

华中科技大学

2022

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: CS1902 班

学 号: U201914942

姓 名: 义俊林

电 话: 13659641737

邮 件: 1745498973@qq.com

完成日期: 2022-10-19



目 录

| | | |
|-------|---|----|
| 1 | 课程实验概述 | 1 |
| 1.1 | 课设目的 | 1 |
| 1.2 | 课程任务 | 1 |
| 1.3 | 实验环境 | 2 |
| 2 | 实验方案设计与结果分析 | 4 |
| 2.1 | PA1 - 开天辟地的篇章: 最简单的计算机 | 4 |
| 2.1.1 | PA1.1 实现单步执行, 打印寄存器状态, 扫描内存 | 4 |
| 2.1.2 | PA1.2 实现算术表达式求值 | 5 |
| 2.1.3 | PA1.3 实现所有要求, 提交完整的实验报告 | 7 |
| 2.1.4 | PA1 必答题 | 9 |
| 2.2 | PA2 - 简单复杂的机器: 冯诺依曼计算机系统 | 11 |
| 2.2.1 | PA2.1 在 NEMU 中运行第一个 C 程序 dummy | 11 |
| 2.2.2 | PA2.2 实现更多的指令, 在 NEMU 中运行所有 cputest | 12 |
| 2.2.3 | PA2.3 运行打字小游戏, 提交完整的实验报告 | 14 |
| 2.2.4 | PA2 必答题 | 20 |
| 2.3 | PA3 - 穿越时空的旅程: 批处理系统 | 21 |
| 2.3.1 | PA3.1 实现自陷操作 _yield() 及其过程 | 21 |
| 2.3.2 | PA3.2 实现用户程序的加载和系统调用, 支撑 TRM 程序的运行 22 | |
| 2.3.3 | PA3.3 运行仙剑奇侠传并展示批处理系统, 提交完整的实验报告 24 | |
| 2.3.4 | PA3 必答题 | 29 |
| 3 | 总结与心得 | 32 |
| 3.1 | PA1 | 32 |
| 3.2 | PA2 | 32 |
| 3.3 | PA3 | 33 |
| 4 | 电子签名 | 34 |

1 课程实验概述

1.1 课设目的

基于 riscv32 架构实现一个经过简化但是功能完备得模拟器 NEMU (NJU EMUlator)，最终在 NEMU 上运行游戏“仙剑奇侠传”，通过实验探索程序在计算机上运行的原理。

主要实验内容如下：

- (1) 图灵机与简易调试器；
- (2) 冯诺依曼计算机系统；
- (3) 批处理系统；
- (4) 分时多任务；
- (5) 程序性能优化；

通过实验：

- (1) 提升学生的计算机系统层面的认知与设计能力，能从计算机系统的高度考虑和解决问题；
- (2) 培养学生具有系统观的，能够进行软，硬件协同设计的思维认知；
- (3) 培养学生对系统有深刻的理解，能够站在系统的高度考虑和解决应用问题的。

1.2 课程任务

- (1) PA0 - 世界诞生的前夜：开发环境配置
 - 安装合适的虚拟机；
 - git clone 框架代码并阅读框架代码；
- (2) PA1 - 开天辟地的篇章：最简单的计算机
 - PA1.1: 实现单步执行，打印寄存器状态，扫描内存
 - PA1.2: 实现算术表达式求值
 - PA1.3: 实现所有要求，提交完整的实验报告
- (3) PA2 - 简单复杂的机器：冯诺依曼计算机系统
 - PA2.1: 在 NEMU 中运行第一个 C 程序 dummy
 - PA2.2: 实现更多的指令，在 NEMU 中运行所有 cputest
 - PA2.3: 运行打字小游戏，提交完整的实验报告
- (4) PA3 - 穿越时空的旅程：批处理系统
 - PA3.1: 实现自陷操作 `_yield()` 及其过程
 - PA3.2: 实现用户程序的加载和系统调用，支撑 TRM 程序的运行
 - PA3.3: 运行仙剑奇侠传并展示批处理系统，提交完整的实验报告
- (5) PA4 - 虚实交错的魔法：分时多任务
 - PA4.1: 实现基本的多道程序系统
 - PA4.2: 实现支持虚存管理的多道程序系统
 - PA4.3: 实现抢占式分时多任务系统，并提交完整的实验报告

(6) PA5 - 天下武功唯快不破: 程序与性能

1.3 实验环境

- (1) CPU 架构: x64
- (2) 平台: windows+ VirtualBox+Ubuntu64
- (3) 操作系统: gnu/linux
- (4) 编译器: gcc、riscv-none-embed-gcc
- (5) 编程语言: c 语言



图 1.1 windows 平台配置

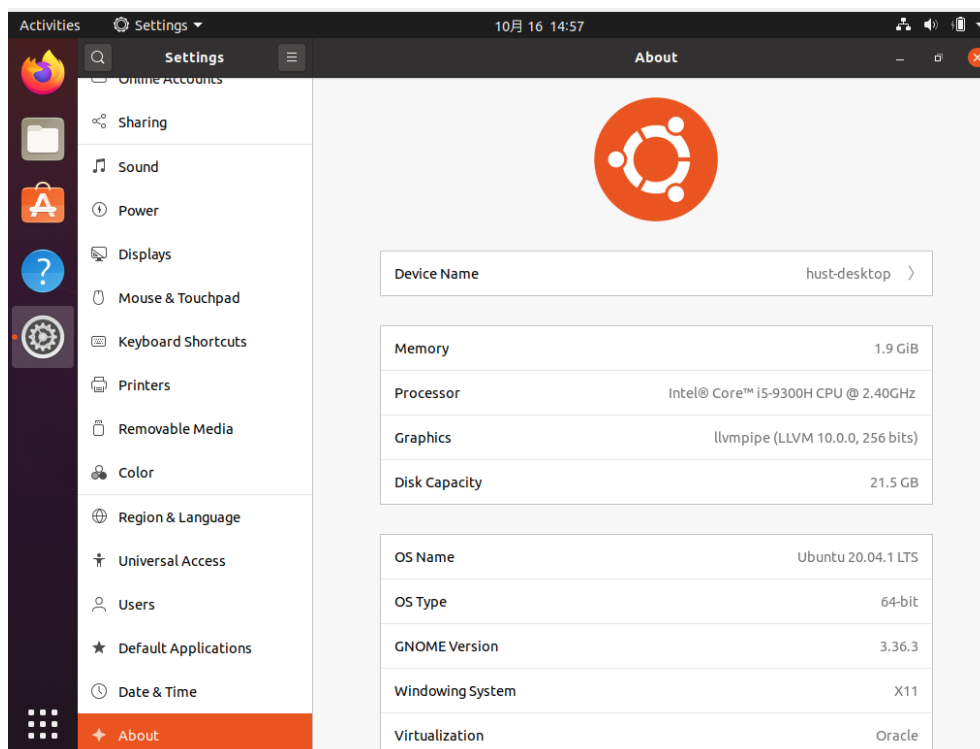


图 1.2 ubuntu 平台配置

```
hust@hust-desktop:~/Desktop$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none:hsa
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 9.3.0-10ubuntu2' --with-bugurl=file:///usr/share/doc/gcc-9/README.Bugs --enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++,gm2 --prefix=/usr --with-gcc-major-version-only --program-suffix=-9 --program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-verify --enable-plugin --enable-default-pie --with-system-zlib --with-target-system-zlib=auto --enable-objc-gc=auto --enable-multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-offload-targets=nvptx-none,hsa --without-cuda-driver --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 9.3.0 (Ubuntu 9.3.0-10ubuntu2)
```

