

华中科技大学

2023

系统能力培养 课程实验报告

题 目: risc-v32 指令模拟器

专 业: 计算机科学与技术

班 级: CS2001 班

学 号: U202015299

姓 名: 刘玺语

电 话: (+1) 310-3085-376

邮 件: 1546868258@qq.com

完成日期: 2024-01-17



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	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	5
2.6	删除监视点	5
2.7	必答题	6
2.8	章节小结	6
3	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.c	7
3.2	实现指令，通过回归测试	8
3.3	输入输出	9
3.4	章节小结	10
4	穿越时空的旅程：批处理系统	11
4.1	实现自陷操作_yield()	11
4.2	实现系统调用	11
4.3	实现文件系统	12
4.4	在 NEMU 中运行仙剑奇侠传	13
4.5	展示批处理系统	14
4.6	章节小结	14
5	综合课设实验总结	15
	参考文献	16

1 课程实验概述

1.1 课设目的

实现简化的 32 位 risc-v 指令模拟器 (NEMU: NJU EMUlator)

支持的功能:

- i. 支持 riscv32 代码
- ii. 支持输入输出设备
- iii. 支持异常流处理
- iv. 支持文件系统

最终达成的效果:

- i. 简易调试器
- ii. 运行 C 程序
- iii. 运行图形游戏
- iv. 运行仙剑奇侠传

1.2 课设任务

PA1:

- i. 简易调试器
- ii. 表达式求值
- iii. 监视点与断点

PA2:

- i. 运行第一个 C 程序
- ii. 丰富指令集, 测试所有程序
- iii. 实现 I/O 指令, 测试打字游戏

PA3:

- i. 异常流处理
- ii. 用户程序与系统调用
- iii. 文件系统
- iv. 运行仙剑奇侠传

1.3 实验环境

本机配置:

处理器 Intel(R) Core(TM) i7-10875H CPU @ 2.30GHz

机带 RAM 32.0 GB 系统类型 64 位操作系统, 基于 x64 的处理器

虚拟机配置:

Ubuntu 20.04 LTS (Focal Fossa)(64-bit)——wiki 提供的镜像

内存大小: 8GB

2 开天辟地的篇章：最简单的计算机

本章节涉及的代码都位于 nemu 文件夹下，主要在 nemu/src/monitor/debug/中设计调试器，同时在 watchpoint.h 文件中新增数据结构，在 reg.c 中实现 isa_reg_display()

2.1 单步执行

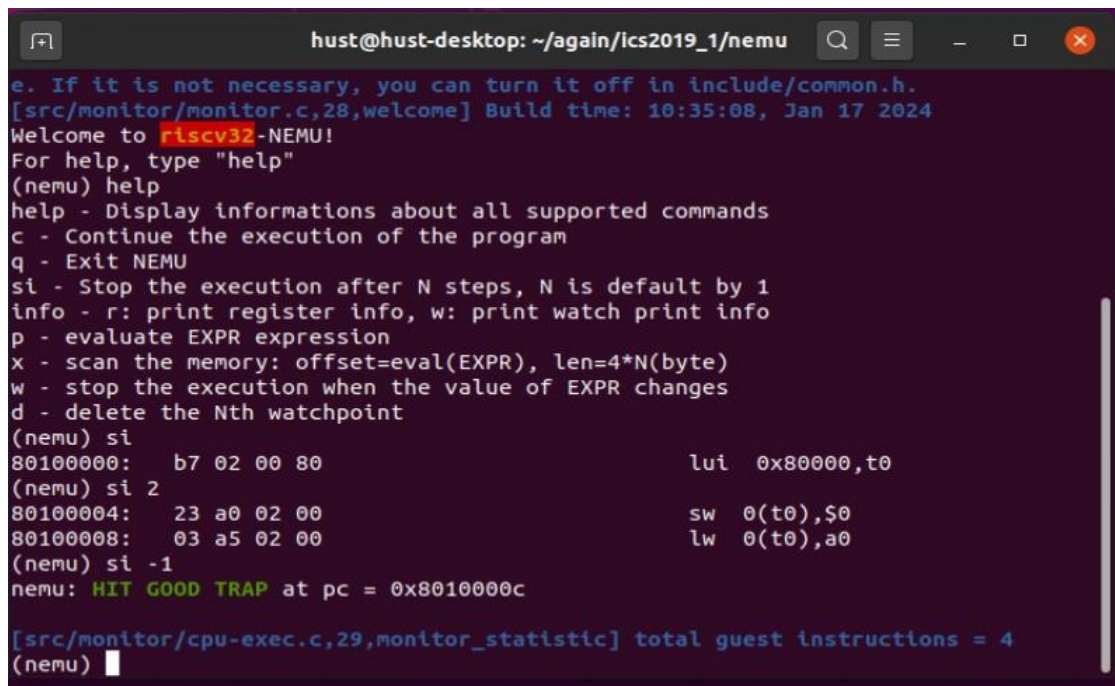
指令：si [N]

功能：连续单步执行 N 步，其中 N 为可选参数，默认值为 1.

实现：参考 cmd_c, 只需处理 char *arg 截取参数，检查参数，将 N 传给 cpu_exec(N)即可。

回答文档问题：为什么 void cpu_exec(uint_64_t n)在 cmd_c 中的入参为-1？这是因为无符号 64 位整型数下，-1 表示 $1 \ll 64 - 1$ ，是这一长度下能表示的最大整数，且代码长度也不可能超过这个数字。综上所述，这样的实现是合理的。

指令效果如图 2.1 所示（图中还包括了 cmd_help 的演示，较为简单不单开一节解释）



```
hust@hust-desktop: ~/again/ics2019_1/nemu
e. If it is not necessary, you can turn it off in include/common.h.
[src/monitor/monitor.c,28,welcome] Build time: 10:35:08, Jan 17 2024
Welcome to riscv32-NEMU!
For help, type "help"
(nemu) help
help - Display informations about all supported commands
c - Continue the execution of the program
q - Exit NEMU
si - Stop the execution after N steps, N is default by 1
info - r: print register info, w: print watch print info
p - evaluate EXPR expression
x - scan the memory: offset=eval(EXPR), len=4*N(byte)
w - stop the execution when the value of EXPR changes
d - delete the Nth watchpoint
(nemu) si
80100000: b7 02 00 80                                lui 0x80000,t0
(nemu) si 2
80100004: 23 a0 02 00                                sw 0(t0),$0
80100008: 03 a5 02 00                                lw 0(t0),a0
(nemu) si -1
nemu: HIT GOOD TRAP at pc = 0x8010000c
[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 4
(nemu) █
```

