

华中科技大学

2022

系统能力培养 课程实验报告

题 目:	指令模拟器
专 业:	计算机科学与技术
班 级:	CS1902 班
学 号:	U201814511
姓 名:	余纪龙
电 话:	18612248474
邮 件:	313613839@qq.com
完成日期:	2022-2-19



目 录

1 课程实验概述	1
2 实验方案设计	错误!未定义书签。
3 实验结果与结果分析.....	错误!未定义书签。
参考文献	11

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 `wa` 内核的计算机系统。系统能力综合训练课程提出 `riscv32` 架构相应的教学版子集，学生通过实现一个简化过的 `riscv32` 模拟器 `nemu`，最终目标是在 `NEMU` 上能够运行“仙剑奇侠传一”，以此让学生能够充分探究“程序在计算机上运行”的基本原理。`NEMU` 收到 `QEMU` 的启发，去除了大量与实验内容总体差异较大的部分。本实验总共包括五个部分连贯的实验内容：

1. 图灵机与建议调试器
2. 冯诺依曼机
3. 批处理
4. 分时多任务
5. 程序优化

1.2 实验环境

- CPU 架构：x64
- 操作系统：GNU/Linux
- 编译器：GCC
- 编程语言：C 语言

2 PA1

2.1 实验任务

1. 实现单步执行，打印寄存器状态，扫描内
2. 实现算术表达式求值。
3. 实现监视点的设置和删除

2.2 实验步骤与结果

本实验的任务是通过构建一个完整的简单计算机系统，来深入理解程序如何在计算机上运行。为了提高机器在调试阶段的效率，我们首先完成一个简单的调试器，方便我们后续进行调试。其功能表如下表 2.1 所示。

命令	命令符	功能
帮助	Help	打印命令帮助信息
继续运行	C	Continue
退出	Q	退出 NEMU
单步执行	Si[n]	单步执行 n 条指令（默认 n=1）
打印程序状态	Info r/w	打印寄存器和监视点状态
表达式求值	P EXPR	求表达式的值
扫描内存	X N EXPR	十六进制形式输出 EXPR 作为起始内存地址，连续 N 个四字节内容
设置监视点	W EXPR	当 EXPR 发生变化时，暂停执行
删除监视点	D N	删除序号为 N 的监视点

起中，help、c、q 指令均已经给出。

首先单步执行功能。框架代码中模拟 CPU 执行方式的函数 `cpu_exc(uint64_t)`,

该函数对应期望 CPU 执行步数。因而，程序首先对单步执行命令后参数读取，若未读取到参数则调用 `cpu_exec(1)`，反之则以读到的参数进行调用。

单步执行功能进行测试得到的结果如图 2.1 所示。可以看出，单步执行在给定参数和默认参数都可以正常执行。

```

Welcome to riscv32-NEMU!
For help, type "help"
(nemu) si
80100000:  b7 02 00 80          lui  0x80000,t0
(nemu) si 3
80100004:  23 a0 02 00          sw   0(t0),$0
80100008:  03 a5 02 00          lw   0(t0),a0
8010000c:  6b 00 00 00          nemu trap
nemu: HIT GOOD TRAP at pc = 0x8010000c

```

图 2.1 单步执行结果

实现打印寄存器功能，只需要在框架代码 `nemu/src/isa/$ISA/reg.c` 目录下的 API 中 `void isa_reg_display(void)`，填写相对应的代码。ISA=riscv32，打开 ISA 对应的 `reg.c` 文件，可以看到其寄存器物理结构如下图 2.2 所示。

```

const char *regsl[] = {
    "$0", "ra", "sp", "gp", "tp", "t0", "t1", "t2",
    "s0", "s1", "a0", "a1", "a2", "a3", "a4", "a5",
    "a6", "a7", "s2", "s3", "s4", "s5", "s6", "s7",
    "s8", "s9", "s10", "s11", "t3", "t4", "t5", "t6"
};

