

华中科技大学

2021

系统能力综合训练 课程设计报告

题 目: riscv32 模拟器设计

专 业: 计算机科学与技术

班 级: CS1703

学 号: U201714670

姓 名: 范唯

电 话: 13451158896

邮 件: 2601426263@qq.com

完成日期: 2021-01-15



目 录

1	课程设计概述.....	1
1.1	课设目的	1
1.2	课设任务	1
1.3	实验环境	2
2	实验过程.....	3
2.1	PA1	3
2.1.1	总体设计	3
2.1.2	详细设计	3
2.1.3	运行结果	4
2.1.4	问题解答	6
2.2	PA2	8
2.2.1	总体设计	8
2.2.2	详细设计	8
2.2.3	运行结果	10
2.2.4	问题解答	12
2.3	PA3	14
2.3.1	总体设计	14
2.3.2	详细设计	14
2.3.3	运行结果	15
2.3.4	问题解答	18
3	设计总结与心得.....	20

3.1 课设总结	20
3.2 课设心得	20
参考文献	22

1 课程设计概述

1.1 课设目的

在代码框架中实现一个简化的 riscv32 模拟器 (NJU EMUlator)。

- 可解释执行 riscv32 执行代码
- 支持输入输出设备
- 支持异常流处理
- 支持精简操作系统---支持文件系统
- 支持虚存管理
- 支持进程分时调度

最终在模拟器上运行“仙剑奇侠传”,让学生探究“程序在计算机上运行”的机理,掌握计算机软硬协同的机制,进一步加深对计算机分层系统栈的理解,梳理大学 3 年所学的全部理论知识,提升学生计算机系统能力。

1.2 课设任务

1) 世界诞生前夜---开发环境:

安装虚拟机或者 docker, 熟悉相关工具和平台, 安装 sourceinsight 阅读代码框架。

2) 开天辟地---图灵机

PA1.1 简易调试器

PA1.2 表达式求值

PA1.3 监视点与断点

3) 简单复杂计算机--冯诺依曼计算机

PA2.1 运行第一个 C 程序

PA2.2 丰富指令集, 测试所有程序

PA2.3 实现 I/O 指令, 测试打字游戏

4) 穿越时空之旅: 异常控制流

PA3.1 实现系统调用

PA3.2 实现文件系统

PA3.3 运行仙剑奇侠传

5) 虚实交错的魔法：分时多任务

PA4.1 实现分页机制

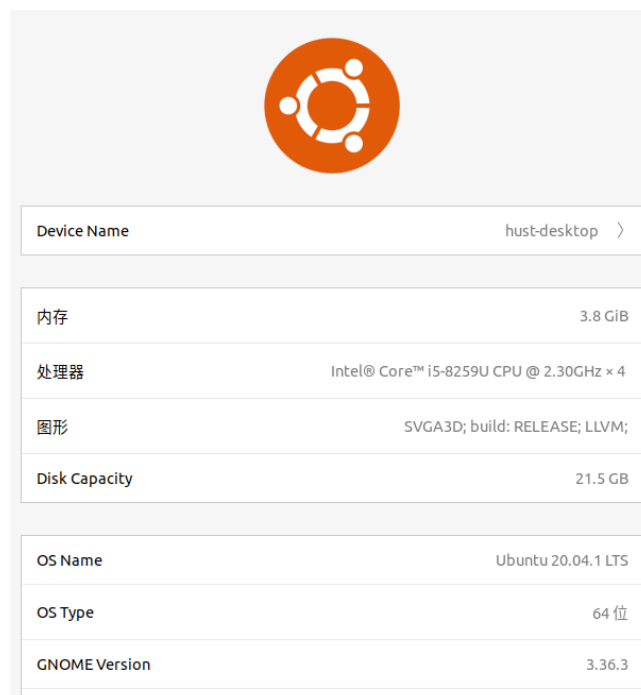
PA4.2 实现进程上下文切换

PA4.3 时钟中断驱动的上下文切换

1.3 实验环境

平台：macOS + GNU/Linux + gcc + C

其他工具：GDB, vim, Git, VScode



2 实验过程

2.1 PA1

2.1.1 总体设计

PA1 最终要求实现一个简易调试器，相当于一个简化版的 `gdb`，要求能实现单步执行、打印寄存器状态、扫描内存、表达式求值、监视点等功能。

PA1.1 需要实现实现单步执行，打印寄存器状态，扫描内存，只需要在 `ui.c` 中声明并实现相关的函数就行。

PA1.2 需要实现表达式求值功能，这是整个 PA1 的核心，表达式求值的实现最终需要在 `expr.c` 文件中实现，在这期间，需要完成补充正则表达式、括号匹配函数、递归求值 `eval` 函数，并最终整合到一起实现函数 `expr`。

PA1.3 需要实现监视点的功能，需要补充 `watchpoint` 结构，完成监视点的相关操作函数，然后在 `ui.c` 中完成对监视点相关命令函数的编写，最后需要实现监视点状态改变的暂停逻辑。

2.1.2 详细设计

PA1.1 首先需要实现单步执行，打印寄存器状态，扫描内存这三个函数。这三个函数在 `ui.c` 中定义为 `cmd_si`，`cmd_info` 和 `cmd_x`。`cmd_si` 调用 `cpu_exec` 执行指定的步数，`cmd_info` 调用 `isa_reg_display` 进行寄存器名称与值的输出，`isa_reg_display` 中使用函数 `reg_name` 和宏 `reg_l` 进行寄存器名称与值的打印，`cmd_x` 中对输入的参数进行分割，使用 `vaddr_read` 函数读出对应地址的值然后打印出来。