图 2.1 cmd_help 和 cmd_si 实机演示

2.2 打印程序状态

指令：info r|w

功能：输出当前寄存器（r）或者监视点（w）的状态。

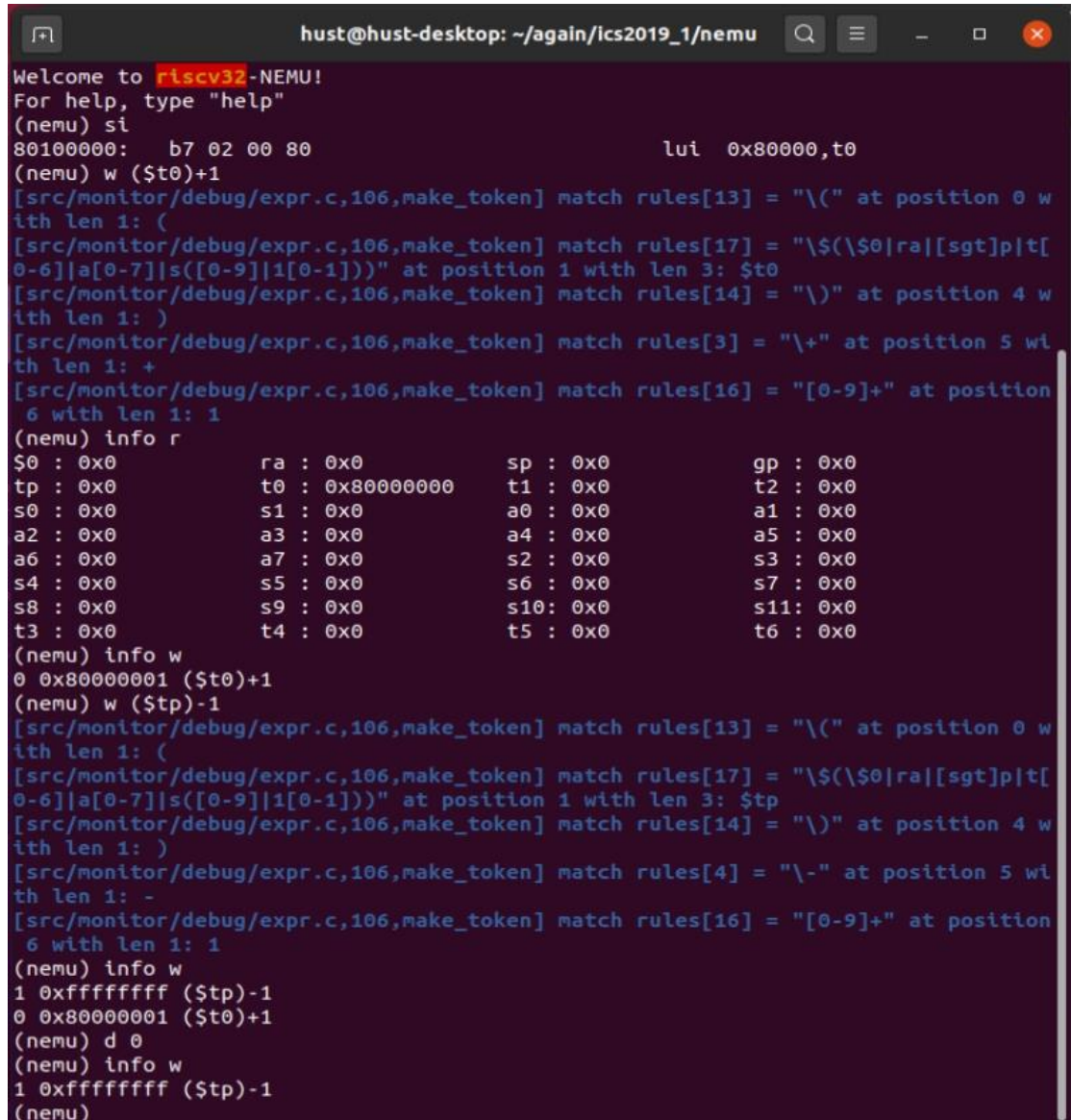
实现：在 watchpoint.c 和 reg.c 文件中分别实现各自的 display 函数，于 cmd_info 中检查参数后进行调用。

Watchpoint 的 display 函数：遍历已分配的监视点链表，输出每个监视点的

信息，包括监视点 ID、监视表达式、当前值

Isa_reg_display(): 在文件中定义了 riscv 架构的所有寄存器名，遍历寄存器名单，调用框架 isa_reg_str2val 接口读取寄存器值，输出信息。

以上两处实现不是难点，需注意的是输出的格式，例如在 printf 中通过 %-8x 限定 16 进制数固定占 8 个字符宽度且向左对齐。调试器的输出信息本意就是为了后续调试过程中更清晰，以便更快定位程序中的问题，这样做是很有必要的。



```
hust@hust-desktop: ~/again/ics2019_1/nemu
Welcome to riscv32-NEMU!
For help, type "help"
(nemu) si
80100000: b7 02 00 80          lui 0x80000,t0
(nemu) w ($t0)+1
[src/monitor/debug/expr.c,106,make_token] match rules[13] = "(" at position 0 with len 1: (
[src/monitor/debug/expr.c,106,make_token] match rules[17] = "$($|ra|[sgt]p|t[0-6]|a[0-7]|s([0-9]|1[0-1]))" at position 1 with len 3: $t0
[src/monitor/debug/expr.c,106,make_token] match rules[14] = ")" at position 4 with len 1: )
[src/monitor/debug/expr.c,106,make_token] match rules[3] = "+" at position 5 with len 1: +
[src/monitor/debug/expr.c,106,make_token] match rules[16] = "[0-9]+" at position 6 with len 1: 1
(nemu) info r
$0 : 0x0          ra : 0x0          sp : 0x0          gp : 0x0
tp : 0x0          t0 : 0x80000000    t1 : 0x0          t2 : 0x0
s0 : 0x0          s1 : 0x0          a0 : 0x0          a1 : 0x0
a2 : 0x0          a3 : 0x0          a4 : 0x0          a5 : 0x0
a6 : 0x0          a7 : 0x0          s2 : 0x0          s3 : 0x0
s4 : 0x0          s5 : 0x0          s6 : 0x0          s7 : 0x0
s8 : 0x0          s9 : 0x0          s10: 0x0          s11: 0x0
t3 : 0x0          t4 : 0x0          t5 : 0x0          t6 : 0x0
(nemu) info w
0 0x800000001 ($t0)+1
(nemu) w ($tp)-1
[src/monitor/debug/expr.c,106,make_token] match rules[13] = "(" at position 0 with len 1: (
[src/monitor/debug/expr.c,106,make_token] match rules[17] = "$($|ra|[sgt]p|t[0-6]|a[0-7]|s([0-9]|1[0-1]))" at position 1 with len 3: $tp
[src/monitor/debug/expr.c,106,make_token] match rules[14] = ")" at position 4 with len 1: )
[src/monitor/debug/expr.c,106,make_token] match rules[4] = "-" at position 5 with len 1: -
[src/monitor/debug/expr.c,106,make_token] match rules[16] = "[0-9]+" at position 6 with len 1: 1
(nemu) info w
1 0xffffffff ($tp)-1
0 0x800000001 ($t0)+1
(nemu) d 0
(nemu) info w
1 0xffffffff ($tp)-1
(nemu)
```