```

图 2.2 寄存器物理结构

对其内容进行打印，需要用十六进制格式顺序输出寄存器名称和内容。得到的测试机俄国如图 2.3 所示。可以看出，该功能正确执行。

```

For help, type "help"
(nemu) info r
$0 = 0x00000000 , ra = 0x00000000 , sp = 0x00000000 , gp = 0x00000000 , tp = 0x0000
0000 , t0 = 0x00000000 , t1 = 0x00000000 , t2 = 0x00000000
s0 = 0x00000000 , s1 = 0x00000000 , a0 = 0x00000000 , a1 = 0x00000000 , a2 = 0x0000
0000 , a3 = 0x00000000 , a4 = 0x00000000 , a5 = 0x00000000
a6 = 0x00000000 , a7 = 0x00000000 , s2 = 0x00000000 , s3 = 0x00000000 , s4 = 0x0000
0000 , s5 = 0x00000000 , s6 = 0x00000000 , s7 = 0x00000000
s8 = 0x00000000 , s9 = 0x00000000 , s10 = 0x00000000 , s11 = 0x00000000 , t3 = 0x00
000000 , t4 = 0x00000000 , t5 = 0x00000000 , t6 = 0x00000000
pc = 0x80100000
(nemu)

```

图 2.3 寄存器内容

实现扫描内存，由于表达式求值暂未实现，所以对其先进行简化，对其简化形式实现。实现该功能，只需要调用框架中的 `paddr_read()` 即可。

对扫描内存的测试结果如图 2.4 所示，可见结果是正确的。

```
(nemu) x 1 0x80100000
      paddr  data
0x80100000  0x800002b7
```

图 2.4 扫描内存测试结果

表达式求值，之际上是与文法相关。对程序预先设置的 token 事别，再完成求值。表达式求值功能预先设置 token 的识别规则如下图 2.5 所示：

```
static struct rule {
    char *regex;
    int token_type;
} rules[] = {
    {" +", TK_NOTYPE},    // spaces
    {"0x[0-9a-f]+", TK_HEX}, // hex number
    {"[0-9]+", TK_DEC},    // dec number
    {"\\+", '+'},         // plus
    {"\\-", '-'},         // minus
    {"\\*", '*'},         // multiply
    {"\\/", '/'},         // divide
    {"\\(", '('},         // left parenthesis
    {"\\)", ')'},         // right parenthesis
    {"==", TK_EQ},        // equal
    {"!=", TK_NEQ},       // not equal
    {"&&", TK_AND},       // and
    {"\\$[0-9a-z$]+", TK_REG} // register
};
```

图 2.5 token 图

测试结果如图 2.6 所示，可以发现表达式求值功能能识别相对应的表达式并且正确求值。

```
(nemu) p 3+(3+2)*5
check parentheses is true, p = 2, q = 6
result = 0x1c 28
(nemu)
```

图 2.6 表达式求值功能测试结果图

监视点功能的实现主要考察对链表操作的管理，进行链表管理时，新建监视点从监视点池中取出一个界点，将表达式的值服于界点，将其放在链表头部。删除监视点，会根据链表节点 ID，将节点内容清空后把空间释放回监视点池。

完成该功能，对其进行测试，得到结果如图 2.7 所示。可见，成功创建并对监视点进行了一系列的操作。

```

come to riscv32-NEMU!
help, type "help"
mu) w $pc==0x80100004
chpoint 0: $pc==0x80100004
mu) info w
      EXPR
      $pc==0x80100004
mu) c
u: HIT GOOD TRAP at pc = 0x8010000c

```

```

(nemu) d 0
(nemu) info w
(nemu)

```

图 2.7 watchpoint 测试结果

2.3 实验问题

- 假设 500 次 NEMU 才能通过 PA，其中 90% 时间用于调试。如果没有实现简单调试器，只能通过 GDB 对其调试。每次调试，不能直接观测程序，需要花费 30s 来从错误信息中分析出有用信息。如果获取分析 20 个信息才足够排除一个 bug。那么调试会花费多长时间，节省多长时间？

A: $500 \times 0.9 \times 30 \times 20$ (调试); $500 \times 0.9 \times 20 \times 20$ (节省)