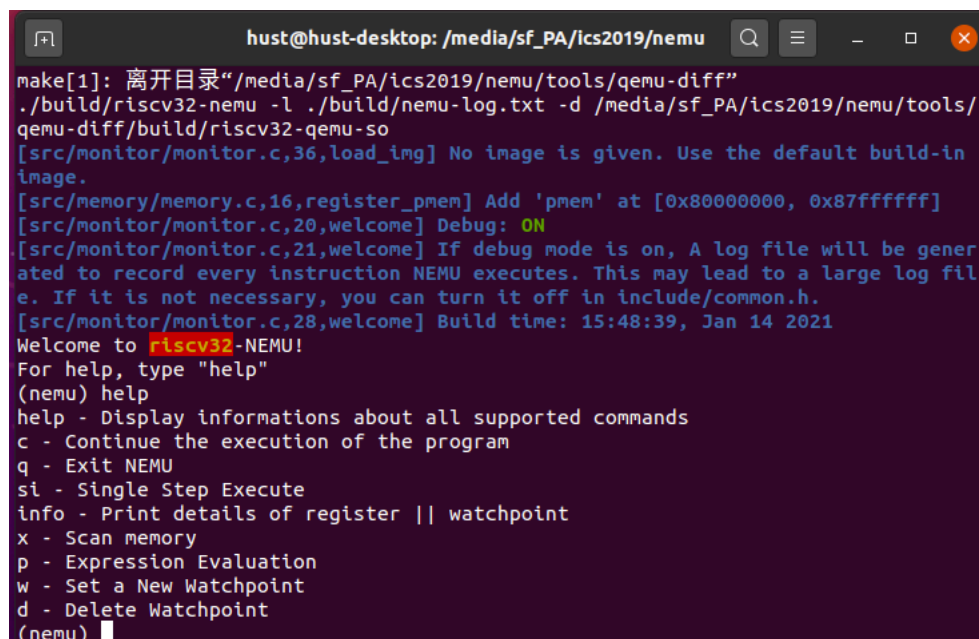
PA1.2 进行表达式求值，首先要在 `expr.c` 中补充规则和相应的正则表达式，例如十六进制数的正则表达式与 `token` 类型为 `{"0[Xx][0-9a-fA-F]+", TK_HEX}`。随后完成函数 `make_token`，用于为算术表达式中的各种 `token` 类型添加规则，并在成功识别出 `token` 后，将 `token` 的信息依次记录到 `tokens` 数组中。接着完成括号匹配函数 `check_parentheses`，完成运算符优先级函数 `op_priority`，根据 C 语言中的优先级来实现对应的操作符的优先级。在这些辅助函数都实现后，就可以进行递归求值函数 `eval` 的编写，`eval` 对传入的参数 `p`、`q` 进行处理，如果 `p>q` 则直接返回 0，`success` 置为 `false`；如果 `p==q` 则判断该 `token` 的类型是不是整数、十六进制整数和寄存器，如果是则返回其值，如果不是则置 `success` 为 `false`，返回 0；随后进行括号匹配检查，如果匹配则返回 `eval(p+1,q-1)`，否则则进入寻找主操作符行为；如果能找到主操作符，则根据主操作符返回相应的值，如果找不到则置

success 为 false，返回 0。完成 eval 函数后，将 eval 函数和 make_token 函数整合到函数 expr 中，即可完成整个表达式求值框架的编写。

PA1.3 要求实现监视点功能，首先要完善 watchpoint 结构，补充相应的变量，因为要监测表达式，所以需要有一个 char 数组存放表达式，同时因为要监测值的变化，所以需要有一个变量用于存储旧值。随后在 watchpoint.c 中实现函数 new_wp, free_wp, print_wp。new_wp 用于新建一个监视点，把链表 free_ 中的节点放入链表 head 中；free_wp 用于释放监视点，根据编号在 head 链表中找到相应的节点，放入链表 free_，这两个函数的操作都是通过链表的插入与删除来实现的，且为了方便快捷在插入链表时都选择头插法。print_wp 函数用于打印监视点信息，不再赘述。完成上述函数后，在 ui.c 中使用上述函数完成命令 cmd_x 与 cmd_d，完善函数 cmd_info，同时要在 cpu.exec 中实现当监视点状态改变使程序暂停执行的逻辑，该逻辑较为简单，程序每执行一步就对所有监视点进行表达式求值，并与旧值进行比较，如果两值不同，则置 nemu 的状态为 NEMU_STOP。

2.1.3 运行结果

进入简易调试器后，输入 help 显示各个指令对应的信息，如图 2.1 所示。



```
hust@hust-desktop: /media/sf_PA/ics2019/nemu
make[1]: 离开目录"/media/sf_PA/ics2019/nemu/tools/qemu-diff"
./build/riscv32-nemu -l ./build/nemu-log.txt -d /media/sf_PA/ics2019/nemu/tools/qemu-diff/build/riscv32-qemu-so
[src/monitor/monitor.c,36,load_img] No image is given. Use the default build-in image.
[src/memory/memory.c,16,register_pmem] Add 'pmem' at [0x80000000, 0x87ffffff]
[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: 15:48:39, Jan 14 2021
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 - Single Step Execute
info - Print details of register || watchpoint
x - Scan memory
p - Expression Evaluation
w - Set a New Watchpoint
d - Delete Watchpoint
(nemu)
```

图 2.1 PA1 运行结果 1

设置四个监视点后，使用 info 打印监视点信息，如图 2.2 所示。

```
hust@hust-desktop: /media/sf_PA/lcs2019/nemu
help - Display informations about all supported commands
c - Continue the execution of the program
q - Exit NEMU
si - Single Step Execute
info - Print details of register || watchpoint
x - Scan memory
p - Expression Evaluation
w - Set a New Watchpoint
d - Delete Watchpoint
(nemu) w $t0
Set Watchpoint Succeed
(nemu) w $t1
Set Watchpoint Succeed
(nemu) w $t2
Set Watchpoint Succeed
(nemu) w $t3
Set Watchpoint Succeed
(nemu) info w
NO      EXPR      VALUE
3       $t3       0
2       $t2       0
1       $t1       0
0       $t0       0
(nemu)
```

图 2.2 PA1 运行结果 2

删除 3 号监视点后打印监视点信息，如图 2.3 所示。

```
(nemu) d 3
Delete No.3 Watchpoint~
(nemu) info w
NO      EXPR      VALUE
2       $t2       0
1       $t1       0
0       $t0       0
(nemu)
```

图 2.3 PA1 运行结果 3

使用命令 si 执行两步，由于寄存器 t0 的值发生改变，程序暂停，使用 info 命令显示监视点的值，发现 t0 寄存器的值变为一个负数，实际上是十六进制 0x80000000，继续使用 si 单步执行，程序最终会 HIT GOOD TRAP，运行结果如图 2.4 所示。

```
hust@hust-desktop: /media/sf_PA/lcs2019/nemu
(nemu) d 3
Delete No.3 Watchpoint~
(nemu) info w
NO      EXPR      VALUE
2       $t2       0
1       $t1       0
0       $t0       0
(nemu) si 2
80100000: b7 02 00 80          lui  0x80000,t0
watchpoint:Status Changed
(nemu) info w
NO      EXPR      VALUE
2       $t2       0
1       $t1       0
0       $t0       -2147483648
(nemu) si 2
80100004: 23 a0 02 00          sw   0(t0),$0
80100008: 03 a5 02 00          lw   0(t0),a0
(nemu) si 2
8010000c: 6b 00 00 00          nemu trap
nemu: HIT GOOD TRAP at pc = 0x8010000c
[src/monitor/cpu-exec.c,32,monitor_statistic] total guest instructions = 4
```

图 2.4 PA1 运行结果 4

使用 `p` 命令进行表达式求值，对于正确的表达式会求出其值，对于错误的表达式会给出相应的提示，如图 2.5 所示。

```
(nemu) p (1+2)*(4/3)
3
(nemu) p --1
1
(nemu) p (3/3)+)(123*4
wrong expression
```

