

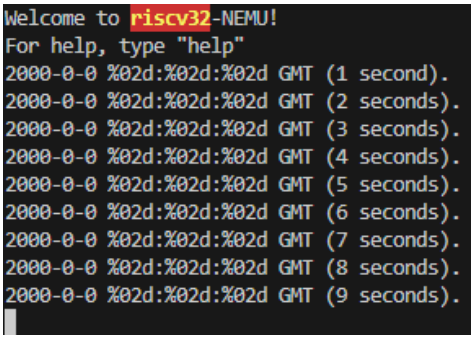


图 3.2.6 **时钟测试截图**

再测试键盘，很完美

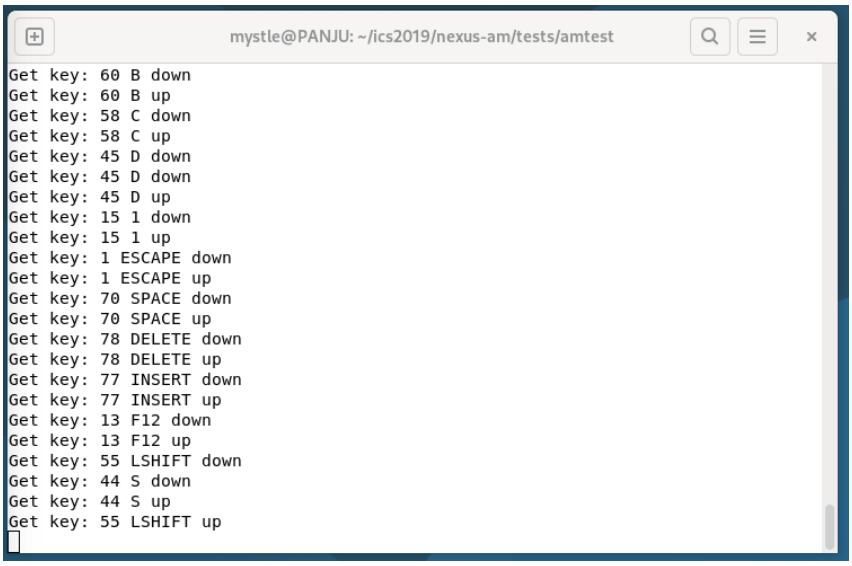


图 3.2.7 **键盘测试截图**

再测试 vga 是否正常，问题不大，就是这个帧数有点太低了。

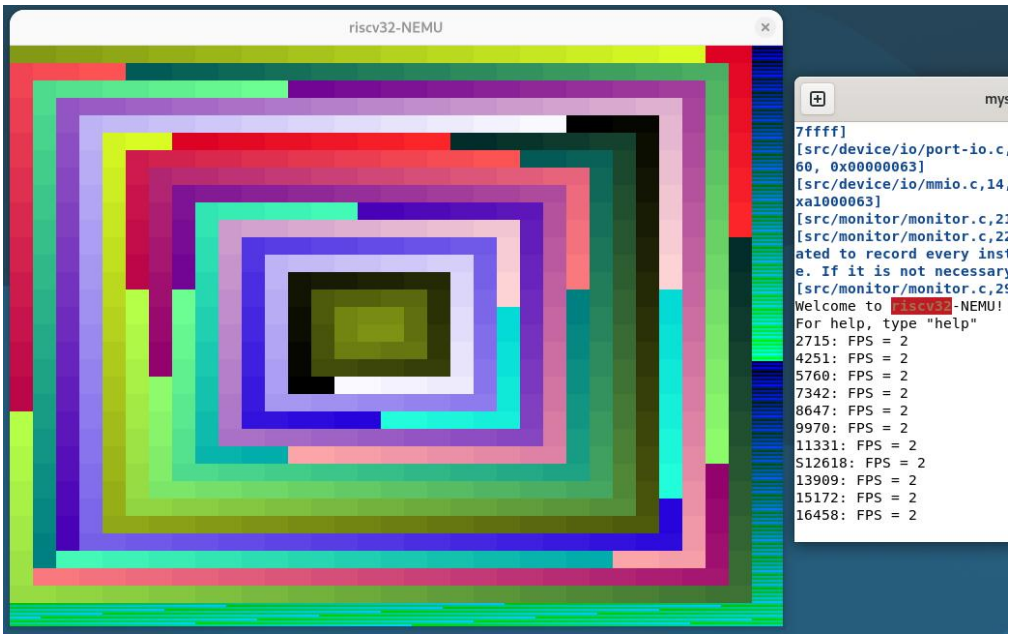


图 3.2.8 vga **测试截图**

然后 doc 里面还介绍了几个有趣的 benchmark，我们可以跑个分跟标准(i7-7700K)比较下：

```
Welcome to riscv32-NEMU!
For help, type "help"
Dhrystone Benchmark, Version C, Version 2.2
Trying 500000 runs through Dhrystone.
Finished in 4635 ms
------------------------------------------------
Dhrystone PASS          190 Marks
                    vs. 100000 Marks (i7-7700K @ 4.20GHz)
nemu: HIT GOOD TRAP at pc = 0x80100fb0

[src/monitor/cpu-exec.c,31,monitor_statistic] total guest instructions = 221985945
make[1]: Leaving directory '/home/mystle/ics2019/nemu'
```

图 3.2.9 Dhrystone 测试截图

```
Welcome to riscv32-NEMU!
For help, type "help"
Empty mainargs. Use "ref" by default
======= Running MicroBench [input *ref*] =======
[qsort] Quick sort: * Passed.
  min time: 501 ms [1020]
[queen] Queen placement: * Passed.
  min time: 765 ms [615]
[bf] Brainf**k interpreter: * Passed.
  min time: 4070 ms [581]
[fib] Fibonacci number: * Passed.
  min time: 7953 ms [356]
[sieve] Eratosthenes sieve: * Passed.
  min time: 9684 ms [406]
[15pz] A* 15-puzzle search: * Passed.
  min time: 1265 ms [354]