- Riscv 指令格式有哪几种？

A: R、I、S、B、U、J

- LUI 指令行为？

A: 加载高位立即数

- Mstatus 寄存器结构：

- A: 如图 2.8 所示：

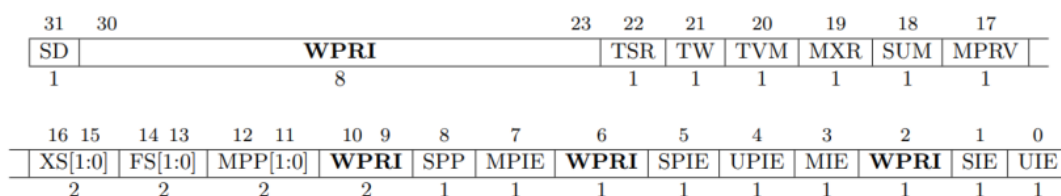


图 2.8 mstatus 寄存器结构

1. 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，在 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 文件共多少行代码?

A: 可以用 `find . -name "*.c|.h" | xargs wc -l` 指令获取 nemu/ 目录下所有.c 和.h 文件代码行数。`find . -name "*.c|.h" | xargs grep "^." | wc -l` 可用来获取去掉空行的数目。

执行结果分别是 5315, 4322

2. 使用 `man` 打开工程目录下的 Makefile, 你能在 CFLAGS 中看到 gcc 的编译选项。请问 `-Wall` 和 `-Werror` 有何作用? 为什么用 `-Wall` 和 `-Werror`?

A: `-Wall` 打开所有警告, 后者将警告当作错误处理。可以严格修改代码中 buggy 的点。

3 PA2

3.1 实验任务

1. 在 NEMU 运行 dummy
2. 实现更多指令，运行 cputest
3. 运行幻灯片播放和打字小游戏

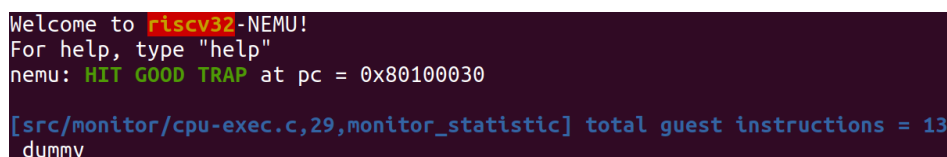
3.2 实验步骤与结果

本实验，需要我们先添加一些基本指令，使得计算机系统能够通过测试机。在本次实验中，我们需要实现的主要是译码和执行阶段，即 ID,EX.

本实验所提供的框架中，指令首先从 `isa_exec()` 开始，调用 `instr_fetch()` 获取当前 pc 对应的指令，紧接着调用 `idex()` 进行译码执行过程。

`Idex()` 根据指令 `opcode6_2` 字段的值在 `opcode_table` 选择合适的译码和执行函数完成指令的执行。这一过程描述了指令从取指到译码然后执行的整个过程。

运行 dummy，根据程序反汇编查看对应 pc 的指令，根据上述流程完成指令的设计后重新测试。实现对应测试的结果图如图 3.1 所示。



```
Welcome to riscv32-NEMU!  
For help, type "help"  
nemu: HIT GOOD TRAP at pc = 0x80100030  
[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 13  
dummy
```

图 3.1 dummy 运行结果图

再根据讲义提示，完成 cputest。在此之前需要先完成 string 和 hello-str 程序。完成后进行测试，测试结果如图 3.2 所示。

```
rust@rust-decktop:~/Desktop/ics2019/nemu$ bash runall.sh ISA=riscv32
compiling NEMU...
Building riscv32-nemu
make: Nothing to be done for 'app'.
NEMU compile OK
compiling testcases...
testcases compile OK
add-longlong] PASS!
add] PASS!
bit] PASS!
bubble-sort] PASS!
div] PASS!
dunny] 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!
novsx] PASS!
mul-longlong] PASS!
pascal] PASS!
prime] PASS!
quick-sort] PASS!
recursion] PASS!
select-sort] PASS!
shift] PASS!
shuxianhua] PASS!
string] PASS!
sub-longlong] PASS!
sum] PASS!
switch] PASS!
to-lower-case] PASS!
unaligned] PASS!
wanshu] PASS!
```