图 2.2 cmd_info, cmd_w, cmd_d 的实机演示

2.3 扫描内存

指令: x <offset> <expr>

功能: 扫描内存中以 offset 为首地址，expr 表达式计算结果为 N，N 个 4 字节的内容。

实现: 调用框架提供的 vaddr_read 函数，expr 求值部分在“表达式求值”部分完成，循环读取内容并输出即可，格式类似 info 指令。


```

Welcome to riscv32-NEMU!
For help, type "help"
(nemu) x 0x80000000 4*8
[src/monitor/debug/expr.c,106,make_token] ma
at position 0 with len 10: 0x80000000
[src/monitor/debug/expr.c,106,make_token] ma
0 with len 1: 4
[src/monitor/debug/expr.c,106,make_token] ma
th len 1: *
[src/monitor/debug/expr.c,106,make_token] ma
2 with len 1: 8
0x80000000: 0x0      0x80000004: 0x0
0x80000008: 0x0      0x8000000c: 0x0
0x80000010: 0x0      0x80000014: 0x0
0x80000018: 0x0      0x8000001c: 0x0
0x80000020: 0x0      0x80000024: 0x0
0x80000028: 0x0      0x8000002c: 0x0
0x80000030: 0x0      0x80000034: 0x0
0x80000038: 0x0      0x8000003c: 0x0
0x80000040: 0x0      0x80000044: 0x0
0x80000048: 0x0      0x8000004c: 0x0
0x80000050: 0x0      0x80000054: 0x0
0x80000058: 0x0      0x8000005c: 0x0
0x80000060: 0x0      0x80000064: 0x0
0x80000068: 0x0      0x8000006c: 0x0
0x80000070: 0x0      0x80000074: 0x0
0x80000078: 0x0      0x8000007c: 0x0

```

图 2.3 cmd_x 实机演示

2.4 表达式求值

指令: p <expr>

功能: 对合法的表达式进行求值, 支持基本的四则运算, 比较, 逻辑运算, *解引用, 以及\$开头的字符串使用寄存器值。

实现: 根据文档说明, 可以将求值操作分为以下几个流程: 词法分析 (将表达式拆分成 token), 特殊处理解引用操作 (改写部分 token 的 type 字段), 调用 eval 函数求值。Eval 函数取 p、q 两数表示求解第 p 到第 q 个 token 表示的子表达式。为了实现 Eval 函数, 我们需要做到: 判断表达式是否被一对匹配的括号包围 (计数栈实现, 当遇到左括号时计数, 当遇到右括号时检查计数), 判断当前表达式的主操作符 (找出当前表达式中优先级最低的操作符, 过程中需要注意在括号内外的同一操作符应当看作不同优先级)。在此之后, 只要根据主操作符将表达式分解成左右两个子表达式递归求解并组合即可。

词法分析部分框架已经准备好了代码, 只需要在 rules 数组中补充匹配规则即可, 查询正则表达式写法匹配所有支持的操作, 除此以外的会作为非法 token 并报错。

对于字面值, 调用 strtoul 函数进行转化, 避免重复造轮子。

```

(nemu) c
nemu: HIT GOOD TRAP at pc = 0x8010000c

[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 4
(nemu) p $t0+(4+3*(2-1))
[src/monitor/debug/expr.c,106,make_token] match rules[17] = "\\$(\\$0|ra|[sgt]p|t[0-6]|a[0-7]|s([0-9]|1[0-1]))" at position 0 with len 3: $t0
[src/monitor/debug/expr.c,106,make_token] match rules[3] = "\\+" at position 3 with len 1: +
[src/monitor/debug/expr.c,106,make_token] match rules[13] = "\\(" at position 4 with len 1: (
[src/monitor/debug/expr.c,106,make_token] match rules[16] = "[0-9]+" at position 5 with len 1: 4
[src/monitor/debug/expr.c,106,make_token] match rules[3] = "\\+" at position 6 with len 1: +
[src/monitor/debug/expr.c,106,make_token] match rules[16] = "[0-9]+" at position 7 with len 1: 3
[src/monitor/debug/expr.c,106,make_token] match rules[1] = "\\*" at position 8 with len 1: *
[src/monitor/debug/expr.c,106,make_token] match rules[13] = "\\(" at position 9 with len 1: (
[src/monitor/debug/expr.c,106,make_token] match rules[16] = "[0-9]+" at position 10 with len 1: 2
[src/monitor/debug/expr.c,106,make_token] match rules[4] = "\\-" at position 11 with len 1: -
[src/monitor/debug/expr.c,106,make_token] match rules[16] = "[0-9]+" at position 12 with len 1: 1
[src/monitor/debug/expr.c,106,make_token] match rules[14] = "\\)" at position 13 with len 1: )
[src/monitor/debug/expr.c,106,make_token] match rules[14] = "\\)" at position 14 with len 1: )
-2147483641
(nemu) █

```

图 2.5 cmd_expr 实机演示（蓝色字符是正则表达式匹配成功的输出）

2.5 设置监视点

指令：w <expr>

功能：新增一个对于 expr 的监视点，在单步执行过程中遇到该值变化则暂停。

实现：从_free链表中取出一个wp结点，插入head链表即可。若成功返回了wp的地址到cmd_w，则调用先前实现的expr()接口计算表达式的值并赋值。

为了真正实现监视点功能，还需修改cmd_si函数的逻辑，将cpu_exec(N)改为for循环执行N次，每次执行后调用watchpoint_update()函数，返回是否有监视点表达式值发生了变化，若有则break退出循环。Update的函数即遍历head为首的链表，逐个计算每个监视点的值（简单实现，若监视点达到上限时可能影响程序性能）。

演示效果如图 2.2 所示。

2.6 删除监视点

指令：d <wp_id>

功能：删除指定id的监视点

实现：链表删除操作

演示效果如图 2.2 所示。

2.7 必答题

选择 ISA=riscv32

$500 \times 0.9 \times 20 \times (30 - 10) = 180000s$

指令格式:

R 寄存器, I 立即数和 load, S store, B branch 条件跳转, U 长立即数, J jump 无条件跳转。

LUI 指令: 高位立即数加载。将符号位扩展的 20 位表示的立即数左移 12 位, 写入 x[rd].