图 2.5 PA1 运行结果 5

2.1.4 问题解答

□ 必答题

你需要在实验报告中回答下列问题：

- **送分题** 我选择的ISA是 `x86`。
- **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：
 - 假设你需要编译500次NEMU才能完成PA。
 - 假设这500次编译当中，有90%的次数是用于调试。
 - 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。
 - 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？

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？

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。
- **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：
 - x86
 - EFLAGS寄存器中的CF位是什么意思？
 - ModR/M字节是什么？
 - mov指令的具体格式是怎么样的？
 - mips32
 - mips32有哪一种指令格式？
 - CP0寄存器是什么？
 - 若除法指令的除数为0，结果会怎样？
 - riscv32
 - riscv32有哪一种指令格式？
 - LUI指令的行为是什么？
 - mstatus寄存器的结构是怎么样的？
- **shell命令** 完成PA1的内容之后，`nemu/` 目录下的所有.c和.h文件总共有多少行代码？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？
- **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`？

ISA: riscv32。

调试需要花费的时间是 $500 \times 90\% \times 30 \times 20 = 270000s = 4500min$

简易调试器可以节约的时间为 $500 \times 90\% \times 20 \times 20 = 180000s = 3000min$

riscv32 有 R、I、S、B、U、J 6 种指令格式。

LUI 指令是将 20 位常量加载到寄存器的高 20 位。

mstatus 寄存器的结构如图 2.6 所示。

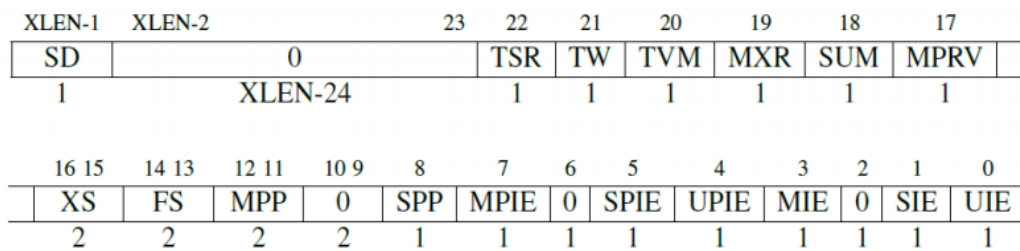


图 2.6 mstatus 寄存器结构

使用命令 `find . -name "[.h|.c]" |xargs cat|wc -l` 得出来的行数是 5877。

使用命令 `find . -name "[.h|.c]" |xargs cat|grep -v ^$|wc -l` 过滤空行后，得出的行数是 4822。

-Wall 的作用是打开 gcc 所有警告。

-Werror 的作用是要求 gcc 将所有警告当成错误处理。

2.2 PA2

2.2.1 总体设计

PA2 要求实现足够多的指令，能够运行各种测试，同时提供对输入输出设备的支持。

PA2.1 要求实现部分指令能够运行 `dummy.c`，观察反汇编代码，找出未实现的指令，编写相应的译码和执行函数即可。

PA2.2 要求实现更多的指令，在 NEMU 中运行所有 `cputest`，这里需要找出观察所有的测试文件的反汇编代码，找出未实现的指令，编写相应的译码和执行函数，为了测试的方便可以先完成 `diff-test`。

PA2.3 要求能够运行打字小游戏以及其他的输入输出测试，这个需要在相关的设备文件下编写设备功能，同时完成相关输入输出函数。

2.2.2 详细设计

PA2.1 首先直接使用 `make` 命令跑 `dummy` 程序，会显示 `abort`，随后会在 `build` 文件夹中生成对应的反汇编文件，将其中未实现的指令找出，在手册中查询，找出所有的伪指令，最终可确认 `dummy` 需要实现的新指令有 `auipc addi jal jalr`，随后在 `all-instr.h` 中定义相关指令的执行函数，在 `exec.c` 的 `opcode_table` 中添加新指令，在 `decode.c` 中实现相应的译码辅助函数，在 `rtl.h` 中修改完善 `rtl` 指令，在 `compute.c` 和 `control.c` 等文件中使用 `rtl` 指令实现正确的执行辅助函数，完成上述步骤后，重新编译运行即可。

PA2.2 过程类似于 PA2.1，只不过要实现更多的指令。更多的指令意味着更容易出错，因此首先完成 `diff-test` 是一个不错的选择，`diff-test` 将自己实现的 `nemu` 与 `qemu` 在执行指令的过程中进行对比，每执行一步就比较两者 32 个寄存器中的值是否一样，如果不一样则报错，根据 `abort` 的 PC 值可以在反汇编代码中找出是哪一行出错，从而分析找出解决方法。完成 `diff-test` 后，可以开始指令的实现，为了方便测试找出错误，我选择将测试文件按大小从小到大排列，依次编译运行，找出未实现的指令进行实现。这里以 `sum.c` 为例，阐述实现指令的整个流程：在完成了 `dummy`，观察 `sum` 的反汇编代码，发现还需要实现的指令有 `beq`、`bne`、`add` 和 `sltiu`，其中 `beq` 和 `bne` 指令是 B 型指令，`add` 是 R 型指令，在 `dummy` 中没涉及到这两个类型的指令，所以要进行译码辅助函数的编写，查阅手册根据相关指令的结构编写好对应的译码函数，随后进行执行辅助函数的编写，`beq` 指令和 `bne` 指令因为设计指令的跳转，执行辅助函数要在 `control.c` 中编写，`add` 和 `sltiu` 是单纯的计算指令，所以在 `compute.c` 文件中编写，随后查阅手册，根据指令的相关行为编写对应的执行辅助函数。编写完两个函数后，如果没有声明，需要在 `decode.h` 中声明译码辅助函数，需要在 `all-instr.h` 中声明执行辅助函数。随