图 3.2 测试样例结果图

由于 ISA=riscv32，不需要进行额外代码对串口进行支持，运行 hello world 测试程序的结果如图 3.3 所示。

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

图 3.3 hello world 测试结果

Microbench 中 test 测试结果如图 3.4 所示。键盘测试结果如图 3.5 所示。

```
Welcome to riscv32-NEMU!
For help, type "help"
===== Running MicroBench [input *test*] =====
[qsrt] Quick sort: * Passed.
[queen] Queen placement: * Passed.
[bf] Brainf**k interpreter: * Passed.
[fib] Fibonacci number: * Passed.
[sieve] Eratosthenes sieve: * Passed.
[15pz] A* 15-puzzle search: * Passed.
[dinic] Dinic's maxflow algorithm: * Passed.
[lzip] Lzip compression: * Passed.
[ssort] Suffix sort: * Passed.
[md5] MD5 digest: * Passed.
=====
MicroBench PASS
Total time: 91 ms
nemu: HIT GOOD TRAP at pc = 0x801041e0
```

图 3.4 microbench 测试结果图

```

For help, type "help"
Try to press any key...
Get key: 49 J down
Get key: 49 J up
Get key: 36 I down
Get key: 36 I up
Get key: 30 W down
Get key: 30 W up
Get key: 57 X down
Get key: 57 X up

```

图 3.5 键盘测试结果图

根据讲义中的提示，AM 层面使用屏幕大小寄存器，硬件层面实现同步寄存器。完成后，测试结果如图 3.6 所示。幻灯片播放和打字小游戏的测试结果如图 3.7 所示。

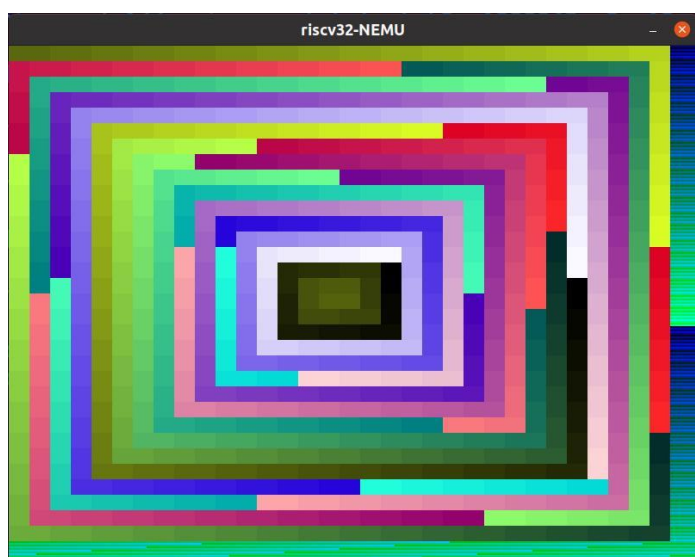


图 3.6 VGA 测试结果图

基于AM的教学生态系统

- 第一届龙芯杯比赛, 南京大学一队展示在CPU上运行教学操作系统Nanos和仙剑奇侠传

我们构建了完整的Project-N生态系统



4

<http://www.nccs.cc.org/uploads/soft/171010/1-1G010133147.pdf>

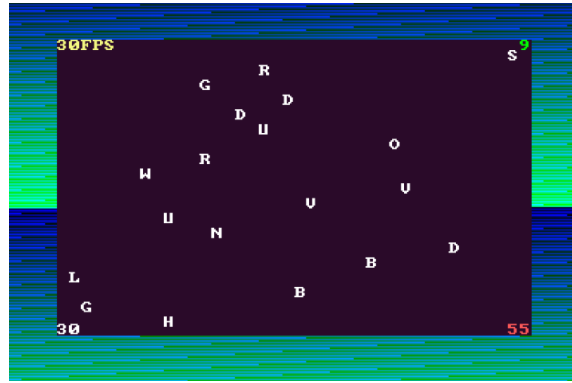


图 3.7 幻灯片和打字小游戏结果图

4 PA3（部分）

4.1 实验任务

1. 实现 `yield()` 及其过程
2. 实现用户程序的加载和系统调用，支撑 TRM 运行
3. 运行 PAL，展示批处理系统

4.2 实验步骤与结果

该实验中，需要了解计算机系统响应机制、系统调用、文件系统。

异常响应机制从设置异常入口地址并注册事件回调函数开始，`riscv32` 将异常入口地址保存在 `stvec` 寄存器中。`Yield90` 通过调用 `raise_intr()` 模拟异常响应机制。`Raise_intr()` 函数主要功能是将 `epc` 的值保存到 `SEPC` 中，异常号保存到 `SCAUSE` 中，最后 `pc` 跳转至 `SEPC` 中的值。