图 1.3 gcc 的版本

2 实验方案设计与结果分析

2.1 PA1 - 开天辟地的篇章: 最简单的计算机

PA1 的主要内容是实现一个 monitor, 即一个简易的 gdb, 其功能如下:

- (1) **help**: 打印命令的帮助信息
- (2) **c**: 继续运行被暂停的程序
- (3) **q**: 退出 NEMU
- (4) **si [N]**: 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
- (5) **info SUBCMD**: **info r** 打印寄存器状态; **info w** 打印监视点信息
- (6) **p EXPR**: 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
- (7) **x N EXPR**: 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
- (8) **w EXPR**: 当表达式 EXPR 的值发生变化时, 暂停程序执行
- (9) **d N**: 删除序号为 N 的监视点

2.1.1 PA1.1 实现单步执行, 打印寄存器状态, 扫描内存

实验涉及修改的文件如下:

- (1) `nemu/src/monitor/debug/ui.c`
- (2) `nemu/src/isa/riscv32/reg.c`

实验内容:

- (1) 单步执行: 在 `ui.c` 中定义单步执行函数 `static int cmd_si(char *args)`, 其中传入的 `args` 参数为整型或者为 NULL, 使用 `atoi` 函数将参数 `args` 转为 `int` 类型的值 `n`, 然后再调用函数 `cpu_exec` 执行 `n` 步即可。
- (2) 打印寄存器的状态: 在 `ui.c` 中定义打印寄存器状态的函数 `static int cmd_info(char *args)`, 其中 `args` 的值应为 `w` 字符。在 `reg.c` 中补充 `isa_reg_display()` 函数的内容。然后在 `cmd_info` 中调用 `isa_reg_display()` 函数。
- (3) 扫描内存: 在 `ui.c` 中定义扫描内存函数 `static int cmd_x(char *args)`, 其中 `args` 传入两个参数, 一个为扫描内存的数量 `N`, 使用 `atoi` 函数解析, 另一个是扫描内存的初始地址, 使用 `expr` 函数解析, 不过表达式 `expr` 函数在 PA1.2 节中设计, 故使 `expr` 永远返回 0。调用 `vaddr_read` 宏, 其原型是 `isa_vaddr_read(vaddr_t, int)` 函数, 循环读取起始地址后的 `N` 个 4 字节的数据。

实验结果:

```

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

```

图 2.1 单步执行

```

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

```

图 2.2 打印寄存器指令

由于没有实现 `expr` 函数，故在 PA1.2 节再展示扫描内存的功能。

2.1.2 PA1.2 实现算术表达式求值

实验涉及修改的文件如下：

- `ics2019/nemu/src/monitor/debug/expr.c`
- `nemu/src/moniter/debug/ui.c`

实验内容：

- (1) 添加更多的 token:
 - `TK_INT`: 正整数
 - `TK_HEX`: 十六进制数
 - `TK_REG`: 寄存器名称

- TK_NOTEQ: 不等于 !=
 - TK_AND: 并运算 &&
 - TK_OR: 或运算 ||
 - TK_DEREFERENCE: 解析*
 - TK_NEGATIVE: 负数-
- (2) 补充规则和正则表达式: 规则就是解析一串字符为 token 的规则, 在 Token 结构体中有整型成员 type 记录 token 的类型和字符数组成员 str 记录 token 的 ID。例如 [0-9]+ 就是 token TK_INT 的正则表达式, 其余不一一列出。
 - (3) 解析 token: 在 make_token 函数中实现解析字符串中 token 的功能, 并依次将 token 的类型记录在 tokens 数组中。
 - (4) 运算符优先级函数: 定义函数 int operator_priority(int op) 来返回运算符的优先级, 优先级 1~7, 其中 1 为最高优先级。
 - (5) 括号匹配函数: 定义函数 bool check_parentheses(int p, int q), 其功能是检测边缘括号匹配问题。
 - (6) 表达式计算函数: 定义函数 uint32_t eval(int p, int q, bool *success), 其中 p, q 是表达式的范围, success 接收表达式是否成功。首先表达式检测是否 p>q, 若是则表达式错误; 然后检测是否 p==q, 若是表示这个 token 是一个数值或寄存器; 然后检测是否 check_parentheses(p, q) == true, 即 pq 是 () 的 token, 则递归 eval(p+1, q-1, success) 计算值; 然后是其他情况, 即找主运算操作符。
 - (7) 完成 expr 函数: 完成对 TK_DEREFERENCE 和 TK_NEGATIVE 类型 token 的解析, 再调用 eval 函数计算表达式的值。
 - (8) 在 ui.c 定义 static int cmd_p(char *args) 函数, 其中 args 为表达式, 调用 expr.c 文件中的 expr 函数计算表达式的值, 输出数据为十进制和十六进制, 若表达式错误则输出 expression error 信息。

实验结果:

```
Welcome to riscv32-NEMU!
For help, type "help"
(nemu) x 4 $pc
80100000
0x80100000 : 0x800002b7
0x80100004 : 0x0002a023
0x80100008 : 0x0002a503
0x8010000c : 0x0000006b
```

图 2.3 扫描内存指令


```

Welcome to riscv32-NEMU!
For help, type "help"
(nemu) p $pc+123
decimal: -2146434949
hex    : 0x8010007b
(nemu) p $pc
decimal: -2146435072
hex    : 0x80100000
(nemu) p -1+2
decimal: 1
hex    : 0x1
(nemu) p (6-2)/2+(3+1)*5
decimal: 22
hex    : 0x16
(nemu) p ((1-1)
expression error

```

图 2.4 表达式求值

2.1.3 PA1.3 实现所有要求, 提交完整的实验报告

实验涉及修改的文件如下:

- nemu/src/monitor/debug/ui.c
- nemu/include/monitor/watchpoint.h
- nemu/src/monitor/debug/watchpoint.c
- nemu/src/monitor/cpu-exec.c

实验内容:

- (1) 修改 WP 结构体的内容: 在 watchpoint.h 中为 WP 结构体添加 char expr[32] 成员和 uint32_t value 成员, 其作用分别为记录表达式的名称和表达式的值。
- (2) 创建监视点: 在 watchpoint.c 中定义函数 WP* new_wp(uint32_t value, char *expr), value 为 expr 函数计算表达式返回的值, expr 为名称。监视点数量有限制, 是一个容量大小为 32 的 WP 数组 wp_pool。wp_pool 数组由 WP* 类型的 head 指针和 free_指针管理。head 指针管理存有监视点的 wp_pool 的元素, free_指针管理 wp_pool 剩余元素。创建监视点时, 首先检查 free_是否为 NULL, 若为 NULL 代表没有空间了, 否则使用头插法将 free_指向的首结点插入 head 链表中, free_指向下一结点。再将数据写入刚刚插入 head 中的结点即可。在 ui.c 中定义创建监视点命令的函数 static int cmd_w(char *args), args 接收一个表达式。调用 new_wp 函数创建监视点。
- (3) 删除监视点: 在 watchpoint.c 中定义函数 WP* free_wp(int NO), NO 为监视点的序号。首先检查 NO 的范围是否在 0~31 并且 head 不等于 NULL, 若是则返回 NULL, 否则遍历 head 链表检查结点的序号是否等于 NO, 若有则从 head 链表中删除并将该结点插入到 free_链表中, 返回该结点的地址, 否则返回 NULL。在 ui.c 中定义删除监视点命令的函数 static int cmd_d(char *args), args 接收一个整型数。调用 free_wp 函数删除监视点。
- (4) 显示监视点: 在 watchpoint.c 中定义函数 void wp_display(), 其功能是遍历 head 链表, 并将结点的信息显示出来。显示监视点是命令 info 的功能, 该功能在 PA1.1 已经定义, 若 args 接收的字符为 w 则调用 wp_display 函数显示监视点列表的值。