[dinic] Dinic's maxflow algorithm: * Passed.
  min time: 1508 ms [721]
[lzip] Lzip compression: * Passed.
  min time: 1245 ms [609]
[ssort] Suffix sort: * Passed.
  min time: 540 ms [834]
[md5] MD5 digest: * Passed.
  min time: 7241 ms [238]
------------------------------------------------
MicroBench PASS         573 Marks
                    vs. 100000 Marks (i7-7700K @ 4.20GHz)
Total time: 39395 ms
nemu: HIT GOOD TRAP at pc = 0x801041e0

[src/monitor/cpu-exec.c,31,monitor_statistic] total guest instructions = 1865089947
make[1]: Leaving directory '/home/mystle/ics2019/nemu'
```

图 3.2.10 MicroBench 测试截图

```
Welcome to riscv32-NEMU!
For help, type "help"
Running CoreMark for 1000 iterations
2K performance run parameters for coremark.
CoreMark Size    : 666
Total time (ms)  : 6688
Iterations       : 1000
Compiler version : GCC8.2.0
seedcrc          : 0x%04x
[0]crclist       : 0x%04x
[0]crcmatrix     : 0x%04x
[0]crcstate      : 0x%04x
[0]crcfinal      : 0x%04x
Finised in 6688 ms.
------------------------------------------------
CoreMark PASS       436 Marks
                vs. 100000 Marks (i7-7700K @ 4.20GHz)
nemu: HIT GOOD TRAP at pc = 0x80102498

[src/monitor/cpu-exec.c,31,monitor_statistic] total guest instructions = 306789279
make[1]: Leaving directory '/home/mystle/ics2019/nemu'
```