Find ./nemu -name “*.c” -o -name “*.h” | xargs cat | wc -l

“回到过去”操作, 使用 git log 查看 commit, git revert 回滚

-Wall 表示 warning all, 编译时报告所有警告, -Werror 表示将 warning 看作 error, warning 也会引起编译失败。其作用是更严格的检查不规范的代码, 保证了项目的安全性。

2.8 章节小结

在 pal 中实现了简易调试器, 其中表达式求值部分很有挑战性, 词法分析的部分让我回想起了编译原理实验的过程, 说明文档提供了 eval 函数的设计框架, 进一步体现了先思考后编码的重要性。另外也学到了 string 库中的一系列先前不常用的函数, strtok 对 arg 进行分词, strtoul 是 atoi 的加强版本, 会自动检测进制。

3 简单复杂的机器：冯诺依曼计算机系统

本章节的编码涉及 nexus-am，指令实现部分主要在 decode（译码）和 all-instr 部分。

3.1 运行第一个 C 程序——dummy.c

在 nexus-am/tests/cputest 目录下输入 `make ARCH=riscv-nemu ALL=dummy run` 进行编译并投入运行。第一次运行时在 `pc=0x8010000c` 处 abort，没有实现程序对应处的指令。打开 `dummy-riscv32-nemu.txt` 文件可以得知，需要补充实现 `li,auipc,addi,jal,mv,sw,jalr` 指令（这部分指令也是大萝卜的计组课设中的老朋友了，尤其是 `auipc` 给我留下了深刻的印象，由于早期实现时没有注意细节，一直到检查前才发现是整个指令的问题）。

阅读文档可以得知，向 `nemu` 中添加新指令，要在 `opcode_table` 中填写 `makeDHelper`，`makeEHelper` 以及操作数宽度。在此之后用 RTL 实现正确的执行辅助函数。需要注意的是，RTL 指令要遵守文档中的调用约定。

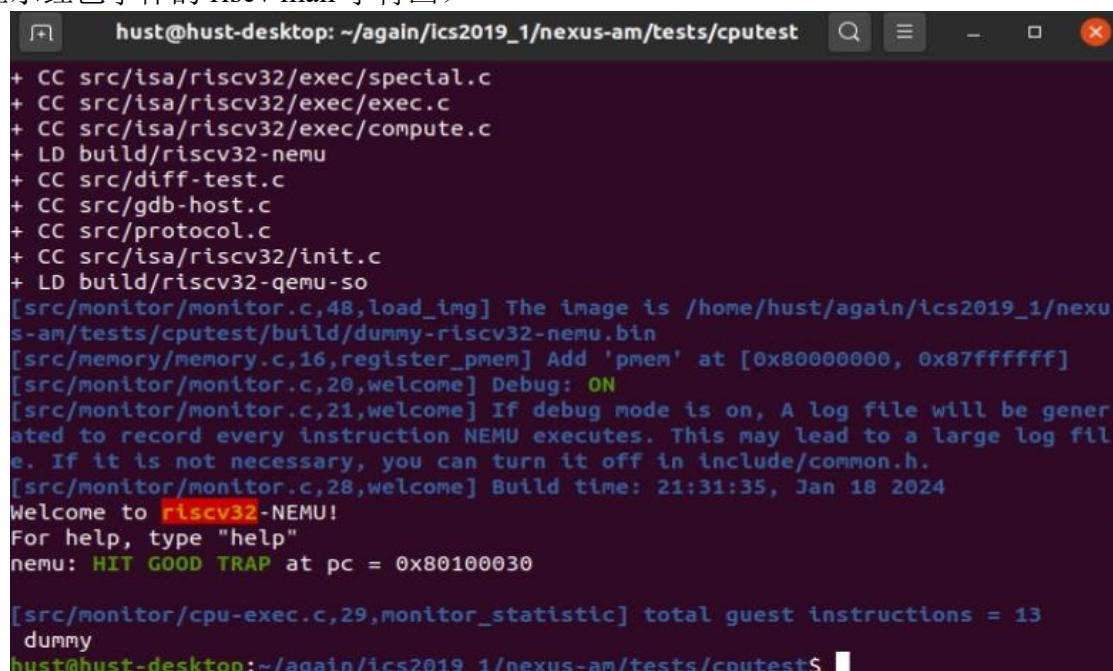
也是在这部分刷新了我对于宏定义的认识，仔细阅读之后才明白是通过很多宏定义来完成了一部分函数调用和定义的工作，使得代码量大大减少，也更为精简了。

`li,mv,addi` 三个指令的 `opcode` 相同，需要在 `makeEHelper` 函数中通过 `rs1` 和 `simmml_0` 来进一步区分。

`Auipc` 和 `lui` 指令有相似之处，但 `auipc` 需要将立即数于读取的 `pc` 值相加后的结果存入目的寄存器

`Jal` 和 `jalr` 指令相似，按照手册实现即可。

上述工作均完成无误后，重新运行 `dummy` 程序可以看到如下结果（失败时则显示红色字体的 `riscv man` 字符画）



```
hust@hust-desktop: ~/again/ics2019_1/nexus-am/tests/cputest
+ 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
+ CC src/diff-test.c
+ CC src/gdb-host.c
+ CC src/protocol.c
+ CC src/isa/riscv32/init.c
+ LD build/riscv32-qemu-so
[src/monitor/monitor.c,48,load_img] The image is /home/hust/again/ics2019_1/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: 21:31:35, Jan 18 2024
Welcome to riscv32-NEMU!
For help, type "help"
nemu: HIT GOOD TRAP at pc = 0x80100030

[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 13
dummy
hust@hust-desktop:~/again/ics2019_1/nexus-am/tests/cputest$
```