- (5) 对于添加的监视点,在执行过程中若发生改变,则应该使程序暂停执行。完成此功能需要在 `watchpoint.c` 中定义函数 `bool wp_update()`函数,其功能使遍历 `head` 链表,重新计算表达式的值与旧值进行比较,若表达式的值与旧值不同,则更新旧值,返回 `false`,否则返回 `true`。在 `cpu-exec.c` 中调用 `wp_update()`函数,若其返回 `false`,则设置 `nemu_state.state = NEMU_STOP`。

实验结果:

首先创建两个监视点,然后显示监视点列表,使用 `si` 指令执行 4 步,执行过程中 `t0` 寄存器的值发生了改变故暂停执行,删除监视点 1 在显示监视点列表,发现监视点 1 已经删除。

```
Welcome to riscv32-NEMU!
For help, type "help"
(nemu) w $t0
watchpoint $t0 set success
(nemu) w 1+2
watchpoint 1+2 set success
(nemu) info w
NO  EXPR          DECIMAL    HEX
1   1+2           3          0x3
0   $t0           0          0x0
(nemu) si 4
80100000: b7 02 00 80          lui  0x80000,t0
0   $t0          -2147483648  0x80000000
watchpoint changed
(nemu) d 1
watchpoint 1 1+2 3 delete success
(nemu) info w
NO  EXPR          DECIMAL    HEX
0   $t0          -2147483648  0x80000000
```

图 2.5 监视点功能展示

2.1.4 PA1 必答题

□ 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 .
- **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, 这些数字假设你能够通过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有哪几种指令格式?
 - CP0寄存器是什么?
 - 若除法指令的除数为0, 结果会怎样?
 - riscv32
 - riscv32有哪几种指令格式?
 - 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` ?

图 2.6 必答题

- (1) 我选择的 ISA 是 `riscv32`
- (2) GDB 调试花费的时间为 $500 \times 0.9 \times 30 \times 20 = 270000s$; 简易调试器节省的时间为 $270000 - 500 \times 0.9 \times 10 \times 20 = 180000s$

(3)

- riscv32 有哪几种指令格式？

riscv32 有 6 种指令格式，分别为 R、I、S、B、U、J 类型。其中 rx 表示寄存器的标号， imm_{xx} 表示立即数的值，opcode 表示操作码， $funct7$ 、 $funct3$ 和 opcode 用来区分同类型指令。

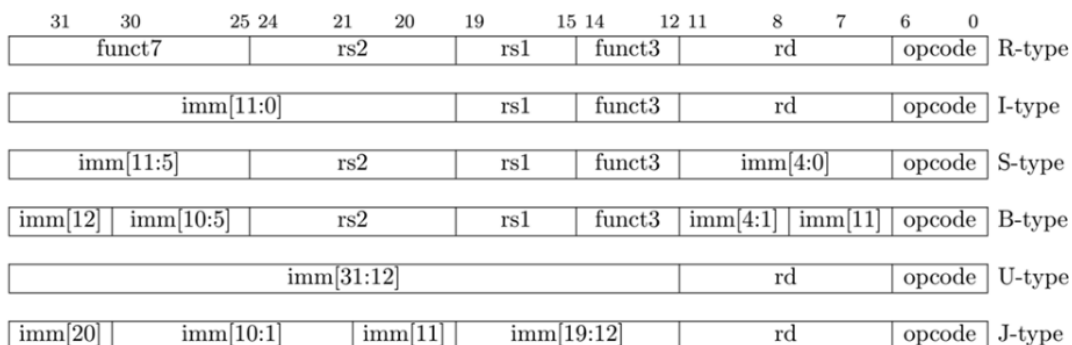


图 2.7 指令类型

- LUI 指令的行为是什么？

`lui rd, immediate` 表示 $x[rd] = sext(immediate[31:12] \ll 12)$ ，即 20 位立即数 `immediate` 左移 12 位，并将低 12 位置零，写入 $x[rd]$ 中。

- mstatus 寄存器的结构是怎么样的？

`mstatus` (Machine Status) 它保存全局中断使能，以及许多其他的状态。

3.1.6 Machine Status Registers (`mstatus` and `mstatush`)

The `mstatus` register is an `MXLEN`-bit read/write register formatted as shown in Figure 3.6 for RV32 and Figure 3.7 for RV64. The `mstatus` register keeps track of and controls the hart's current operating state. A restricted view of `mstatus` appears as the `sstatus` register in the S-level ISA.

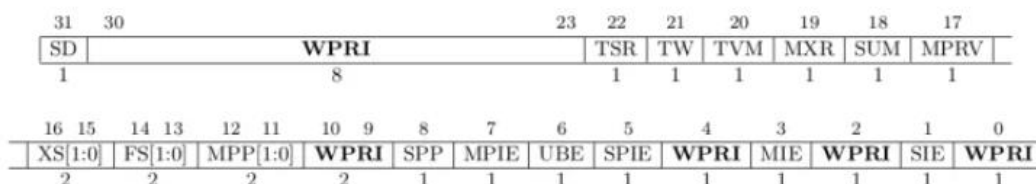


Figure 3.6: Machine-mode status register (`mstatus`) for RV32.

图 2.8 mstatus 信息

(4)

- `nemu/` 目录下的所有 `.c` 和 `.h` 文件总共有多少行代码？

使用命令 `find . -name "*.c|.h" -type f | xargs wc -l` 得到 5563 行。

- 去空行之后，`nemu/` 目录下的所有 `.c` 和 `.h` 文件总共有多少行代码？

使用命令 `find . -name "*.c|.h" -type f | xargs grep -v ^$ | wc -l` 得到 4578 行。

(5)

`-Wall` 使 GCC 编译后显示所有的警告信息。`-Werror` 会将所有的警告当成错误进行处理，并且取消编译操作。使用 `-Wall` 和 `-Werror` 就是为了找出可能存在的错误，尽可能地避免程序运行出错，优化程序。

2.2 PA2 - 简单复杂的机器：冯诺依曼计算机系统

基于自选的 riscv32 架构设计足够多的指令以便能够通过各种测试、支持输入输出设备。

2.2.1 PA2.1 在 NEMU 中运行第一个 C 程序 dummy

准备交叉编译环境：

交叉编译器的下载地址：<https://github.com/ilg-archived/riscv-none-gcc/releases/download/v8.1.0-2-20181019/gnu-mcu-eclipse-riscv-none-gcc-8.1.0-2-20181019-0952-centos64.tgz>

下载完毕后，进入压缩包所在的目录

```
hust@hust-desktop:~/gnu-mcu-eclipse$ tar -xvf riscv-none-gcc-8.1.0-2-20181019.tgz gnu-mcu-eclipse/
```

图 2.9 解压缩包

然后在 ~/.bashrc 文件中添加如下内容添加 PATH 路径

```
export PATH=$PATH:/home/hust/gnu-mcu-eclipse/riscv-none-gcc/8.1.0-2-20181019-0952/bin
```

图 2.10 添加 path 路径

然后检查添加路径是否成功

```
hust@hust-desktop:~/gnu-mcu-eclipse$ riscv-none-embed-gcc  
riscv-none-embed-gcc: fatal error: no input files  
compilation terminated.
```

图 2.11 输出 compilation terminated 表示成功

实验涉及修改的文件：

- nemu/src/isa/riscv32/exec/all-instr.h
- nemu/src/isa/riscv32/exec/compute.c
- nemu/src/isa/riscv32/exec/control.c
- nemu/src/isa/riscv32/exec/exec.c
- nemu/src/isa/riscv32/decode.c
- nemu/src/isa/riscv32/include/isa/decode.h