图 3.2.11 CoreMark 测试截图

可以发现，分数都低的离谱，最多的也就百分之一不到。
然后是提供的其他显示相关测试，包括两个游戏

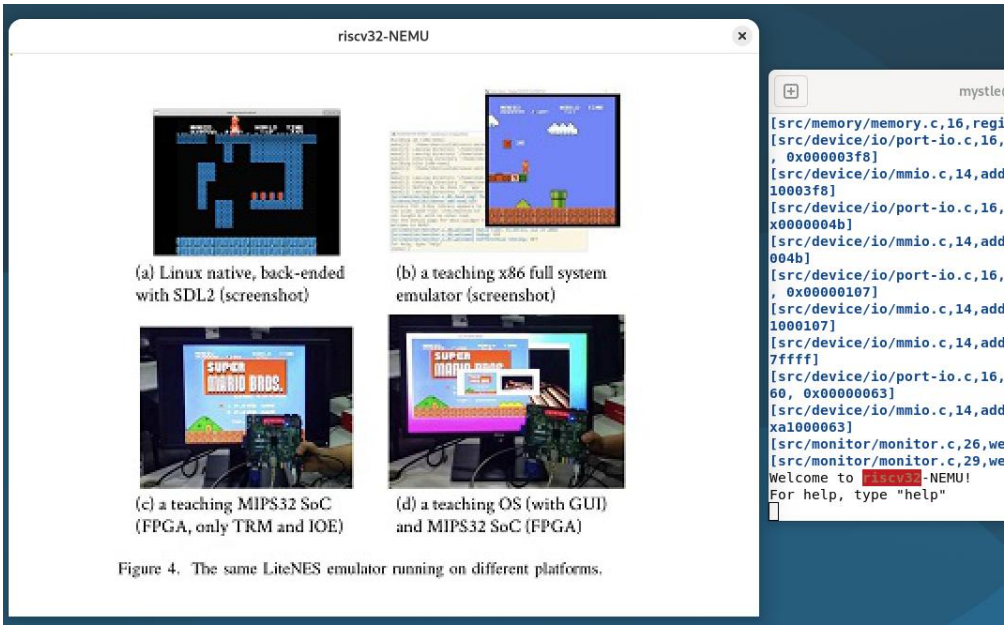


图 3.2.12 Slider 测试截图

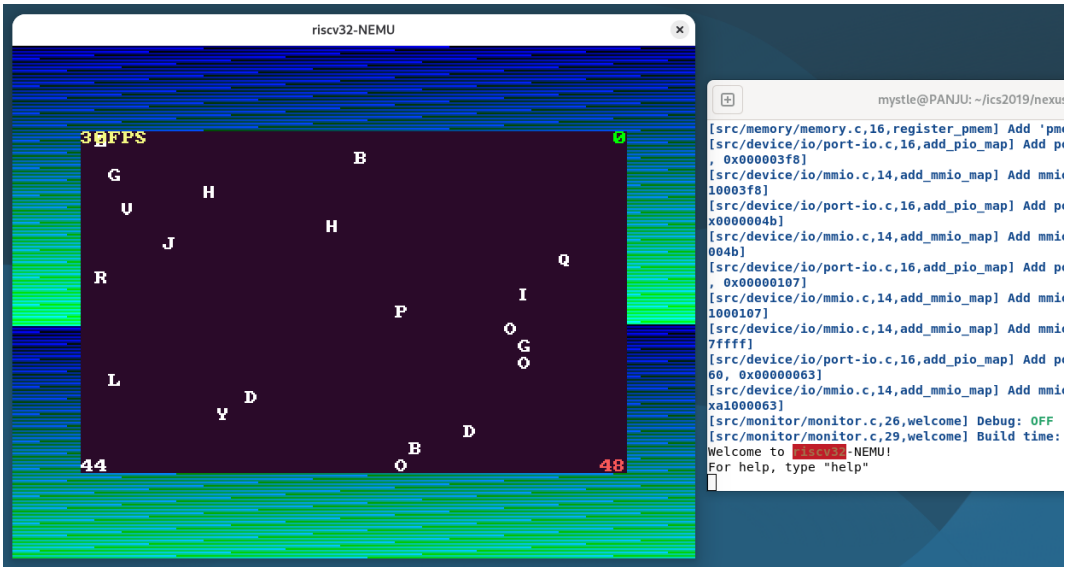


图 3.2.13 Typing 测试截图

图 3.2.14 litenes 测试截图

虽然我是想玩的，但是 8 帧还是太过于电竞了。。。

3.3 PA3 - 穿越时空的旅程: 批处理系统

3.3.1 PA3.1: 实现自陷操作_yield()及其过程

因为 3.3 较 3.1 多实现了文件系统和系统调用修改起来太过麻烦，故这里不再重新测试。

3.3.2 PA3.2: 实现用户程序加载和系统调用，支撑 TRM 程序运行

首先是 dummy 的测试，当完成 PA3.3 后，需要将 sys_exit()改回正常返回：

```
Welcome to riscv32-NEMU!
For help, type "help"
(nemu) c
[/home/mystle/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/mystle/ics2019/nanos-lite/src/main.c,15,main] Build time: 16:28:50, Jan 15 2026
[/home/mystle/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = %p, e
nd = %p, size = -2146420708 bytes
[/home/mystle/ics2019/nanos-lite/src/device.c,62,init_device] Initializing devices...
[/home/mystle/ics2019/nanos-lite/src/irq.c,21,init_irq] Initializing interrupt/exception ha
ndler...
[/home/mystle/ics2019/nanos-lite/src/proc.c,28,init_proc] Initializing processes...
loader: filename=/bin/dummy
[/home/mystle/ics2019/nanos-lite/src/loader.c,96,naive_uload] Jump to entry = 8300008c
Handling yield event
Program exited with status 0
nemu: HIT GOOD TRAP at pc = 0x801010e0
```

图 3.3.1 dummy 测试截图

然后是 hello 的测试：

```
Welcome to riscv32-NEMU!
For help, type "help"
(nemu) c
[/home/mystle/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/mystle/ics2019/nanos-lite/src/main.c,15,main] Build time: 16:26:16, Jan 15 2026