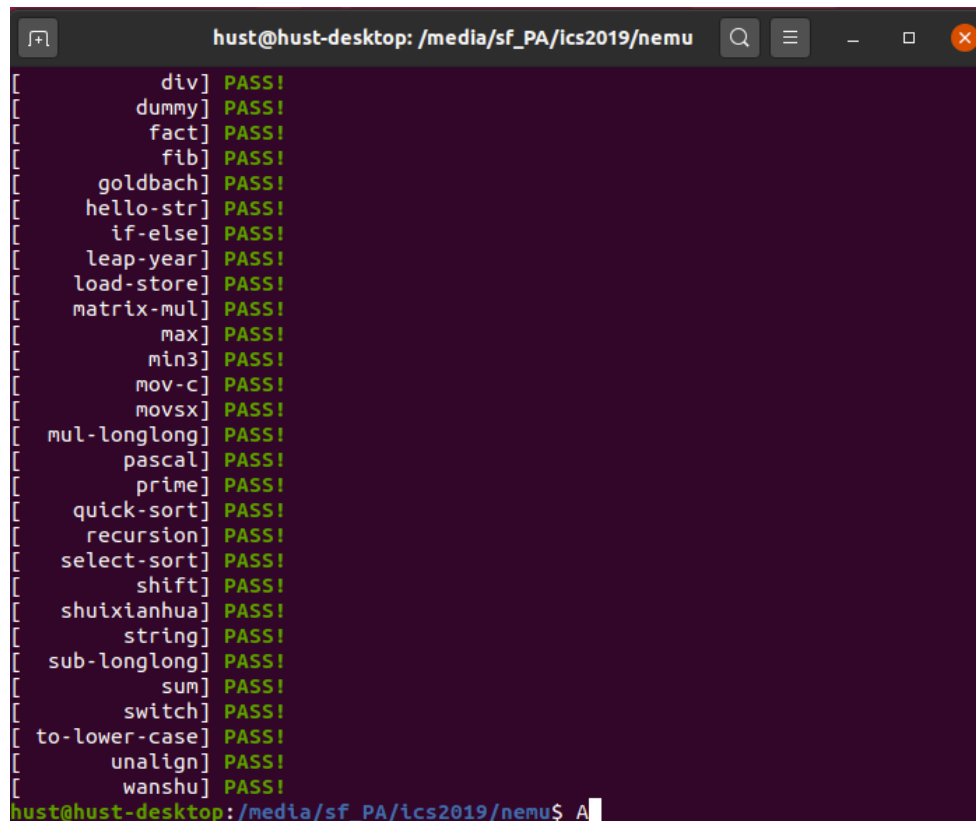
后在 `exec.c` 的 `opcode_table` 中指明指令对应的译码和执行辅助函数,做完这些后,使用 `make` 命令执行对应的测试文件,如果出错就检查对应的译码和执行函数,通过就进入下一个指令的编写,重复上述过程直到所有测试文件通过。

此外,在测试文件 `hello-str` 和 `string` 中要使用库函数 `sprintf`、`strcmp`、`strcat`、`strcpy` 和 `memset`,这些函数需要我们在 `nexus-am/libs/klib/src/stdio.c` 以及 `string.c` 中编写,`string.c` 中需要编写的是平常经常使用到的字符串相关的函数,较为简单例如 `strcmp`、`strlen`、`strcat` 等,在 `stdio.c` 中需要编写 `sprintf`、`printf` 等函数,这两个函数都通过中间函数 `vsprintf` 实现。`vsprintf` 需要对传入的字符串 `fmt` 以及可变参数列表 `va_list` 进行处理,当在 `fmt` 字符串中读取到 `%d`、`%x`、`%s` 的子串时,调用 `va_arg` 函数将 `va_list` 中对应类型的参数,然后转换成字符串传入输出字符串 `out` 中,这样就完成了 `vsprintf` 函数的编写。在 `sprintf` 和 `printf` 函数中简单的调用 `vsprintf` 函数即可完成各自的功能。至此 PA2.2 的内容全部完成,使用一键回归测试可以测试所有的程序。

PA2.3 要完成串口、时钟、键盘、VGA 四个输入输出设备程序的编写。串口在 `trm.c` 中已经实现,我们还需要编写 `printf` 函数以便程序运行,由于 `printf` 函数已在 PA2.2 中写好,不再赘述。时钟的功能需要在 `nemu-timer.c` 中完善,在启动 `init` 时通过 `RTC_ADDR` 获取启动时间,运行时钟功能时通过地址 `RTC_ADDR` 获取当前时间,将 `uptime->hi` 设为 0, `uptime->lo` 设为当前时间与启动时间的差值,即可完成时钟功能。键盘的功能需要在 `nemu-input.c` 中完善,通过地址 `KBD_ADDR` 获取键盘按键信息,放入 `kbd->keycode` 中,将按键信息与 `KEYDOWN_MASK` 相与,如果值为 1 就说明是按下,值为 0 说明是松开。最后需要实现的是 VGA 设备,在 `nemu-video.c` 中完善相关的功能,VGA 设备需要将 `pixels` 中的像素信息写入 VGA 对应的地址空间中,就能正确显示图像。

2.2.3 运行结果

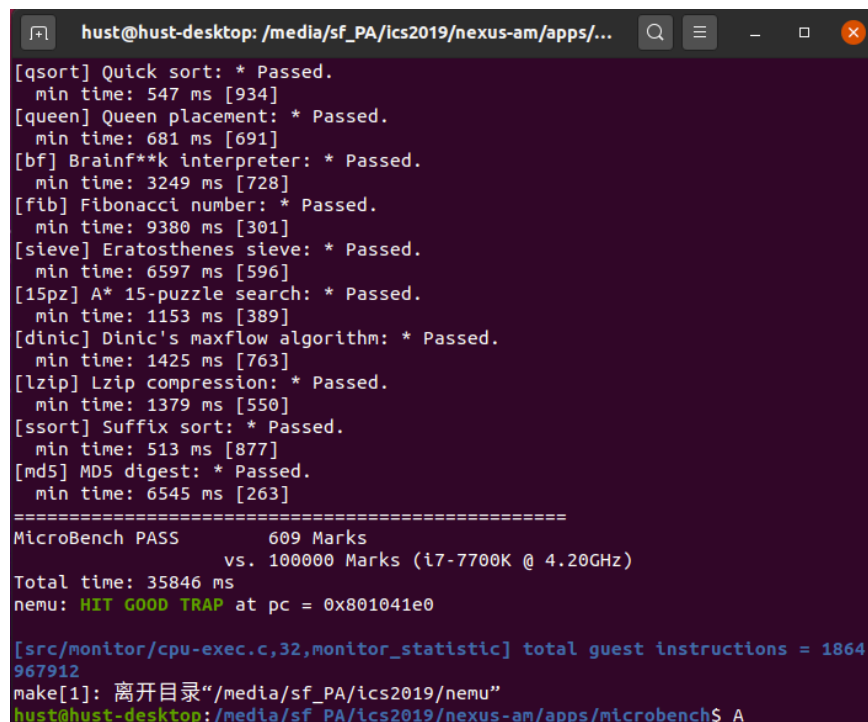
在 nemu 目录下，使用一键回归测试，运行结果如图 2.7 所示。



```
hust@hust-desktop: /media/sf_PA/ics2019/nemu
[ 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: /media/sf_PA/ics2019/nemu$ A
```