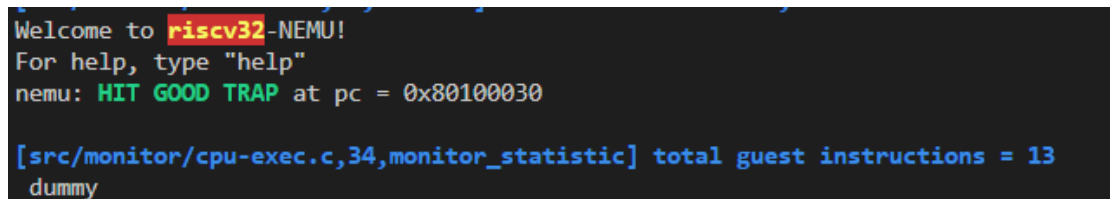
实验内容：

- (1) 首先在 nexus-am/tests/cputest 目录下执行 `make ARCH=riscv32-nemu ALL=dummy run` 命令，然后可以找到在目录 `nexus-am/tests/cputest/build/` 找到 `dummy-riscv32-nemu.txt` 文件，查看其反汇编代码，可以发现有一部分指令没有实现，没有实现的指令是 `auipc` (U 型指令)、`addi` (I 型指令)、`jal` (J 型指令) 和 `jalr` (即 `ret`, I 型指令)，需要实现这几个指令才能正常运行 `dummy.c`。
- (2) 声明译码辅助函数：在 `decode.h` 文件中声明译码辅助函数，即将 `decode.c` 文件中的定义的函数声明到该头文件中。
- (3) 添加译码辅助函数：在 `decode.c` 文件中，已经定义了 U 型指令的译码辅助函数，接下来需要 I 型指令和 J 型指令的译码辅助函数。所谓译码辅助函数也就是解析指令各个比特的信息，其有效信息存进 `DecodeInfo` 结构体中。例如 I 型指令有 `rs1` 字段、立即数字段和 `rd` 字段，则译码的主要作用就是将这些字段的信息存进 `DecodeInfo` 结构体对应的 `Operand` 成

员中。其他类型指令不一一介绍。

- (4) 声明执行辅助函数：在 `all-instr.h` 文件中添加需要编写的执行辅助函数。
- (5) 定义执行辅助函数：`auipc` 指令属于计算类型，在 `compute.c` 中定义 `make_EHelper(auipc)` 函数，调用 `rtl` 函数实现 `auipc` 指令的功能。`addi` 指令是 I 型指令，属于计算类型，在 `compute.c` 定义，由于之后实验需要的 I 型指令有很多，它们的 `opcode` 字段有很多都相同，在 `exec.c` 中 `opcode_table` 表根据指令的 `opcode` 来确定这个指令使用什么译码和执行辅助函数，因此将 `opcode` 为 4 的 I 型指令的执行辅助函数定义为 `make_EHelper(I_opcode_4)`，根据 I 型指令的 `funct3` 和 `funct7` 字段来区分不同的 I 型指令，其中 `addi` 指令的 `funct3` 字段为 0，调用 `rtl` 函数实现 `addi` 指令。`jal` 和 `jalr` 指令属于控制转移指令，在 `control.c` 中定义，分别定义为 `make_EHelper(jal)` 和 `make_EHelper(jalr)`，调用 `rtl` 函数实现即可。
- (6) 完善 `opcode_table`：在 `exec.c` 中，根据自己定义的译码辅助函数、执行辅助函数和指令的 `opcode` 来填写 `opcode_table` 表。
- (7) 完成如上工作后再运行 `make ARCH=riscv32-nemu ALL=dummy run` 命令，`dummy` 成功 HIT GOOD TRAP。

实验结果：



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

[src/monitor/cpu-exec.c,34,monitor_statistic] total guest instructions = 13
dummy
```

图 2.12 成功运行 dummy 程序

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

实验涉及修改的文件：

- `nemu/src/isa/riscv32/exec/all-instr.h`
- `nemu/src/isa/riscv32/exec/compute.c`
- `nemu/src/isa/riscv32/exec/control.c`
- `nemu/src/isa/riscv32/exec/exec.c`
- `nemu/src/isa/riscv32/decode.c`
- `nemu/src/isa/riscv32/include/isa/decode.h`
- `nexus-am/Makefile.check`
- `nexus-am/libs/klib/src/stdio.c`
- `nexus-am/libs/klib/src/string.c`
- `nemu/include/common.h`
- `nemu/src/isa/riscv32/diff-test.c`

实验内容：

- (1) 修改 `Makefile.check` 文件，使 AM 项目中的程序默认编译到 `riscv32-nemu` 的 AM 中


```
## ARCH ?= native
ARCH ?= riscv32-nemu
```

图 2.13 Makefile.chec 修改内容

- (2) 实现 diff-test: 在 common.h 定义 DIFF_TEST 宏, 表示开启 diff-test。在 diff-test.c 文件编写 isa_difftest_checkregs(CPU_state *ref_r, vaddr_t pc) 函数, 比较 ref_r 中 32 个寄存器的值与 cpu 中 32 个寄存器的值是否相同, 不相同则返回 false。
- (3) 校准指令: 开启了 diff-test 后, 有一些指令需要校准, 对于 risc32 来说其 jalr 指令需要添加 difftest_skip_dut(1, 2) 来进行校准。
- (4) 添加更多的指令来运行更多的程序, nexus-am/tests/cputest 有很多测试程序, 需要添加指令, 使得能够运行所有的测试。添加指令的这个过程和上一节实验的过程类型, 其过程都是 “查看汇编代码->添加译码辅助函数->添加执行辅助函数->填写 opcode_table”。
- (5) 对于大多数 nexus-am/tests/cputest 下的程序添加指令就能通过测试, 但是对于 string.c 和 hello-str.c 两个程序则不同, 其需要完成 nexus-am/libs/klib/src 目录下的 stdio.c 和 string.c 自定义库函数。实现 stdio.c 和 string.c 文件中的库函数就可以使 string.c 和 hello-str.c 两个程序通过测试。
- (6) 在 nemu 目录下使用命令 bash runall.sh ISA=riscv32 进行一键回归测试。

实验结果:

```
hust@hust-desktop:~/hust/ics2019/nemu$ bash runall.sh ISA=riscv32
compiling NEMU...
Building riscv32-nemu
make: Nothing to be done for 'app'.
NEMU compile OK
compiling testcases...
testcases compile OK
[ add-longlong] PASS!
[ add] PASS!
[ bit] PASS!
[ bubble-sort] PASS!
[ div] PASS!
[ dummy] PASS!
[ fact] PASS!
[ fib] PASS!
[ goldbach] PASS!
[ hello-str] PASS!
[ if-else] PASS!
[ leap-year] PASS!
[ load-store] PASS!
[ matrix-mul] PASS!
[ max] PASS!
[ min3] PASS!
[ mov-c] PASS!
[ movsx] PASS!
[ mul-longlong] PASS!
[ pascal] PASS!
[ prime] PASS!
[ quick-sort] PASS!
[ recursion] PASS!
[ select-sort] PASS!
[ shift] PASS!
[ shuixianhua] PASS!
[ string] PASS!
[ sub-longlong] PASS!
[ sum] PASS!
[ switch] PASS!
[ to-lower-case] PASS!
[ unalign] PASS!
[ wanshu] PASS!
```

图 2.14 一键回归测试

2.2.3 PA2.3 运行打字小游戏, 提交完整的实验报告

实验涉及修改的文件:

- nemu/src/device/vga.c
- nemu/include/common.h
- nexus-am/am/src/nemu-common/nemu-input.c
- nexus-am/am/src/nemu-common/nemu-timer.c
- nexus-am/am/src/nemu-common/nemu-video.c

实验内容:

- (1) 本节实验任务是完成串口、时钟、键盘和 VGA 输入输出设备程序的编写。在 common.h 中定义宏#define HAS_IOE 开启设备。
- (2) 串口设备程序: 框架代码已经在 nexus-am/am/src/nemu-common/trm.c 中提供。由于串口已经完成, 在 nexus-am/tests/amtest/目录下输入命令 make mainargs=h run 即可测试 hello.c 程序, 终端会输出 10 行 Hello, AM World @ riscv32。
- (3) 时钟设备程序: 在 nemu-timer.c 文件中, 定义静态变量 start_time 接收 riscv32-nemu 的启动时间, 在 __am_timer_init 函数中, 调用 inl 函数获取地址 RTC_ADDR 的时间。然后在 __am_timer_read 函数中, 定义临时变量 cur_time 接收 inl 函数读取地址 RTC_ADDR 的返回值, uptime->lo = cur_time - start_time 为当前时间与启动时间的差值, uptime->hi=0。完成时间设备程序后, 就可以在 nexus-am/tests/amtest/目录下输入命令 make mainargs=t run 测试 rtc.c 程序, 终端会输出 2000-0-0 2d:2d:2d GMT (1 second)., 其中 second 依次递增。
- (4) 键盘设备程序: 在 nemu-input.c 文件中, 调用 inl 函数读取 KBD_ADDR 获取键盘的活动信息, 将其返回值存入 kbd->keycode 中, kbd -> keycode & KEYDOWN_MASK 的真值存入 kbd -> keydown 中, 其中 KEYDOWN_MASK = 0x8000, 是键盘掩码。在 nexus-am/tests/amtest/目录下输入命令 make mainargs=k run 测试 keyboard.c 程序, 假如实现正确, 按下按钮和松开按钮都会输出对应键盘的信息。
- (5) VGA 设备程序: 在 vga.c 文件中 vga_io_handle 函数内, 添加如下代码, 如果 vga 设备发生了写事件, 则需要调用 update_screen 函数。在 nemu-video.c 文件中 __am_video_read 函数中初始化 _DEV_VIDEO_INFO_t *变量 info 的长和宽, 然后在 __am_video_write 实现对 vga 设备的写入, 其需要将 pixels 数组中的信息写入到 fb 数组中。同时在 __am_vga_init 添加讲义中的代码, 即可初始化为一块五彩缤纷的屏幕。

实验结果:


```

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
Hello, AM World @ riscv32
nemu: HIT GOOD TRAP at pc = 0x80100e60

[src/monitor/cpu-exec.c,34,monitor_statistic] total guest instructions = 2188

```

图 2.15 hello.c 程序测试

```

Welcome to riscv32-NEMU!
For help, type "help"
2000-0-0 2d:2d:2d GMT (1 second).
2000-0-0 2d:2d:2d GMT (2 seconds).
2000-0-0 2d:2d:2d GMT (3 seconds).
2000-0-0 2d:2d:2d GMT (4 seconds).
2000-0-0 2d:2d:2d GMT (5 seconds).
2000-0-0 2d:2d:2d GMT (6 seconds).
2000-0-0 2d:2d:2d GMT (7 seconds).
2000-0-0 2d:2d:2d GMT (8 seconds).
2000-0-0 2d:2d:2d GMT (9 seconds).

```

图 2.16 rtc.c 程序测试

```

Welcome to riscv32-NEMU!
For help, type "help"
Dhrystone Benchmark, Version C, Version 2.2
Trying 500000 runs through Dhrystone.
Finished in 13474 ms
=====
Dhrystone PASS          65 Marks
                        vs. 100000 Marks (i7-7700K @ 4.20GHz)
nemu: HIT GOOD TRAP at pc = 0x80100d64

```

图 2.17 dhrystone 测试

```

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)    : 16092
Iterations         : 1000
Compiler version   : GCC8.2.0
seedcrc            : 0x4x
[0]crclist         : 0x4x
[0]crcmatrix       : 0x4x
[0]crcstate        : 0x4x
[0]crcfinal        : 0x4x
Finised in 16092 ms.

=====
CoreMark PASS      181 Marks
                   vs. 100000 Marks (i7-7700K @ 4.20GHz)
nemu: HIT GOOD TRAP at pc = 0x8010224c

```

图 2.18 coremark 测试