[/home/mystle/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = %p, e
nd = %p, size = -2146420644 bytes
[/home/mystle/ics2019/nanos-lite/src/device.c,62,init_device] Initializing devices...
[/home/mystle/ics2019/nanos-lite/src/irq.c,21,init_irq] Initializing interrupt/exception ha
ndler...
[/home/mystle/ics2019/nanos-lite/src/proc.c,28,init_proc] Initializing processes...
loader: filename=/bin/hello
[/home/mystle/ics2019/nanos-lite/src/loader.c,96,naive_uload] Jump to entry = 83000100
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!
```

图 3.3.2 hello 测试截图

顺利通过两个测试。

3.3.3 PA3.3: 运行仙剑奇侠传展示批处理系统，提交完整实验报告

首先先测试文件系统的正确性，跑 text 测试文件：

```
Welcome to riscv32-NEMU!
For help, type "help"
(nemu) c
[/home/mystle/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/mystle/ics2019/nanos-lite/src/main.c,15,main] Build time: 16:31:23, Jan 15 2026
[/home/mystle/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = %p, e
nd = %p, size = -2146420708 bytes
[/home/mystle/ics2019/nanos-lite/src/device.c,62,init_device] Initializing devices...
[/home/mystle/ics2019/nanos-lite/src/irq.c,21,init_irq] Initializing interrupt/exception ha
ndler...
[/home/mystle/ics2019/nanos-lite/src/proc.c,28,init_proc] Initializing processes...
loader: filename=/bin/text
[/home/mystle/ics2019/nanos-lite/src/loader.c,96,naive_uload] Jump to entry = 830002b8
PASS!!!
Program exited with status 0
nemu: HIT GOOD TRAP at pc = 0x801010e0
```