图 2.7 PA2 运行结果 1

microbench 测试的运行结果如图 2.8 所示。总分为 609 分



```
hust@hust-desktop: /media/sf_PA/ics2019/nexus-am/apps/...
[qsort] Quick sort: * Passed.
min time: 547 ms [934]
[queen] Queen placement: * Passed.
min time: 681 ms [691]
[bf] Brainf**k interpreter: * Passed.
min time: 3249 ms [728]
[fib] Fibonacci number: * Passed.
min time: 9380 ms [301]
[sieve] Eratosthenes sieve: * Passed.
min time: 6597 ms [596]
[15pz] A* 15-puzzle search: * Passed.
min time: 1153 ms [389]
[binic] Dinic's maxflow algorithm: * Passed.
min time: 1425 ms [763]
[lzip] Lzip compression: * Passed.
min time: 1379 ms [550]
[ssort] Suffix sort: * Passed.
min time: 513 ms [877]
[md5] MD5 digest: * Passed.
min time: 6545 ms [263]
=====
MicroBench PASS      609 Marks
                        vs. 100000 Marks (i7-7700K @ 4.20GHz)
Total time: 35846 ms
nemu: HIT GOOD TRAP at pc = 0x801041e0

[src/monitor/cpu-exec.c,32,monitor_statistic] total guest instructions = 1864
967912
make[1]: 离开目录"/media/sf_PA/ics2019/nemu"
hust@hust-desktop: /media/sf_PA/ics2019/nexus-am/apps/microbench$ A
```

图 2.8 PA2 运行结果 2

slider 测试的运行结果如图 2.9 所示。

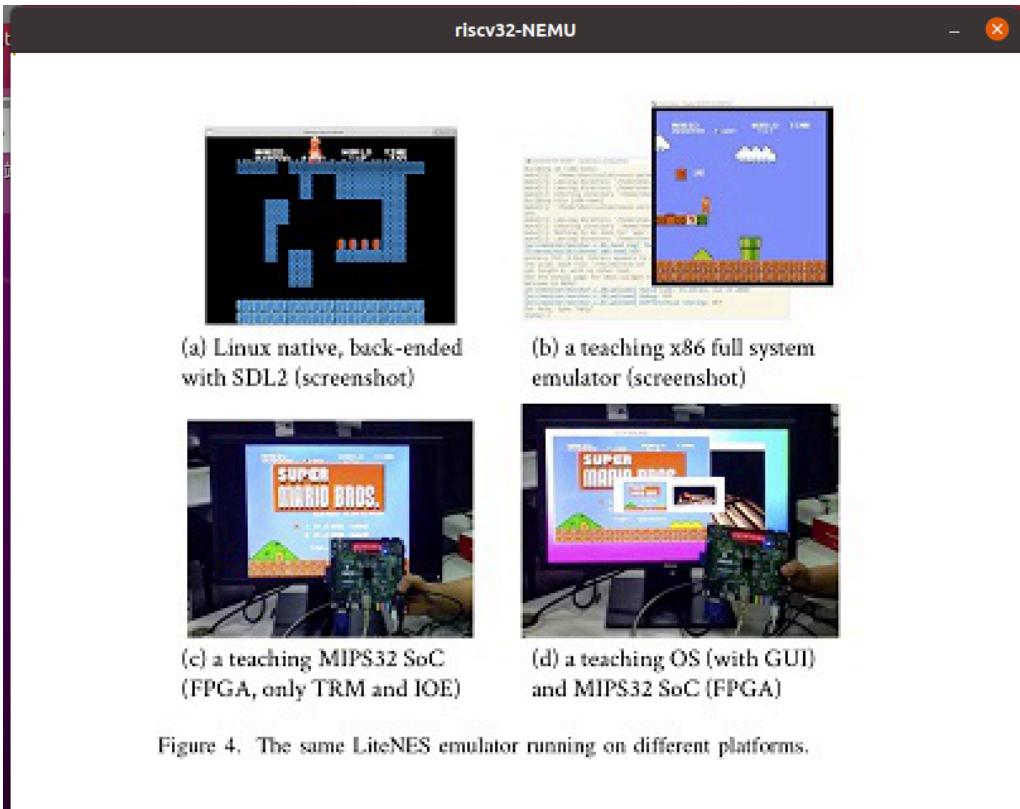


图 2.9 PA2 运行结果 3

打字小游戏 typing 的测试结果如图 2.10 所示。



图 2.10 PA2 运行结果 4

Litenes 测试结果如图 2.11 所示。



图 2.11 PA2 运行结果 5

2.2.4 问题解答

必答题

你需要在实验报告中用自己的语言, 尽可能详细地回答下列问题.

- **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)
- **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?
- **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?
 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.
 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过这样的问题? (回答完本题后可以删除添加的代码.)
- **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:
 - `Makefile` 中使用了变量, 包含文件等特性
 - `Makefile` 运用并重写了一些implicit rules
 - 在 `man make` 中搜索 `-n` 选项, 也许会对你有帮助
 - RTFM

1) 一条指令在 nemu 中执行的过程:

首先根据 PC 值通过 `instr_fetch` 函数取指令,从选出的指令中选取其 `opcode`,在 `opcode_table` 中进行索引,找到该指令对应的译码辅助函数和执行辅助函数,随后通过译码辅助函数进行译码,将译码得到的相关信息保存在 `decinfo` 中,接着通过执行辅助函数执行指令,执行辅助函数通过 `rtl` 指令对译码得到的信息进行相关操作,计算、读取、保存等等,最后通过 `update_pc` 函数更新 PC 值。

2) 以 `rtl_li` 函数为例单独去掉 `static` 和单独去掉 `inline` 进行重新编译,都不会报错,但将两者同时去掉时,就会报错。原因是都去掉时,在另一个文件中也有对 `rtl_li` 的定义,会出现重复定义的错误,而具有 `static` 关键字时,函数会被限制在本文件内,不会出现重复定义的错误,具有 `inline` 关键字时,函数在预编译时就会展开,不会出现重复定义的错误,但如果将两者同时去掉,就会出现重复定义的错误。

3) 添加后,使用 `grep` 命令查看,共有 81 个 `dummy` 实体;继续添加后,重新编译运行,使用 `grep` 命令,共有 82 个 `dummy` 实体;修改代码后,重新编译报错,原因是两个都初始化后,会产生两个强符号,导致错误。

4) 敲入 `make` 后,会将 `makefile` 文件中第一个目标文件作为最终的目标文件,如果文件不存在,或是文件所依赖的后面的.o 文件的修改时间比这个文件晚,就会重新编译;如果目标文件依赖的.o 文件也不存在,就根据这个.o 文件的生成规则生成,然后生成上一层.o 文件,中间某一步出错就会直接报错。

2.3 PA3

2.3.1 总体设计

PA3 的主要任务是实现系统调用和文件系统，使得 nemu 最终能够运行仙剑奇侠传。

PA3.1 要求实现自陷操作 `_yield` 及其过程，要实现自陷操作，首先要在 `cpu` 结构中添加控制状态寄存器，然后实现自陷需要的指令，通过异常号识别出自陷异常，完成自陷事件。

PA3.2 需要实现用户程序的加载和系统调用，支撑 TRM 程序的运行，需要通过实现 `loader` 函数，增加系统调用，完善堆区管理函数。

PA3.3 需要实现文件系统，完成 `fs_open`, `fs_read` 等函数，完成设备函数的编写，使用 `fs` 函数修改 `loader`，最终可运行仙剑奇侠传。