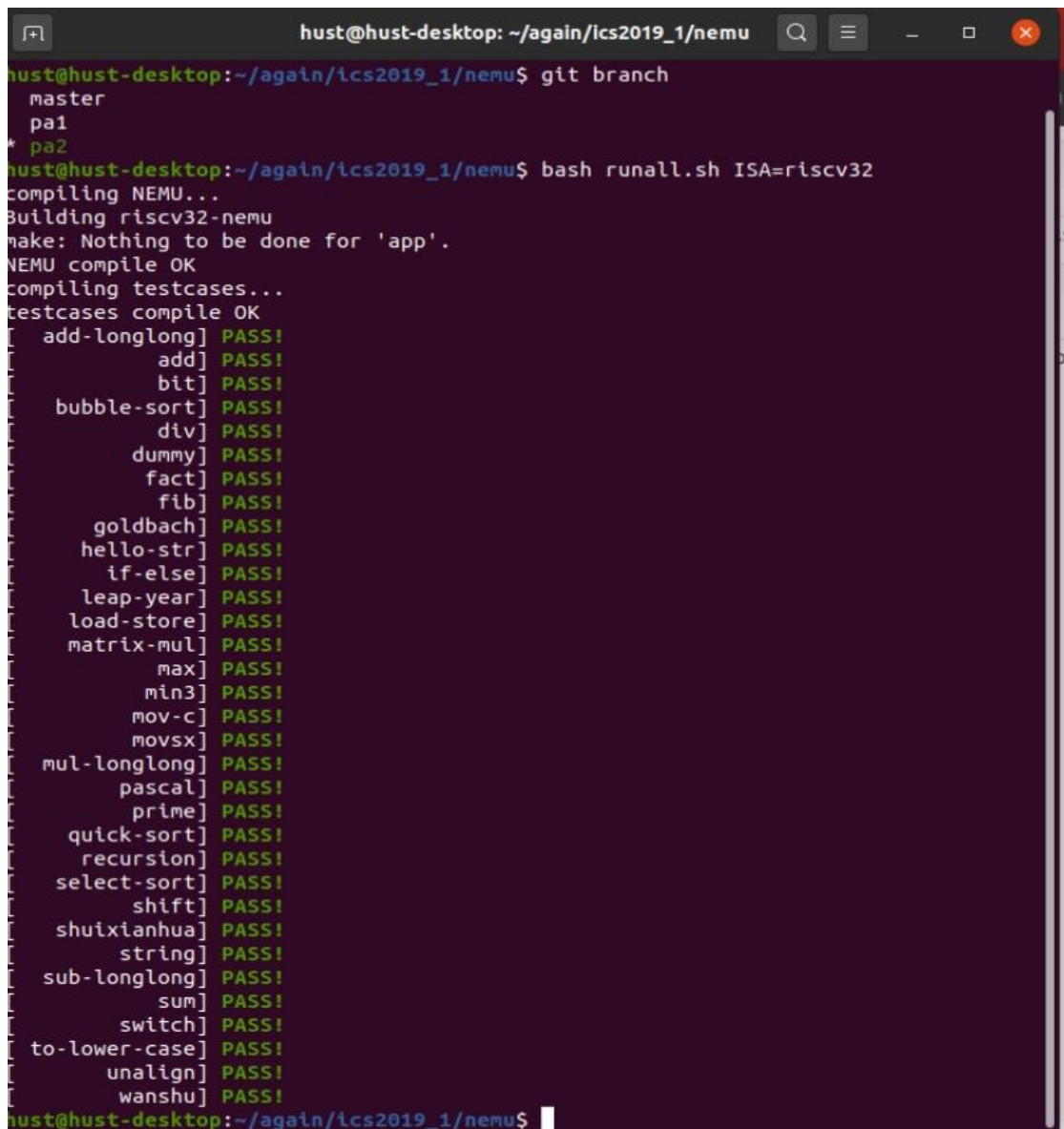
图 3.1 dummy 正确执行

3.2 实现指令，通过回归测试

这一部分需要更完整地实现其余指令，为后续阶段运行用户程序做准备。单论每一条指令的实现，和上一小节描述的过程都是差不多的：跑测试程序，检查 abort 的行数出现的什么问题，没实现的代码查询 riscv 手册，opcode 添加到表中，再添加 make_EHelper 和 make_DHelper 的宏调用。实现但是出错的甚至直接引起 coredump，经检查是手滑写错下标导致越界引起的...

除此之外遇到比较困难的是 hello-str，除了指令之外需要自己实现 sprintf() 函数，一开始想着自己实现但是遇到的困难有点大，最后参考了 vsprintf 函数和 va_list 结构进行了实现，获得每个参数后对其进行相应的格式化处理，最终调用 _putc 函数显示结果。

上述未提及的还有零零碎碎的各种小 bug，一一进行调试处理后运行 runall.sh 测试脚本，演示结果如图 3.2 所示。




```
hust@hust-desktop: ~/again/lcs2019_1/nemu
hust@hust-desktop:~/again/lcs2019_1/nemu$ git branch
  master
  pa1
* pa2
hust@hust-desktop:~/again/lcs2019_1/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:~/again/lcs2019_1/nemu$
```

图 3.2 回归测试

3.3 输入输出

定义 HAS_IOE 表示存在输入输出设备，即可运行 Hello World 程序。

实现时钟功能：在 `_am_timer_init` 中初始化 `boot_time`，再在 `switch` 的 `_DEVREG_TIMER_UPDATE` 中，求当前时间-初始时间，结果赋值给 `uptime->io`。
演示效果如图 3.3 所示：



```
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).
```

图 3.3 real-time 实机演示效果

实现键盘功能：`_DEVREG_INPUT_KBD` 通过 `inl(KBD_ADDR)` 从 MMIO 读取键盘码，比较 `KEYDWON_MASK` 检查是否按下

实现颜色动画：`_DEVREG_VIDEO_INFO` 通过 `inl(SCREEN_ADDR)` 获取屏幕的高、宽信息，从低到高分别占 2 字节。`_DEVREG_VIDEO_FBCTL` 中通过 `ctl` 获取需要绘制的坐标和长宽，以及像素信息 `pixels`，通过 `screen_width` 和 `screen_height` 信息获取屏幕尺寸，之后就是逐个拷贝到 MMIO 中的操作了，注意实现 `vga_io_handler`，更新屏幕。这部分实验的操作让我回想起了 22 年秋季学期的嵌入式开发课程，当时涉及屏幕显示操作的部分，也是实现了一系列绘图函数
演示效果如图 3.4,3.5 所示。