```

Welcome to riscv32-NEMU!
For help, type "help"
===== Running MicroBench [input *ref*] =====
[qsort] Quick sort: * Passed.
      min time: 1390 ms [367]
[queen] Queen placement: * Passed.
      min time: 1833 ms [256]
[bf] Brainf**k interpreter: * Passed.
      min time: 10058 ms [235]
[fib] Fibonacci number: * Passed.
      min time: 20675 ms [136]
[sieve] Eratosthenes sieve: * Passed.
      min time: 21723 ms [181]
[15pz] A* 15-puzzle search: * Passed.
      min time: 2750 ms [163]
[dinic] Dinic's maxflow algorithm: * Passed.
      min time: 3582 ms [303]
[lzip] Lzip compression: * Passed.
      min time: 2958 ms [256]
[ssort] Suffix sort: * Passed.
      min time: 1275 ms [353]
[md5] MD5 digest: * Passed.
      min time: 17514 ms [98]
=====
MicroBench PASS    234 Marks
                   vs. 100000 Marks (i7-7700K @ 4.20GHz)
Total time: 95641 ms
nemu: HIT GOOD TRAP at pc = 0x801041e0

```

图 2.19 microbench 测试

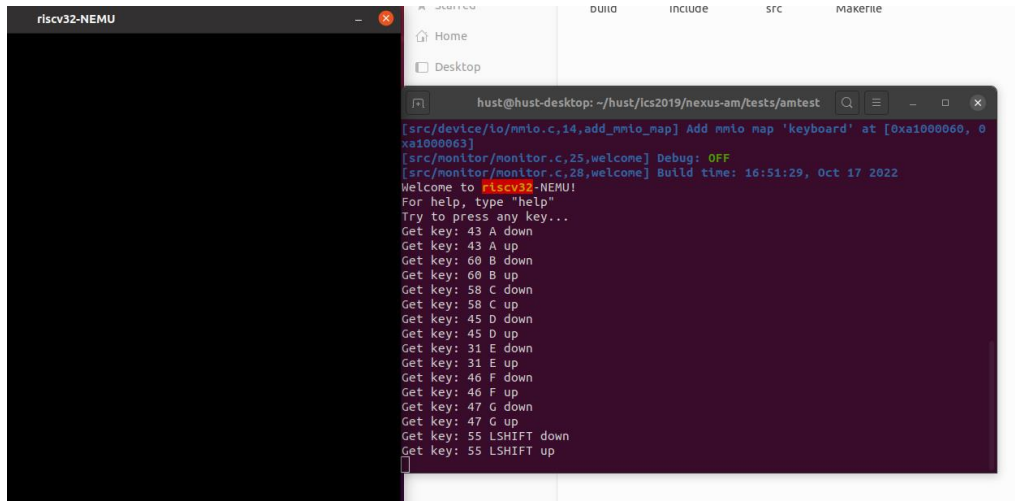


图 2.20 keyboard.c 程序测试

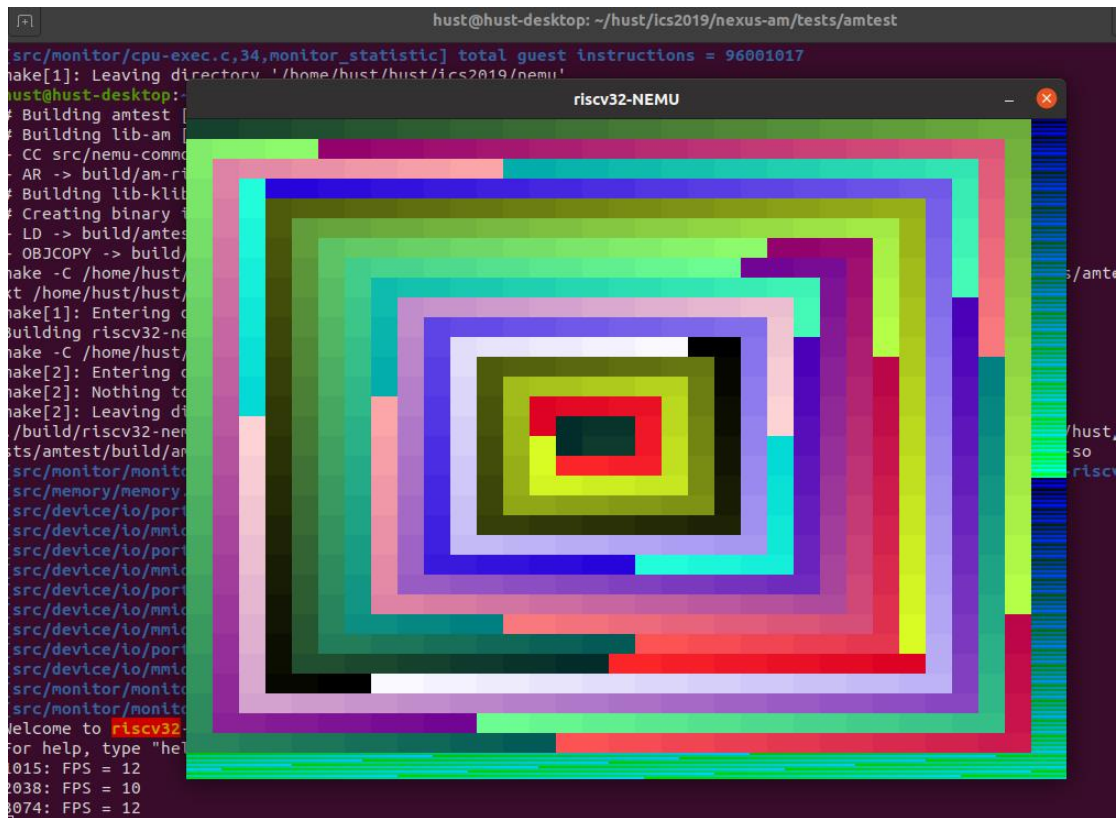


图 2.21 video.c 程序测试

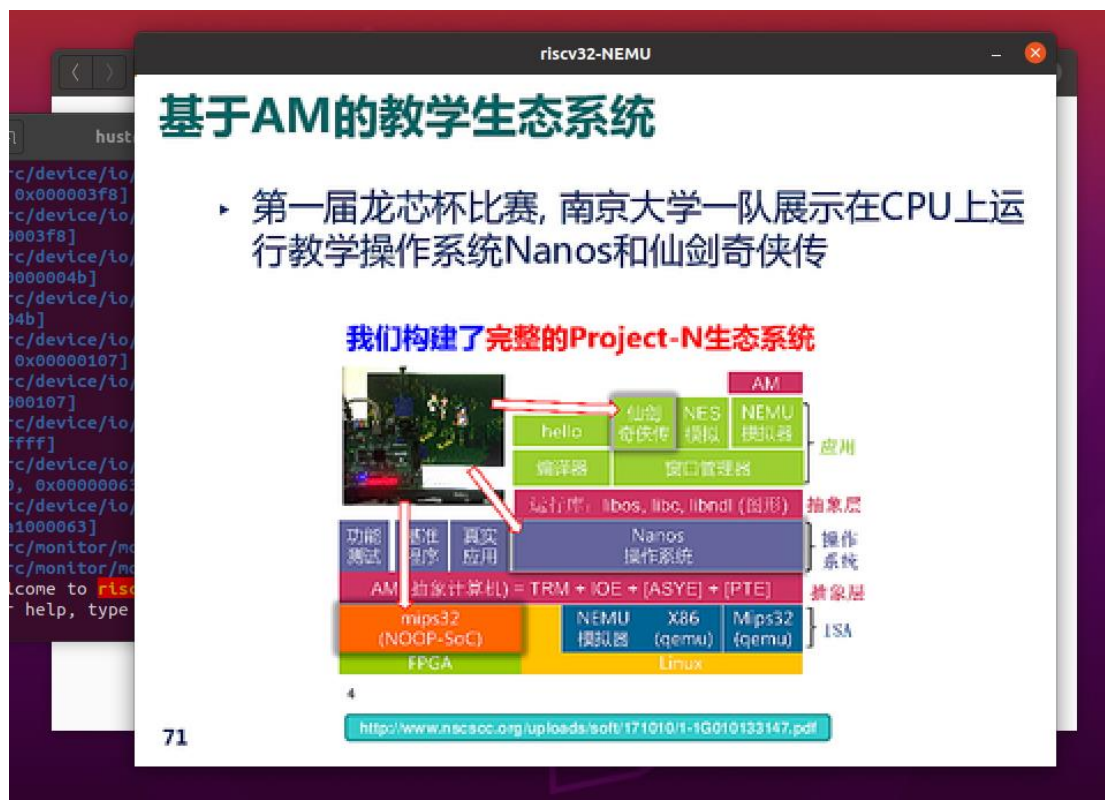


图 2.22 slider 测试

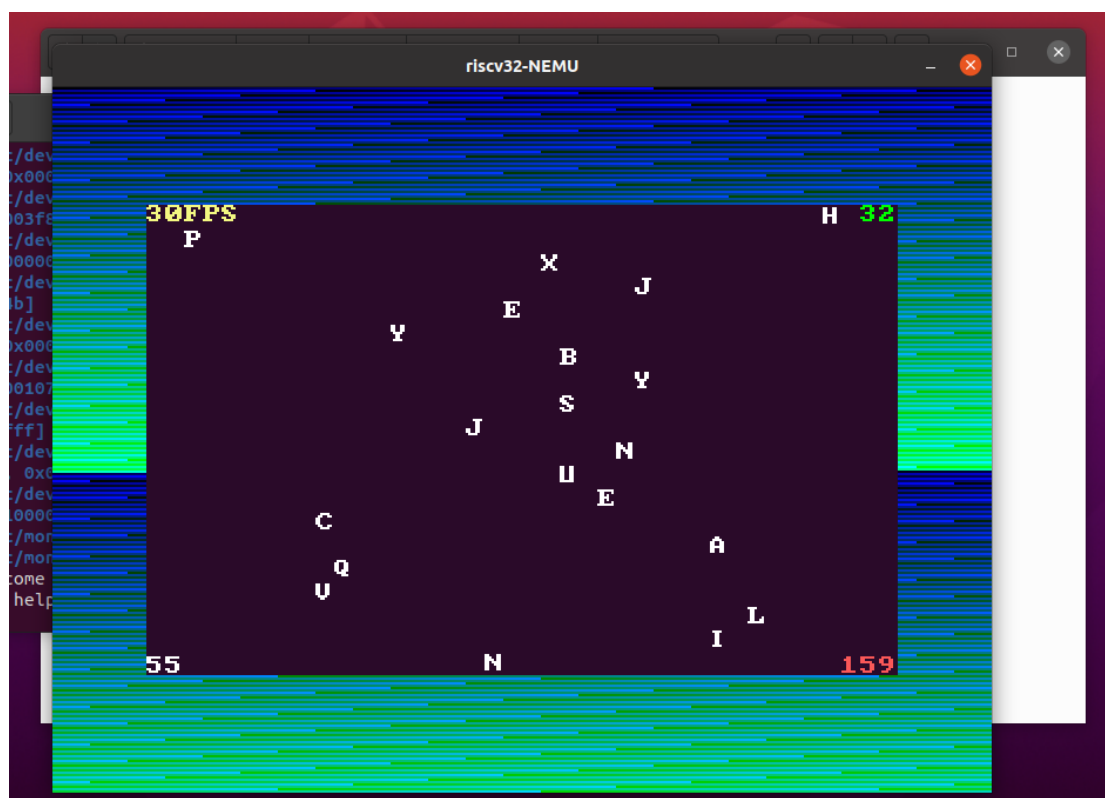


图 2.23 typing 测试

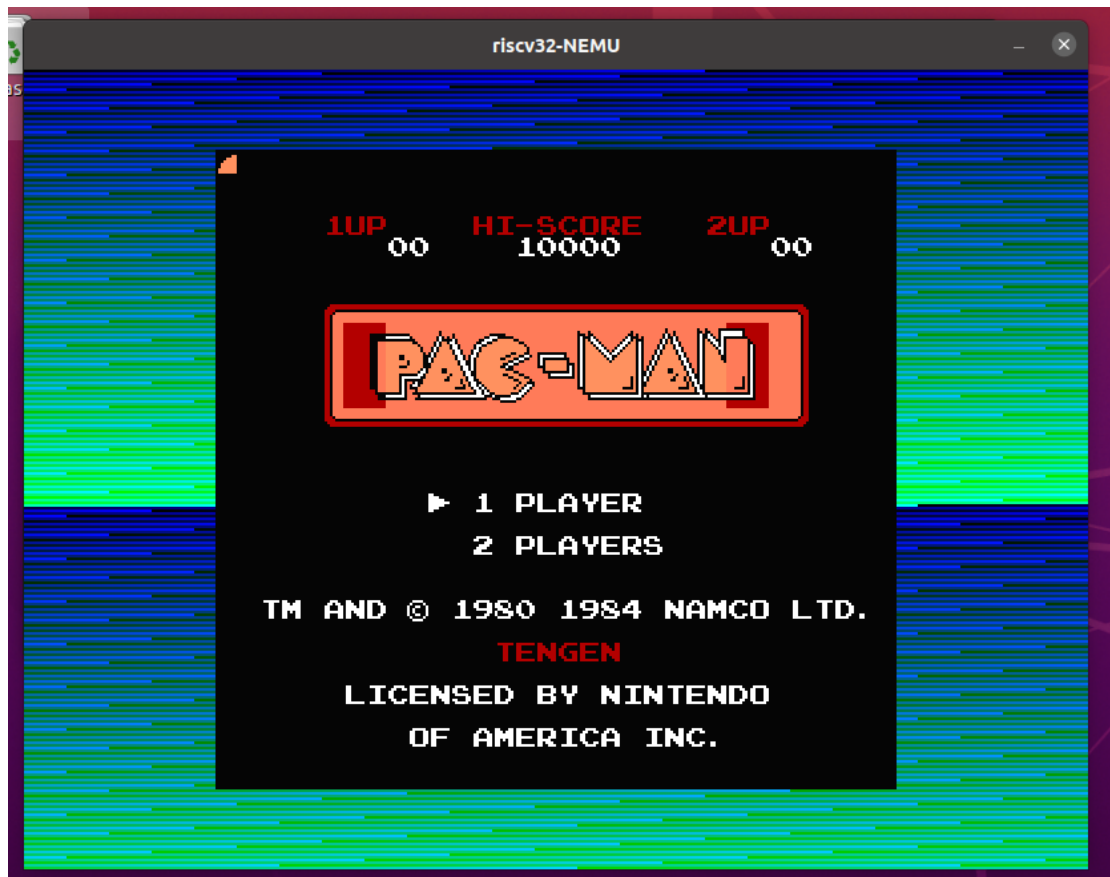


图 2.24 litenes 测试

2.2.4 PA2 必答题

□ 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, 也许对你有帮助
 - RTFM

图 2.25 PA2 必答题

- (1) 首先根据 PC 值通过 `instr_fetch` 函数取指令, 根据指令的 `opcode` 在 `opcode_table` 中进行索引找到对应的译码辅助函数和执行辅助函数, 然后通过译码辅助函数译码, 将译码得到的相关信息保存在 `decinfo` 结构体中, 接着通过执行辅助函数执行, 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, 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` 或 `inline` 不会报错, 但同时去掉时就会报错。都去掉报错的原因是在其他文件中有对 `rtl_li` 的定义, 因此会出现重复定义。单独去掉 `inline` 时, 其有 `static` 关键字, 函数会被限制在本文件内, 故不会出现重定义。单独去掉 `static`, 其有 `inline` 关键字, 函数在预编译时就会展开, 故不会出现重定义。
- (3)
 - 第一次添加, 使用 `grep` 命令查看, 有 81 个 `dummy`;
 - 第二次添加, 使用 `grep` 命令查看, 有 82 个 `dummy`;
 - 初始化后, 重新编译会报错, 因为两个 `dummy` 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 - 穿越时空的旅程: 批处理系统

PA3 的主要内容是实现系统调用和文件系统。

2.3.1 PA3.1 实现自陷操作_yield()及其过程

实验涉及修改的文件:

- nemu/src/isa/riscv32/exec/all-instr.h
- nemu/src/isa/riscv32/exec/system.c
- nemu/src/isa/riscv32/include/isa/reg.h
- nemu/src/isa/riscv32/exec/exec.c
- nemu/src/isa/riscv32/intr.c
- nexus-am/am/include/arch/riscv32-nemu.h
- nexus-am/am/src/riscv32/nemu/cte.c
- nanos-lite/include/common.h
- nanos-lite/src/irq.c

实验内容:

- (1) 声明执行辅助函数: 在 all-instr.h 中声明 make_EHelper(system);
- (2) 定义执行辅助函数: 在 system.c 定义 make_EHelper(system)函数, 该函数根据指令的 funct3 字段区分 ecall、sret、csrrw 和 csrrs 指令, 其中 ecall 和 sret 的 funct3 字段相同, 则需要根据立即数字段加以区分。对于 csrrw 和 csrrs 指令需要添加额外的寄存器来存储程序的状态, 故在 reg.h 中添加如下寄存器: stvec (存放异常入口地址), sepc (存放触发异常的 PC), sstatus (存放处理器的状态), scause (存放触发异常的原因)。对于 csrrw 和 csrrs 指令来说, 根据其 csr 字段的不同, 其具有不同的作用。在 system.c 文件中定义枚举类型 SSTATUS=0x100, STVEC=0x105, SEPC=0x141, SCAUSE=0x142, 分别对应 csr 字段不同时的情况。调用 rtl 函数完成指令的编写。
- (3) 完善 opcode_table: 在 exec.c 文件中为 opcode_table 表添加 IDEX(I, system)。
- (4) 执行更多初始化工作: 在 common.h 中定义#define HAS_CTE, 这会在使 main 函数执行更多的初始化工作。
- (5) 实现异常响应机制: 在 intr.c 文件中, 需要完成函数 void raise_intr(uint32_t NO, vaddr_t epc)来实现模拟异常响应机制, 其内容是将触发异常的 PC(epc)存入 CPU 的 sepc 寄存器中, 并将异常号存进 scause 寄存器中, 然后使用 rtl_j 函数跳转到存放异常入口地址处理程序 (stvec)。这个函数会在 system.c 中的 ecall 指令会用到。
- (6) 重新组织_Context 结构体: 查看 nexus-am/am/src/\$ISA/nemu/trap.S 的汇编指令, 将_Context 结构体中的成员调整位置为 uintptr_t gpr[32], cause, status, epc。
- (7) 处理异常号: 补充 cte.c 文件中的_Context* __am_irq_handle(_Context*c) 函数, 这个函数根据中断异常号将事件设置为_EVENT_YIELD (自陷事件)、_EVENT_SYSCALL (正常系统调用事件)和_EVENT_ERROR (未实现系统调用事件)。为了实现自陷事件, 当遇到中断异常号为-1 的情况时, 将事件标为_EVENT_YIELD。

- (8) 处理事件：补充 `irq.c` 文件中的 `static _Context * do_event (_Event e , _Context * c)` 函数。根据 `e` 事件的事件号来处理不同的事件，事件总共分为三种事件 `_EVENT_YIELD`、`_EVENT_SYSCALL` 和 `_EVENT_ERROR`。若要实现自陷事件，则需要处理 `_EVENT_YIELD`，使用 `LOG` 函数输出对应的信息即可。

实验结果：