2.3.2 详细设计

PA3.1 要实现自陷操作，要完成自陷操作，首先要实现自陷指令，需要实现的指令有 `csrrs`、`csrrw`、`ecall` 和 `sret`，通过 `ecall` 指令进入自陷操作，`csrrs` 和 `csrrw` 指令对控制状态寄存器进行修改，`sret` 指令用于自陷后返回，然后需要在 `intr.c` 文件中进行 `raise_intr` 函数的编写，`raise_intr` 函数用于模拟相应过程：将当前 PC 值保存到 `sepc` 寄存器，在 `scause` 寄存器中设置异常号，从 `stvec` 寄存器中取出异常入口地址，跳转到异常入口地址；触发自陷操作后，需要保存上下文，根据 `trap.S` 汇编代码的压栈的顺序重构 `_Context` 成员体结构，接着要实现正确的事件分发，需要在 `__am_irq_handle` 函数中通过异常号识别出自陷异常，在 `do_event` 函数中识别出自陷事件 `_EVENT_YIELD`，最后通过 `sret` 指令返回并恢复上下文。

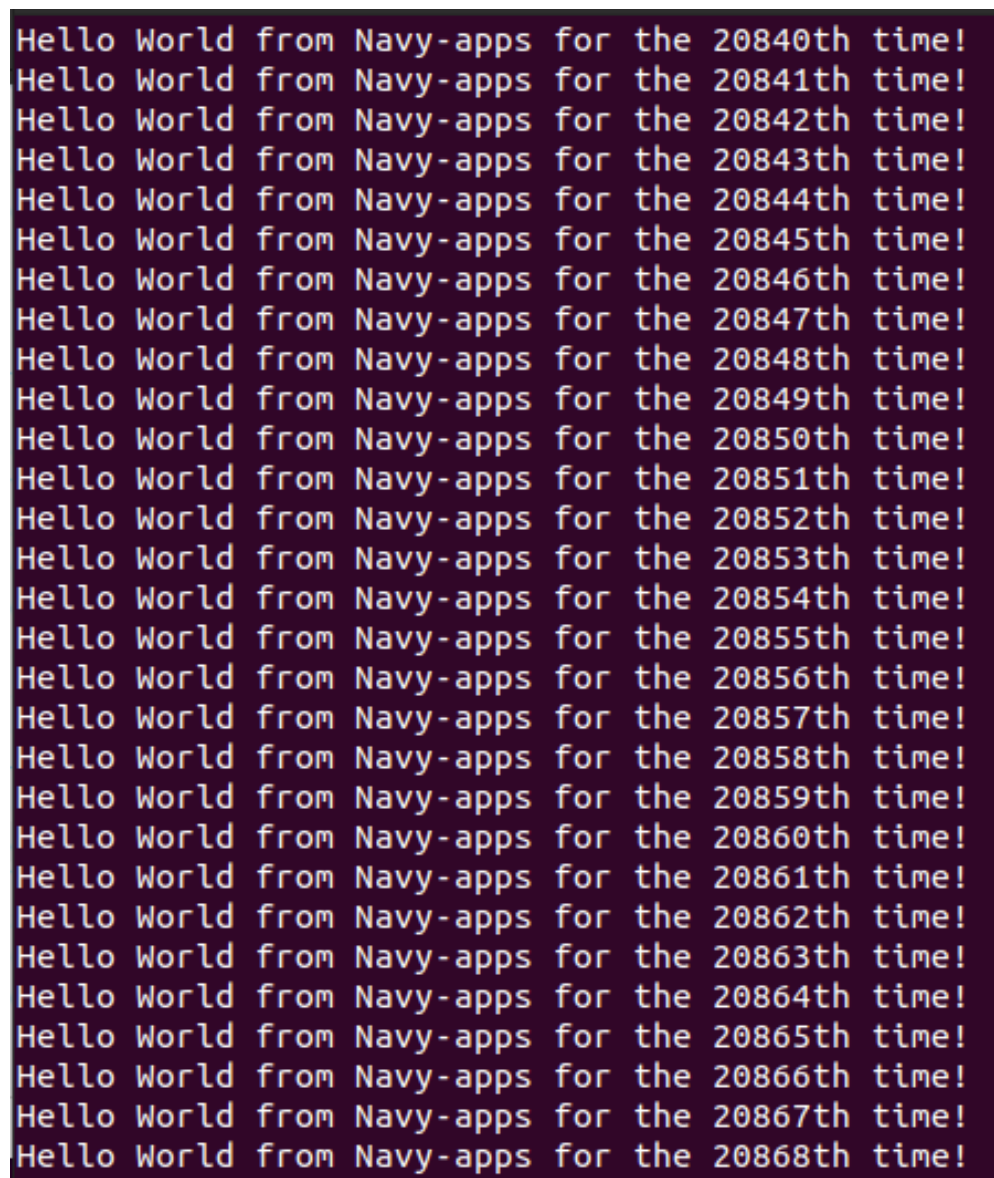
PA3.2 需要实现用户程序的加载和系统调用，支撑 TRM 程序的运行，为了加载用户程序首先要实现 `loader` 函数，因为目前还没有实现文件系统，所以直接在 `loader` 函数中通过 `ramdisk_read` 函数把可执行文件中的代码和数据放置在正确的内存位置，然后跳转到程序入口，接着需要完成系统调用的实现，和 3.1 中识别出自陷事件类似，让 `nemu` 能够识别出系统调用，然后在 `do_event` 中添加 `do_syscall` 的调用，根据 `nanos.c` 中的 `ARGS_ARRAY` 在 `riscv32-nemu` 中实现正确的 GPR 宏，随后添加 `SYS_yield`、`SYS_read`、`SYS_write` 和 `SYS_brk` 系统调用，完成函数 `_sbrk` 实现堆区管理。

PA3.3 需要实现文件系统，添加设备的支持，最终能够运行仙剑奇侠传。首先需要实现函数 `fs_open`、`fs_read` 和 `fs_close`，因为 `ramdisk` 中的文件数量增加之后，就不适合直接在 `loader` 函数中直接使用 `ramdisk_read`，在完成了这几个 `fs` 函数后，替换掉 `loader` 函数中的 `ramdisk_read` 函数，修改其逻辑这样就可以在

loader 中使用文件名来指定加载的程序了，随后完成 fs_write 和 fs_lseek 函数，使其能够进行输出，完成这些函数后，要补充相关的系统调用；要实现虚拟文件系统 VFS，把 IOE 抽象成文件，首先需要在 VFS 中补充多种特殊文件的支持，接着实现函数 serial_write,完成串口的写入，然后完成实现 events_read 函数，支持读操作，最后需要完成 init_fs、fb_write、fbsync_write、init_device、dispinfo_read 函数，实现对 VGA 设备的支持。要注意的是因为在 Finfo 结构中添加了读函数指针和写函数指针，所以要修改修改 fs_write 和 fs_read 的逻辑，完成这些以后，如果没有错误，就能够运行仙剑奇侠传了。