图 3.3.3 text 测试截图

然后测试 device(time, keyboard)的正确性，不知道为啥跑太快了只来得及截中间一部分

```
receive time event for the 356352th time: t 7082
receive time event for the 357376th time: t 7109
receive event: ku F
receive time event for the 358400th time: t 7133
receive event: kd A
receive event: kd D
receive event: kd S
receive event: ku S
receive event: ku A
receive time event for the 359424th time: t 7188
receive time event for the 360448th time: t 7207
receive event: ku D
receive event: kd F
receive time event for the 361472th time: t 7256
receive time event for the 362496th time: t 7281
receive time event for the 363520th time: t 7296
receive event: ku F
receive event: kd A
receive time event for the 364544th time: t 7325
receive time event for the 365568th time: t 7345
```

图 3.3.4 events 测试截图

最后测试的是 bmptest，输出 PA 的 logo：

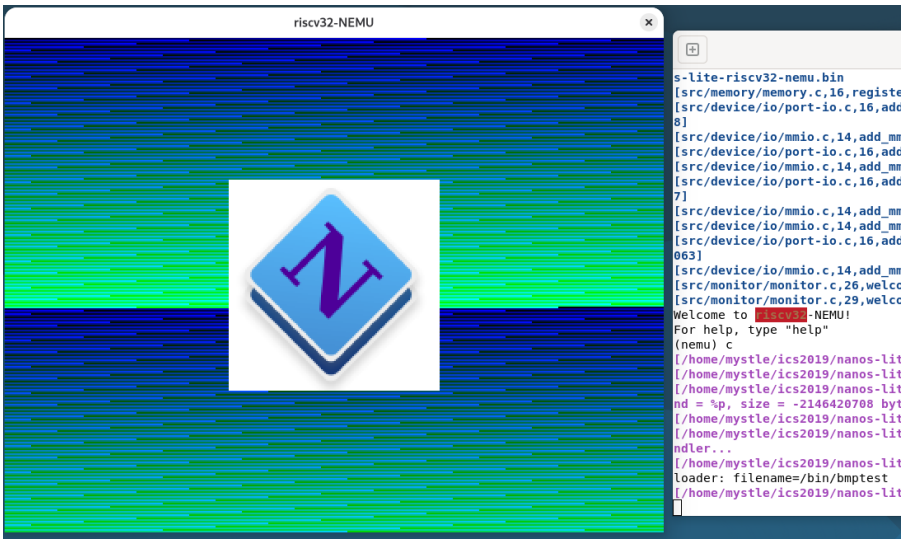


图 3.3.5 bmptest 测试截图

最后的最后，就到了紧张刺激的仙剑奇侠传测试，先测能不能正常显示

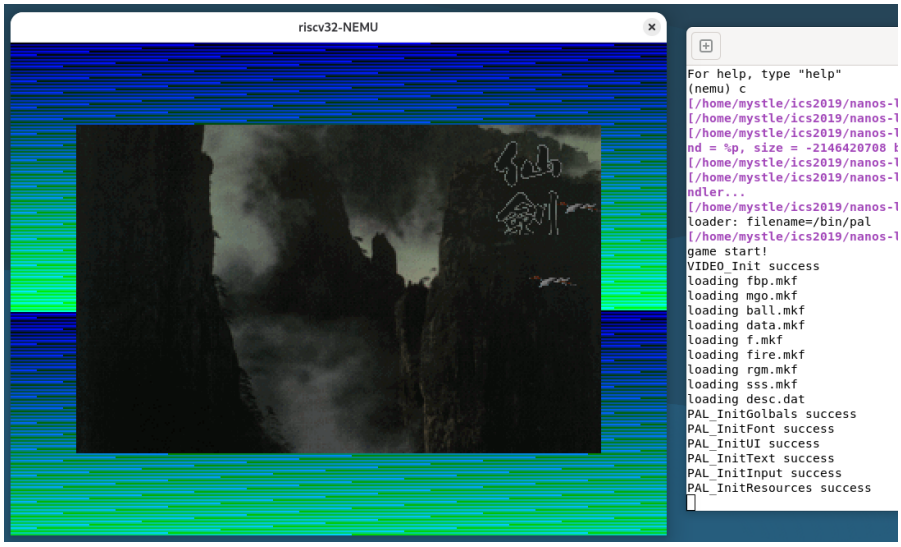
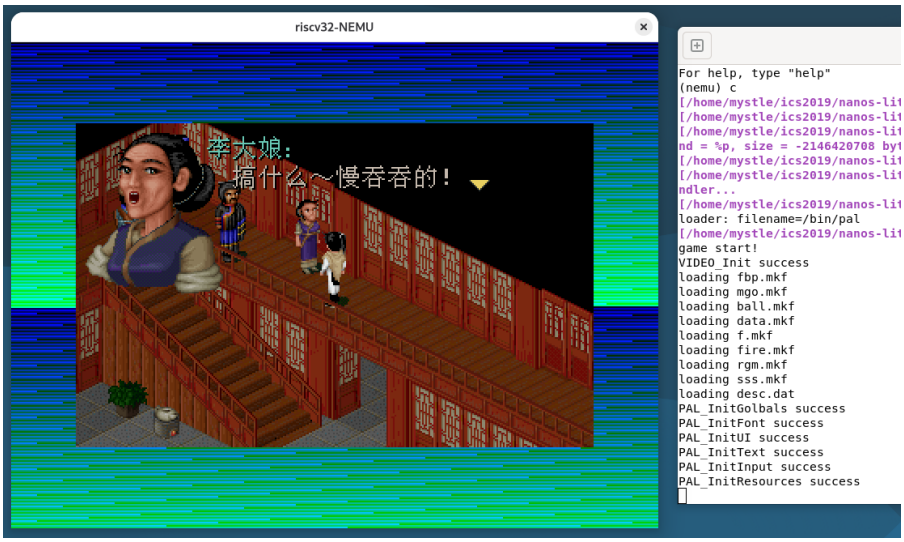


图 3.3.6 仙剑奇侠传入场动画

显示十分正常，就是这个帧率。。。但能玩，至少比红白机高多了。然后测试键盘，主要就是上下左右移动：



运行得十分顺畅，已经能正常游玩了，就是不能存档。

最后的最后，再测试下简易批处理系统，这里就只测试 dummy：

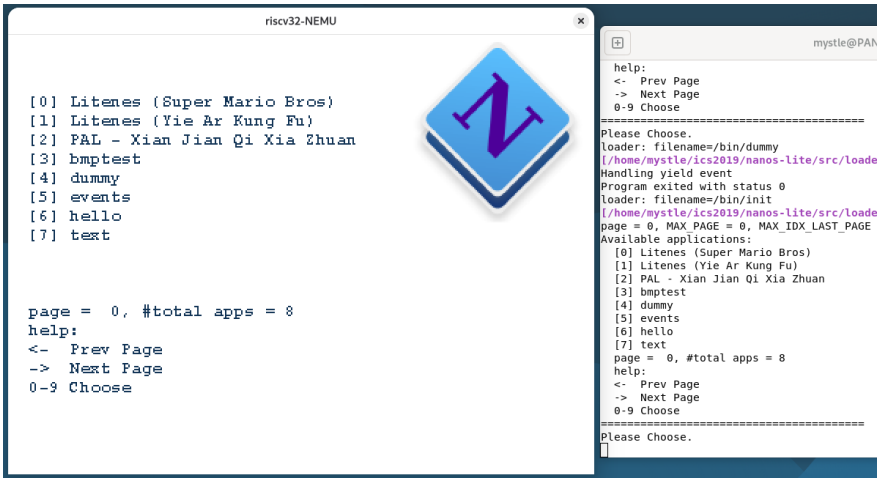


图 3.3.7 init 测试截图

4 总结与课程建议

4.1 实验总结

本次系统能力综合训练课程的实验过程，对我而言是一次痛并快乐着的“穿越”之旅。从 PA1 的零基础起步，到 PA3 成功运行《仙剑奇侠传》，我亲手构建了一个虽简陋但五脏俱全的 RISC-V32 计算机系统。这不仅验证了课本上枯燥的理论，更让我对“程序如何在计算机上运行”这一根本问题有了具象化的认知。