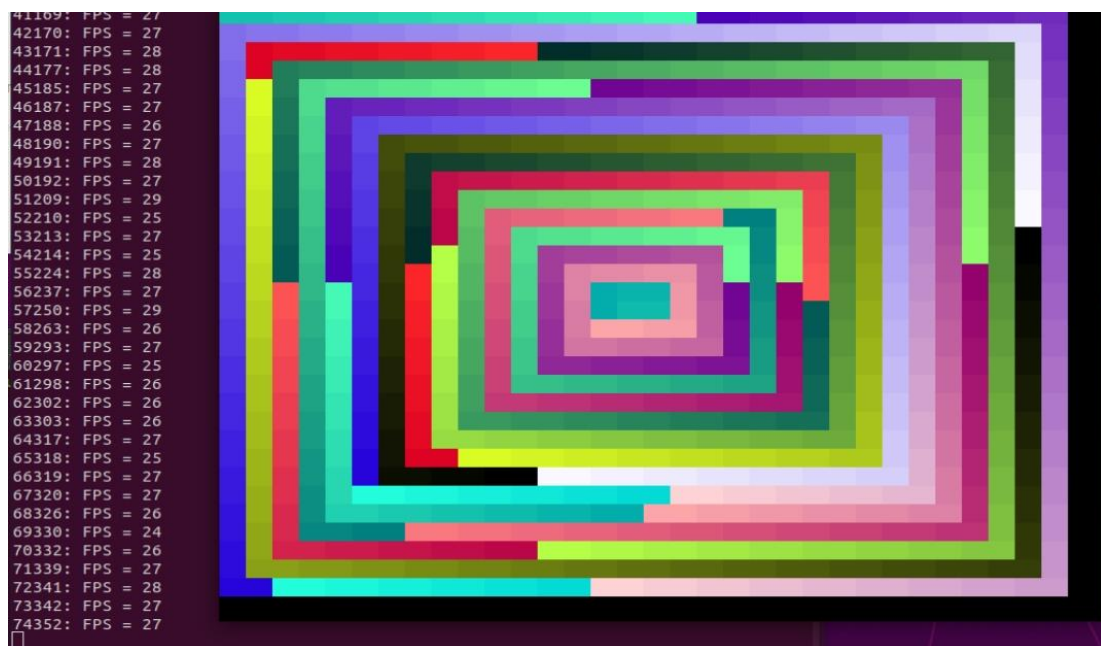


图 3.4 display 实机演示效果

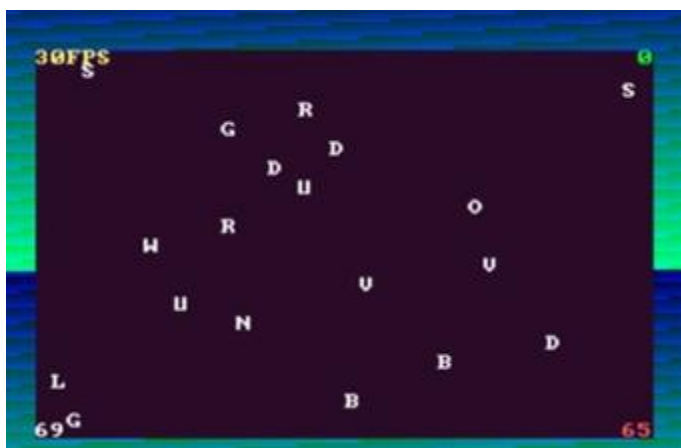


图 3.5 打字游戏实机演示效果

3.4 章节小结

本章节的内容和计算机组成原理以及嵌入式开发课程息息相关，其中指令实现的部分让我回想起往返于 `logism/excel/riscv-man` 之间的那两个星期。指令实现的大部分工作其实是类似的，查文档，查代码，找对应的文件编辑，需要做到的就是小心小心再小心，相似的指令之间看错一点就会导致最终的程序测试失败。

“肉眼观察法”这一陋习恶习在系统开发这一层级（或者说任一层级）都应该立刻改正，不如趁早学习各种调试工具的使用（这也包括了学习框架代码的能力：`DiffTest`）。

4 穿越时空的旅程：批处理系统

4.1 实现自陷操作_yield()

Risc-v 架构下自陷操作需要实现 `ecall` 环境调用指令、`sret` 管理员模式返回指令以及状态寄存器相关指令，过程类似 PA2 的实现。

需要注意的是，设置 IDEX (`SYSTEM`, `system`)，之后实现 `make_E` 和 `make_DHelper` 即可，`ecall` 和 `sret` 指令通过 `funct3` 字段区分，而一系列 CSR 指令则较为相似，`get` 和 `write csr` 函数完成了 csr 寄存器的读写。

这里文档提到需要调整上下文结构体 (`_Context`) 中的成员顺序，调整后的顺序依次为：`gpr`, `cause`, `status`, `epc` 和 `as`, `Context` 用于自陷前保护上下文和返回后的恢复。

事件分发 `_am_irq_handle()` 中，判断异常号来识别自陷异常，将时间编号 `ev.event` 设置为 `_EVENT_YIELD` 表示自陷事件。而后在事件处理函数 (`do_event`) 中会识别到 `yield` 事件并输出信息。

完成上述实现后重新运行 `Nanos-lite` 程序，可以看到输出的信息和返回现场后触发的 `panic`

演示效果如图 4.1 所示。

```
[/home/hust/Desktop/ics2019/nanos-lite/src/device.c,35,init_device] Initializing devices...
[/home/hust/Desktop/ics2019/nanos-lite/src/irq.c,20,init_irq] Initializing interrupt/exception ha
[/home/hust/Desktop/ics2019/nanos-lite/src/proc.c,25,init_proc] Initializing processes...
[/home/hust/Desktop/ics2019/nanos-lite/src/main.c,33,main] Finish initialization
Self trap!
[/home/hust/Desktop/ics2019/nanos-lite/src/main.c,39,main] system panic: Should not reach here
nemu: HIT BAD TRAP at pc = 0x801006f0
```

图 4.1 自陷异常和上下文保存与恢复实机演示

4.2 实现系统调用

`Syscall` 实现和 `yield` 类似。首先需要在 `_am_irq_handle` 函数中新增 `switch(c->case)` 的 `SYS_exit`, `SYS_yield`, `SYS_write`, `SYS_brk` 四个 `case`, 同时将 `event` 设置为 `_EVENT_SYSCALL` 之后就可以在 `do_event` 函数中对应的分支调用 `do_syscall` 完成系统调用。