2.3.3 运行结果

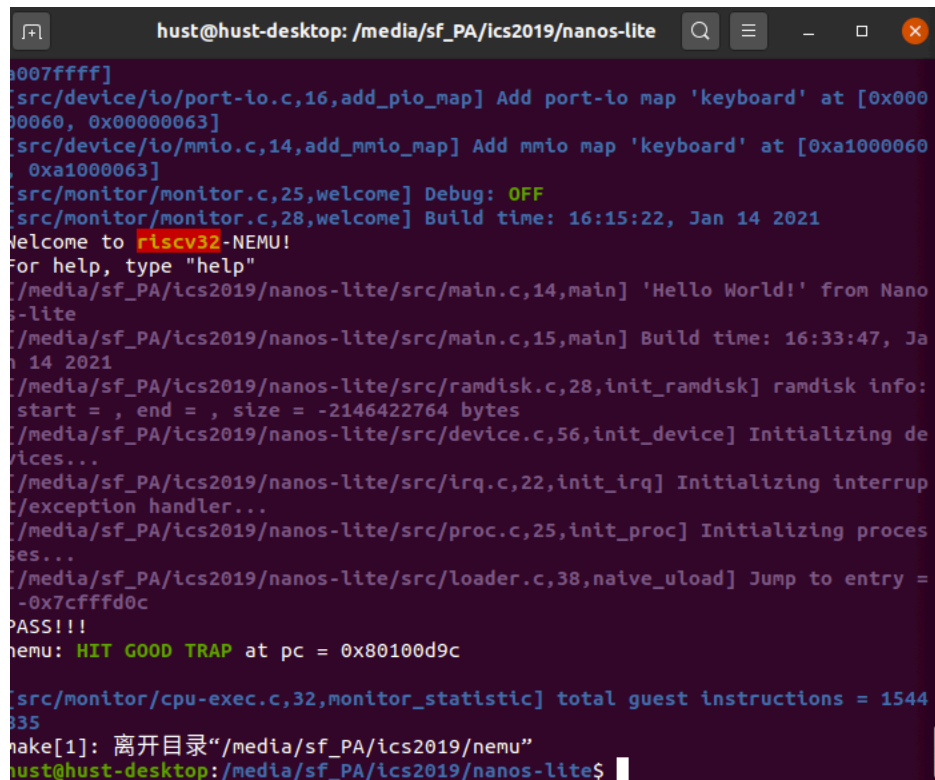
运行 hello 测试的运行结果如图 2.12 所示。



```
Hello World from Navy-apps for the 20840th time!  
Hello World from Navy-apps for the 20841th time!  
Hello World from Navy-apps for the 20842th time!  
Hello World from Navy-apps for the 20843th time!  
Hello World from Navy-apps for the 20844th time!  
Hello World from Navy-apps for the 20845th time!  
Hello World from Navy-apps for the 20846th time!  
Hello World from Navy-apps for the 20847th time!  
Hello World from Navy-apps for the 20848th time!  
Hello World from Navy-apps for the 20849th time!  
Hello World from Navy-apps for the 20850th time!  
Hello World from Navy-apps for the 20851th time!  
Hello World from Navy-apps for the 20852th time!  
Hello World from Navy-apps for the 20853th time!  
Hello World from Navy-apps for the 20854th time!  
Hello World from Navy-apps for the 20855th time!  
Hello World from Navy-apps for the 20856th time!  
Hello World from Navy-apps for the 20857th time!  
Hello World from Navy-apps for the 20858th time!  
Hello World from Navy-apps for the 20859th time!  
Hello World from Navy-apps for the 20860th time!  
Hello World from Navy-apps for the 20861th time!  
Hello World from Navy-apps for the 20862th time!  
Hello World from Navy-apps for the 20863th time!  
Hello World from Navy-apps for the 20864th time!  
Hello World from Navy-apps for the 20865th time!  
Hello World from Navy-apps for the 20866th time!  
Hello World from Navy-apps for the 20867th time!  
Hello World from Navy-apps for the 20868th time!
```

图 2.12 PA3 运行结果 1

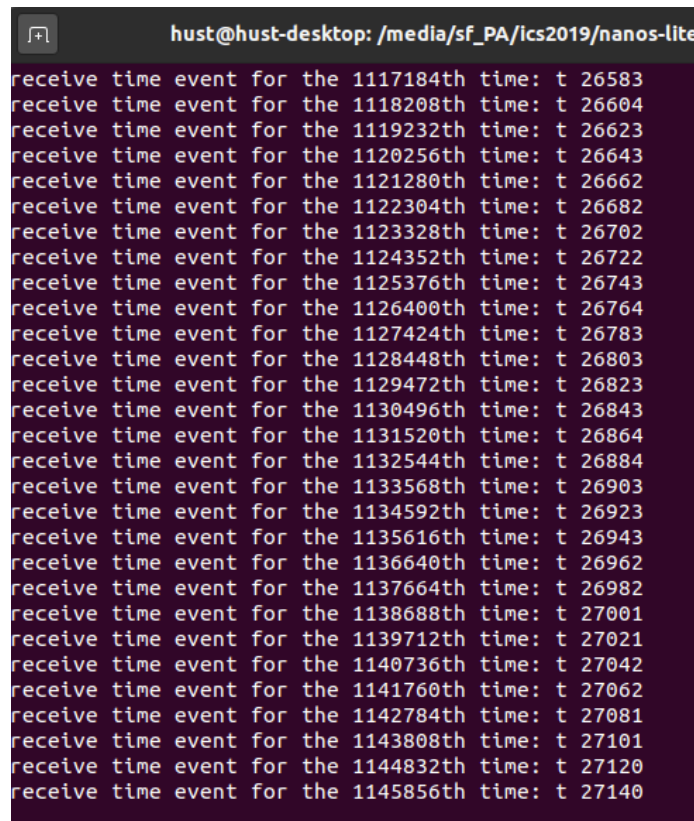
运行 text 测试的运行结果如图 2.13 所示。



```
hust@hust-desktop: /media/sf_PA/ics2019/nanos-lite
[0007ffff]
src/device/io/port-io.c,16,add_pio_map] Add port-io map 'keyboard' at [0x000
00060, 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
src/monitor/monitor.c,28,welcome] Build time: 16:15:22, Jan 14 2021
Welcome to riscv32-NEMU!
For help, type "help"
/media/sf_PA/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nano
-lite
/media/sf_PA/ics2019/nanos-lite/src/main.c,15,main] Build time: 16:33:47, Ja
n 14 2021
/media/sf_PA/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info:
start = , end = , size = -2146422764 bytes
/media/sf_PA/ics2019/nanos-lite/src/device.c,56,init_device] Initializing de
vices...
/media/sf_PA/ics2019/nanos-lite/src/irq.c,22,init_irq] Initializing interrup
t/exception handler...
/media/sf_PA/ics2019/nanos-lite/src/proc.c,25,init_proc] Initializing proces
ses...
/media/sf_PA/ics2019/nanos-lite/src/loader.c,38,naive_oload] Jump to entry =
-0x7cffffd0c
PASS!!!
nemu: HIT GOOD TRAP at pc = 0x80100d9c
src/monitor/cpu-exec.c,32,monitor_statistic] total guest instructions = 1544
335
make[1]: 离开目录"/media/sf_PA/ics2019/nemu"
hust@hust-desktop: /media/sf_PA/ics2019/nanos-lite$
```