回首整个实验流程，每个阶段都有其独特的挑战与收获：

- **PA1：工具与基础的打磨。** 这一阶段主要是在熟悉 NEMU 的基础设施。虽然单步执行和寄存器打印相对简单，但表达式求值（尤其是正则表达式的处理和负数/解引用的逻辑）和监视点的链表管理，让我复习了大量 C 语言的底层操作。实现简易调试器的过程，也让我深刻理解了 GDB 等工具背后的工作原理。
- **PA2：指令集的构建与 Diff-test 的威力。** 这是代码量最大、细节最繁琐的一章。在实现 RISC-V32 指令集时，**符号扩展**的问题让我吃了不少苦头（如报告中提到的 B-Type 指令立即数处理）。特别是在运行 CoreMark 和 MicroBench 时，一旦出错往往难以定位。这里必须再次吹爆 diff-test 这个“究极无敌大杀器”，如果没有它来对比 QEMU 的状态，那个隐蔽的栈指针偏差 bug 可能要耗费我数倍的时间。此外，通过 Gemini 协助解决 QEMU 启动报错的经历，也让我意识到合理利用 AI 工具辅助 Debug 的重要性。
- **PA3：系统调用与操作系统的雏形。** 相比于 PA2 的硬件细节，PA3 更侧重于软硬协同。由于之前有过 PKE 实验的经验，我在实现异常处理和系统调用时相对顺手。虽然 PA3 的文件系统抽象相比 PKE 简化了许多（例如设备文件与普通文件的统一接口），但亲手打通 User App -> Nanos-lite -> AM -> NEMU 这一整条调用链带来的成就感是无与伦比的。当看到《仙剑奇侠传》的界面虽然以“感人”的帧率显示出来，键盘操作能够得到响应时，所有的 Debug 工作都变得值得了。

4.2 课程建议

总体而言，本课程的文档质量非常高，引导性强且循序渐进。为了让后续的学弟学妹们能有更好的体验，我结合自己的踩坑经历提出以下几点建议：

1. **关于 Diff-test 的引入时机与强调：** 建议在 PA2 的早期就更强力地推介 diff-test 工具。很多时候我们习惯于用 printf 调试，但在面对数以亿计的指令执行时，Log 几乎是不可读的。如果在实现基础指令（如 lw, sw）后就强制要求配置好 diff-test 环境，可能会大大减少学生在后续复杂指令实现上的无效调试时间。

2. **关于外设文档的细节补充：** 在 PA2.3 实现 VGA 部分时，关于 am 层面的 frame buffer 写入逻辑（如 x, y, w, h 的边界检查和内存映射计算）描述得比较简略。虽然能参考代码框架推断，但对于第一次接触显存映射的同学来说可能略显晦涩。此外，关于如何优化 VGA 性能（例如提示使用 memset 代替逐像素循环写入）也可以作为选做思考题明确列出，以解决“电竞帧数”的问题。

3. **关于环境配置的坑：** 我在配置环境时遇到了 QEMU 版本或参数导致的 Failed to set FD nonblocking 错误，这可能与 VirtualBox 或特定 Linux 发行版有关。建议在实验文档的 FAQ 部分增加一些关于 QEMU 高版本兼容性或参数配置的排错指南，避免大家在非代码逻辑的问题上卡太久。

总的来说，这是一门硬核但极具价值的课程，感谢老师和助教的答疑解惑！

电子签名： 

参考文献