此处需要使用到上下文中的 `GPR1~4` 以及 `GPRx`，分别对应了系统调用的类型，参数 1~3 以及系统调用返回值，文档给出了具体方案，实现完毕后演示效果如图 4.2 所示。

```
[/home/hust/Desktop/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start
[/home/hust/Desktop/ics2019/nanos-lite/src/device.c,35,init_device] Initializing devices...
[/home/hust/Desktop/ics2019/nanos-lite/src/irq.c,20,init_irq] Initializing interrupt/excep
[/home/hust/Desktop/ics2019/nanos-lite/src/proc.c,25,init_proc] Initializing processes...
[/home/hust/Desktop/ics2019/nanos-lite/src/loader.c,29,naive_uoload] Jump to entry = 830001
Hello World!
Hello World from Navy-apps for the 2th time!
Hello World from Navy-apps for the 3th time!
Hello World from Navy-apps for the 4th time!
Hello World from Navy-apps for the 5th time!
Hello World from Navy-apps for the 6th time!
```

图 4.2 Syscall 实机演示效果

4.3 实现文件系统

为了实现功能完备的文件系统，新增了以下系统调用：`fs_open`, `fs_read`, `fs_close`, `fs_write` 以及 `fs_lseek`, 文档描述较为全面，此处不再赘述。需要注意的是，`Finfo` 结构的 `open_offset` 以及 `file_tables` 数组的数据需要进行一定的修改（初次实现时遇到了编译以及越界问题）。

解决以上问题后，运行 `bin/text` 程序则能成功输出 `PASS!!!` 信息，集体情况如图 4.3 所示。

```
[/home/hust/Desktop/lcs2019/nanos-lite/src/math.c,15,math] built time: 2020-09-20 11:00:59, Jan 11 2022
[/home/hust/Desktop/lcs2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = , end = , size = -2146423336 by
[/home/hust/Desktop/lcs2019/nanos-lite/src/device.c,49,init_device] Initializing devices...
[/home/hust/Desktop/lcs2019/nanos-lite/src/irq.c,19,init_irq] Initializing interrupt/exception handler...
[/home/hust/Desktop/lcs2019/nanos-lite/src/proc.c,27,init_proc] Initializing processes...
[/home/hust/Desktop/lcs2019/nanos-lite/src/fs.c,50,fs_open] open file : /bin/text
[/home/hust/Desktop/lcs2019/nanos-lite/src/loader.c,49,naive_uoload] Jump to entry = 830002f4
[/home/hust/Desktop/lcs2019/nanos-lite/src/fs.c,50,fs_open] open file : /share/texts/num
PASS!!!
nemu: HIT GOOD TRAP at pc = 0x80100dc4
```

图 4.3 fs 相关 syscall 实机演示效果

下一步我们要对设备输入进一步抽象，将其抽象为文件。为此需要实现 `events_read()` 函数，根据 `keycode` 来判断按键事件(类似 PA2 IO 实现键盘输入)，而后将相应信息拷贝到 `buf` 数组。使用 Nanos-lite 运行 `bin/events` 测试，演示效果如图 4.4 所示。

```
Start to receive events...
receive time event for the 1024th time: t 4030
receive time event for the 2048th time: t 5262
receive event: kd 0
receive event: ku 0
receive time event for the 3072th time: t 6422
receive event: kd K
receive event: ku K
receive time event for the 4096th time: t 7555
```

图 4.4 键盘事件实机演示效果

最后一步将 VGA 抽象为文件（实现步骤文档给出的较为完善，不再赘述）Nanos-lite 加载 `bin/bmptest` 进行测试，演示效果如图 4.5 所示。