图 2.13 PA3 运行结果 2

运行 events 测试的结果如图 2.14 所示。



```
hust@hust-desktop: /media/sf_PA/ics2019/nanos-lite
receive time event for the 1117184th time: t 26583
receive time event for the 1118208th time: t 26604
receive time event for the 1119232th time: t 26623
receive time event for the 1120256th time: t 26643
receive time event for the 1121280th time: t 26662
receive time event for the 1122304th time: t 26682
receive time event for the 1123328th time: t 26702
receive time event for the 1124352th time: t 26722
receive time event for the 1125376th time: t 26743
receive time event for the 1126400th time: t 26764
receive time event for the 1127424th time: t 26783
receive time event for the 1128448th time: t 26803
receive time event for the 1129472th time: t 26823
receive time event for the 1130496th time: t 26843
receive time event for the 1131520th time: t 26864
receive time event for the 1132544th time: t 26884
receive time event for the 1133568th time: t 26903
receive time event for the 1134592th time: t 26923
receive time event for the 1135616th time: t 26943
receive time event for the 1136640th time: t 26962
receive time event for the 1137664th time: t 26982
receive time event for the 1138688th time: t 27001
receive time event for the 1139712th time: t 27021
receive time event for the 1140736th time: t 27042
receive time event for the 1141760th time: t 27062
receive time event for the 1142784th time: t 27081
receive time event for the 1143808th time: t 27101
receive time event for the 1144832th time: t 27120
receive time event for the 1145856th time: t 27140
```

图 2.14 PA3 运行结果 3

运行 bptest 的运行结果如图 2.15 所示。



图 2.15 PA3 运行结果 4

仙剑奇侠传的运行结果如图 2.16 所示。





图 2.16 PA3 运行结果 5

2.3.4 问题解答

必答题 - 理解计算机系统

- 理解上下文结构体的前世今生 (见PA3.1阶段)
- 理解穿越时空的旅程 (见PA3.1阶段)
- hello程序是什么, 它从而何来, 要到哪里去 (见PA3.2阶段)
- 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?

1) `c` 指向的上下文结构 `_Context` 是在执行自陷操作时，通过 `trap.S` 的汇编程序运行进行赋值的。`riscv-nemu.h` 定义了相关的结构，`trap.S` 对上下文结构体进行赋值，讲义讲清了流程，实现的指令使自陷操作顺利执行。

2) `Nanos-lite` 调用中断，操纵 `AM` 发起自陷指令的汇编代码，随后保存上下文，转入 `CPU` 自陷指令的内存区域，执行完毕后，恢复上下文，返回运行时环境。

3) `hello.c` 被编译成 `ELF` 文件后，位于 `ramdisk` 中，通过 `naive_oload` 函数读入内存并放在正确的位置，交给操作系统进行调用执行，它通过 `SYS_write` 系统调用来输出字符，程序执行完毕后操作系统会回收其内存空间。

4) 操作系统通过库函数读出画面的像素信息，画面通过 `VGA` 输出，`VGA` 被抽象成设备文件，`fs_wrtie` 函数在一步步执行中调用了 `draw_rect` 函数，`draw_rect` 函数把像素信息写入到 `VGA` 对应的地址空间中，最后通过 `update_screen` 函数将画面显示在屏幕上。

3 设计总结与心得

3.1 课设总结

本次实验的整体设计思路还是非常清晰的，唯一美中不足的可能就是没能在更早一些的年级就遇上这个实验吧。**强烈建议把这个课设放到大三或者大二的时候**，这样会让学生对这个课程设计有更深的体会。

PA1 的整个核心思想就是实现一个编译器，最难的点可能就是实现四则运算，遇到了各种各样的玄学 bug。

PA2 就变得十分类似于组原实验了，对于 riscv32 的指令的不熟悉让我非常尴尬，也翻阅了大量资料去解决每一个 bug。

PA3 开始就变得十分精彩了，因为有游戏可以玩了，实现了自陷指令、系统调用、文件系统，这都是操作系统的经典知识了。而 PA3 的开发量相对比较小，仙剑的铁杆粉丝在成功运行出来时，非常激动。

3.2 课设心得

1.学会变通，之前我总是喜欢直接在虚拟机里面开 vscode 硬撸代码，但是后来发现了通过本机 ssh 和 vscode 的 ssh 插件，可以把虚拟机当成一个服务器，然后用本机 ssh 到虚拟机，再用 vscode 可视化出来，效率直接 up！

2. PA1 主要作用是熟悉工程的架构，有一个平缓的学习曲线，和后面试验关系不大，表达式求值中间有一个非常坑的地方，那就是 `expr.c` 中间定义了一个宏表示可以处理的最长的符号个数。

3.尽量早的完成 diff test, 完成 diff test 的文档在完成指令填写之后，这导致很多人手动 debug 完成了大部分的指令填写之后才发现有这一个神器。指令填写过程中间一旦遇到 bug,依赖于阅读汇编代码以及自己的 gdb 来 debug, 需要话费及其长的时间，而且经过优化的代码难以阅读，遇到较大的项目，对应代码简直和天书一样。尽量早的完成 diff test, 完成 diff test 的文档在完成指令填写之后，这导致很多人手动 debug 完成了大部分的指令填写之后才发现有这一个神器。指令填写过程中间一旦遇到 bug,依赖于阅读汇编代码以及自己的 gdb 来 debug, 需要话费及其长的时间，而且经过优化的代码难以阅读，遇到较大的项目，对应代码简直和天书一样。

4. 随着试验的进行，大家逐渐会模糊一件事情，当你想要调试的时候，printf 到底 glibc 提供的(也就是物理机上安装的操作系统上的运行时库提供的),还是

klib 提供的(就是哪一个嵌入式库提供的), 还是 Nemu 中间的 printf(就是 PA2 在 Nemu 中间实现的一系列的 io 函数)。当然其中使用 Log 函数是如何实现的。

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]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.
- [5]张晨曦，王志英. 计算机系统结构. 高等教育出版社，2008 年.