`Yield()` 使 CPU 转至 `__am_asm_trap` 中执行。根据其汇编代码段可以看到，程序把寄存器的值保存到栈后，令 CPU 转至 `__am_irq_handle` 中异常处理。该函数根据上下文结构指针保存的异常号，调用事件回调函数对异常进行处理。处理完成后，`__am_asm_trap` 汇编代码段根据保存到栈的值，对程序状态恢复，完成异常处理过程。

为加入系统调用，需要在 `Navy-apps` 中实现对应系统调用的接口使其通过 `yield()` 调用相应的系统调用处理函数完成流程。

完成上述代码后，对 `dummy` 测试，得到结果如图 4.1 所示。

```
event yield
nemu: HIT GOOD TRAP at pc = 0x80100e2c
```

图 4.1 dummy 测试结果

实现系统调用后，在 `nanos-lite` 上运行 `hello` 程序，结果如图 4.2 所示。

```

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!

```

图 4.2 hello 程序测试图

系统能支持多程序，需要我们了解每个程序位于 ramdisk 上的具体位置。因而，操作系统提供文件抽象，以达到该目的。

完成文件系统的设计实现后，对测试程序 text 进行测试，结果如图 4.3 所示。

```

Welcome to riscv32-NEMU!
For help, type "help"
This is the LOGO!

PASS!!!
nemu: HIT GOOD TRAP at pc = 0x80100e2c

```

图 4.3 text 测试结果图

此外，需要对读写函数指针对应的读写函数进行实现，实现 events_read()函数对键盘进行读取后，在文件系统加入对/dev/events 的支持，events 测试结果如图 4.4 所示。

```

Welcome to riscv32-NEMU!
For help, type "help"
This is the LOGO!
[/home/hust/Desktop/ics2019/nanos-lite/src/main.c,
Nanos-lite
, Jan 6 2022
[/home/hust/Desktop/ics2019/nanos-lite/src/main.c,
info: start = 0x8010316c, end = 0x8224cdf1, size =
[/home/hust/Desktop/ics2019/nanos-lite/src/device.
g devices...
[/home/hust/Desktop/ics2019/nanos-lite/src/irq.c,2
rrupt/exception handler...
[/home/hust/Desktop/ics2019/nanos-lite/src/proc.c,
processes...
[/home/hust/Desktop/ics2019/nanos-lite/src/loader.
ry = 83000180
Start to receive events...
receive time event for the 1024th time: t 7703
receive event: kd J
receive event: ku J
receive time event for the 2048th time: t 11488

```

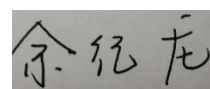
图 4.4 events 测试结果图

5 小结

该实验总体来说耗时长，难度很大。一开始对计算机系统架构也没有真正到沉入了工程中，感觉计算机系统之间的关联，整个庞然大物没有一块不精细的“肥肉”。很多时候遇到了困难，多亏了有以前同学的帮助，还有 PA 详细的文档说明，才能摸索出来。

总之，经过计算机系统能力训练的一次洗礼，让我明白了其实计算机在 高处，仍旧有我目光短浅看不见的有趣。真正经历珠圆玉润的打磨过程，才有先辈们拿出来的机器珠玑奥妙之处。

签名：

Handwritten signature in black ink on a light gray rectangular background. The characters are '余纪龙' (Yu Jilong).

参考文献

- [1] 南京大学计算机科学与技术系计算机系统基础课程实验 2019. nju-projectn.github.io/ics-pa-gitbook/ics2019
- [2] RISC-V 手册. <https://course.cunok.cn/pa/RISC-V-Reader-Chinese-v2p1.pdf>
- [3] RISC-V ISA Specification Volume 1. <https://course.cunok.cn/pa/riscv-spec-20191213.pdf>
- [4] RISC-V ISA Specification Volume 2. <https://course.cunok.cn/pa/riscv-privileged-20190608.pdf>