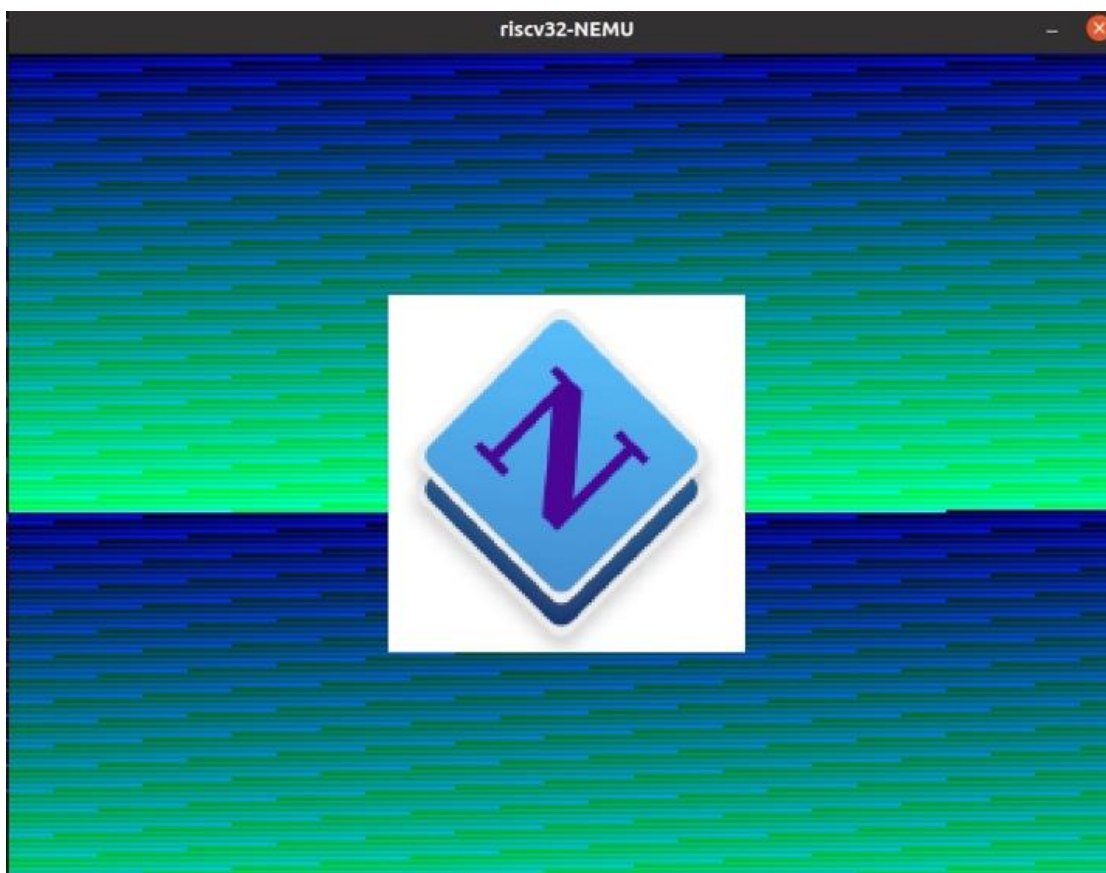


图 4.5 VGA 实机演示效果

4.4 在 NEMU 中运行仙剑奇侠传

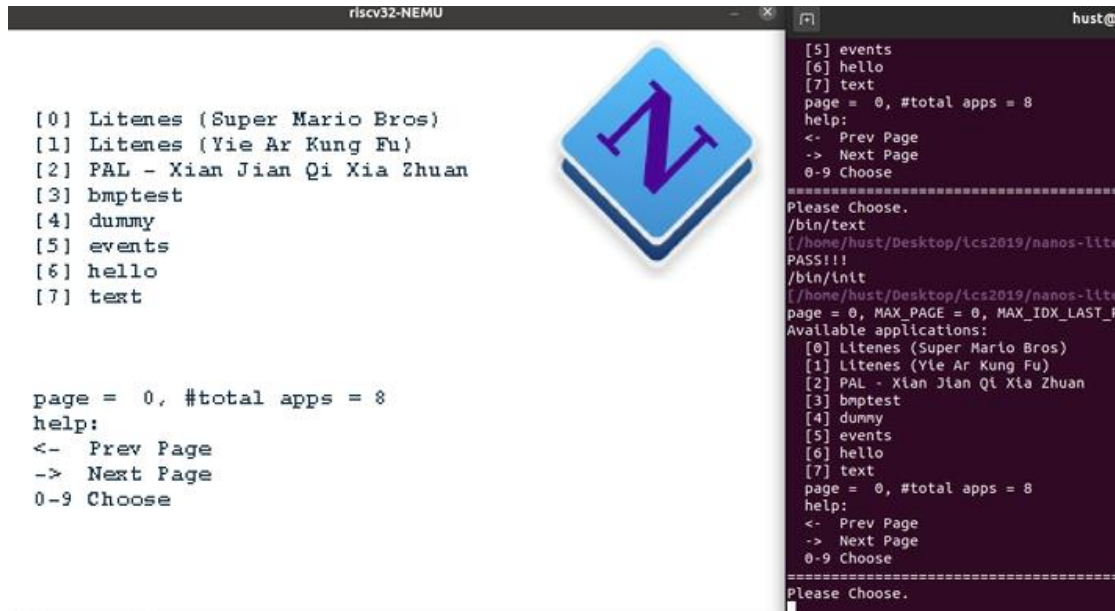
根据文档指示，下载仙剑奇侠传相关数据文件到 `navy-apps/fsimg/share/games/pal/` 目录，更新 `ramdisk`，使用 `Nanos-lite` 加载运行，演示效果如图 4.6 所示。



图 4.5 仙剑奇侠传实机演示效果

4.5 展示批处理系统

根据文档指示，向 VFS 中添加/dex/tty 文件，让它向串口进行写入。此外还需新增 SYS_execve 系统调用。上述步骤完成后再 Nanos-lite 加载 bin/init 即可运行，演示效果如图 4.6 所示。



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

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

[5] events
[6] hello
[7] text
page = 0, #total apps = 8
help:
<- Prev Page
-> Next Page
0-9 Choose

Please Choose.
/bin/text
[/home/hust/Desktop/tcs2019/nanos-lite]
PASS!!!
/bin/init
[/home/hust/Desktop/tcs2019/nanos-lite]
page = 0, MAX_PAGE = 0, MAX_IDX_LAST_P
Available applications:
[0] Litenes (Super Mario Bros)
[1] Litenes (Yie Ar Kung Fu)
[2] PAL - Xian Jian Qi Xia Zhuan
[3] bmptest
[4] dummy
[5] events
[6] hello
[7] text
page = 0, #total apps = 8
help:
<- Prev Page
-> Next Page
0-9 Choose

Please Choose.
```

图 4.6 批处理系统展示实机演示效果

4.6 章节小结

终于！来到了激动人心的最后一章节！完成到此时我的心情真的十分激动，当见到仙剑奇侠传在 NEMU 上运行起来时，语言难以形容我当下的心情。（虽然尝试着玩一段但实在过于卡顿，留待日后细品国产经典单机游戏）。

回到本章节的实现上，由于大都是系统调用，实现难度比 PA2 好了很多，其中键盘和 VGA 的抽象，也在 PA2 中提前预热过，只需结合 FS 的实现即可。

最终批处理系统展示的部分也很好的利用上了用户程序加载的功能，使得 NEMU 到现在为止不必再每次为了运行不同程序而反复 make run。总体而言，这一章使得我对于操作系统的系统调用以及事件分发机制都有了更深入的理解。

5 综合课设实验总结

作为大四上学期的唯一一门课，也是毕设之前的最后一份报告，系统能力综合训练课程梳理了从大一就开始接触的不少知识，不论是最先接触的 C 语言，再到数据结构，操作系统，甚至是表达式求值时用到的词法分析（编译原理）。除此之外还有一系列相关工具的使用：Linux 操作系统、shell 命令、make、git、虚拟机、gdb、gcc...

在这一过程中不只是对过去知识的回顾，更有一些先前没有琢磨透彻的补充。大二大三的学习中久未使用 C，阅读了 NEMU 框架代码才知道宏指令还有这样的用法，在一步步地调试和试错中磨练了自己的编程技巧以及工程能力。

因为出国交流的缘故，任务开始初期的时间没办法花在课设上，直到前段时间这边的学期结束，国内的学期也临近尾声才感到任务繁重、截止日期迫近，随之而来的是心理压力陡升。也感谢 ICS 2019 PA 文档，充分的解释说明，和随处可见的鼓励和督促，让我撑过这一段时间。

经过这门课程的训练，我更加直观地理解了程序从代码到 01 再到最终呈现在用户面前的流程，总体而言我在本次综合课设中收获颇丰。

参考文献

- [1] DAVID A. PATTERSON(美). 计算机组成与设计硬件/软件接口(原书第4版). 北京: 机械工业出版社.
- [2] David Money Harris(美). 数字设计和计算机体系结构(第二版). 机械工业出版社
- [3] 秦磊华, 吴非, 莫正坤. 计算机组成原理. 北京: 清华大学出版社, 2011 年.
- [4] 谭志虎, 秦磊华, 胡迪青. 计算机组成原理实践教程. 北京: 清华大学出版社, 2018.
- [5] 袁春风编著. 计算机组成与系统结构. 北京: 清华大学出版社, 2011 年.
- [6] 张晨曦, 王志英. 计算机系统结构. 高等教育出版社, 2008 年.