```
[/home/hust/hust/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/hust/ics2019/nanos-lite/src/main.c,15,main] Build time: 15:40:37, Oct 18 2022
[/home/hust/hust/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = , end = , size = -./,*,...*( bytes
[/home/hust/hust/ics2019/nanos-lite/src/device.c,81,init_device] Initializing devices...
[/home/hust/hust/ics2019/nanos-lite/src/irq.c,30,init_irq] Initializing interrupt/exception handler...
[/home/hust/hust/ics2019/nanos-lite/src/proc.c,31,init_proc] Initializing processes...
[/home/hust/hust/ics2019/nanos-lite/src/main.c,33,main] Finish initialization
[/home/hust/hust/ics2019/nanos-lite/src/irq.c,14,do_event] self trapping event
[/home/hust/hust/ics2019/nanos-lite/src/main.c,39,main] system panic: Should not reach here
```

图 2.26 自陷事件

2.3.2 PA3.2 实现用户程序的加载和系统调用，支撑 TRM 程序的运行

实验涉及修改的文件：

- `nanos-lite/src/loader.c`
- `nanos-lite/src/syscall.c`
- `nanos-lite/src/irq.c`
- `nanos-lite/src/proc.c`
- `nexus-am/am/src/riscv32/nemu/cte.c`
- `navy-apps/libs/libos/src/nanos.c`
- `nexus-am/am/include/arch/riscv32-nemu.h`

实验内容：

- (1) 实现 loader 文件：在 `loader.c` 文件中定义了 `static uintptr_t loader(PCB *pcb, const char *filename)` 函数，需要完善补充此函数。此函数的作用是把用户程序加载到正确的内存位置，然后执行用户程序。需要读取 `ramdisk` 文件中的数据，则需要调用 `size_t ramdisk_read(void*, size_t, size_t)` 函数，根据 `Elf_Ehdr` 和 `Elf_Phdr` 宏的内容，读取数据。在 `proc.c` 文件中的 `void init_proc()` 函数中添加语句 `naive_uload(NULL, NULL)` 就会读取 `dummy` 程序，此时会引起 1 号异常处理事件。
- (2) 识别系统调用：在 `nanos.c` 中有待完成的系统调用函数，完成需要的系统调用函数，实际上就是调用 `intptr_t _syscall(intptr_t type, intptr_t a0, intptr_t a1, intptr_t a2)` 函数，然后系统调用函数返回该函数的返回值。系统调用号在 `navy-apps/libs/libos/src/syscall.h` 定义，根据自己的需要实现系统调用函数。本次实现的系统调用号有 `SYS_exit`。然后在 `cte.c` 文件中的 `__am_irq_handle(_Context *c)` 函数中添加 `case SYS_exit: ev.event = _EVENT_SYSCALL; break;` 代码。接下来在 `irq.c` 文件中的 `do_event(_Event e, _Context *c)` 函数中添加 `case _EVENT_SYSCALL: do_syscall(c); break;` 使得 `do_event` 函数能够识别 `_EVENT_SYSCALL`，并将此事件交给 `do_syscall` 函数处理。
- (3) 实现 `SYS_yield` 系统调用：在 `syscall.c` 文件中，`do_syscall(_Context *c)` 用来处理系统调用，完善该函数。首先需要实现 `riscv32-nemu.h` 文件下的

GPR?宏, 实现正确为#define GPR2 gpr[10]、#define GPR3 gpr[11]、#define GPR4 gpr[12]、#define GPRx gpr[10], 回到 do_syscall 函数, 使数组 a 的元素指向正确的 GPR?寄存器, a[1] = c->GPR2、a[2] = c->GPR3、a[3] = c->GPR4。a[0]为系统调用号, 通过 switch 语句对不同的系统调用号进行处理, 本次处理两个系统调用号, 分别为 SYS_yield 和 SYS_exit。SYS_yield 情况下, 调用 _yield()函数实现自陷。SYS_exit 情况下, 调用 _exit()函数结束程序, 但是这里不这样做, 直接调用 _halt()函数结束程序。两个系统调用的返回值都设为 0, 存入 GPRx 中。

- (4) 在 Nanos-lite 上运行 Hello world: 输出是通过 SYS_write 系统调用来实现, 和上述流程一样, 先在 nanos.c 文件中实现 SYS_write 系统调用, 然后在 cte.c 文件中识别 SYS_write 系统调用, 将这个事件标为 _EVENT_SYSCALL 事件, 然后在 syscall.c 文件中处理系统调用, 即调用 ramdisk_write 函数实现对 ramdisk 的读写。将 nanos-lite/Makefile 中 SINGLE_APP = \$(NAVY_HOME)/tests/dummy 改为 SINGLE_APP = \$(NAVY_HOME)/tests/hello, 再次编译运行程序即可看到连续输出的 Hello world 字符串。
- (5) 实现堆区管理: 在 nanos.c 文件中, 编写 void *_sbrk(intptr_t increment)函数。根据讲义按照其工作方式编写即可, 后续流程不一一介绍, 在 do_syscall 函数中, 处理该系统调用时使 GPRx 为 0, 即堆区分配总是成功, 因为 PA3 不需要用到这个功能。

实验结果:

```

Welcome to riscv32-NEMU!
For help, type "help"
[/home/hust/hust/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/hust/ics2019/nanos-lite/src/main.c,15,main] Build time: 17:27:27, Oct 18 2022
[/home/hust/hust/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = , end = , size = -./,*,-0-. bytes
[/home/hust/hust/ics2019/nanos-lite/src/device.c,81,init_device] Initializing devices...
[/home/hust/hust/ics2019/nanos-lite/src/irq.c,30,init_irq] Initializing interrupt/exception handler...
[/home/hust/hust/ics2019/nanos-lite/src/proc.c,31,init_proc] Initializing processes...
[/home/hust/hust/ics2019/nanos-lite/src/loader.c,52,naive_uoload] Jump to entry = 0x83000c8
[/home/hust/hust/ics2019/nanos-lite/src/irq.c,14,do_event] self trapping event
nemu: HIT GOOD TRAP at pc = 0x80100cc0

```

图 2.27 dummy 程序测试

```

Welcome to riscv32-NEMU!
For help, type "help"
[/home/hust/hust/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/hust/ics2019/nanos-lite/src/main.c,15,main] Build time: 17:37:43, Oct 18 2022
[/home/hust/hust/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = , end = , size = -./,*,-0-. bytes
[/home/hust/hust/ics2019/nanos-lite/src/device.c,81,init_device] Initializing devices...
[/home/hust/hust/ics2019/nanos-lite/src/irq.c,30,init_irq] Initializing interrupt/exception handler...
[/home/hust/hust/ics2019/nanos-lite/src/proc.c,31,init_proc] Initializing processes...
[/home/hust/hust/ics2019/nanos-lite/src/loader.c,52,naive_uoload] Jump to entry = 0x83000120
Hello World!
Hello World from Navy-apps for the 2th time!
Hello World from Navy-apps for the 3th time!
Hello World from Navy-apps for the 4th time!
Hello World from Navy-apps for the 5th time!
Hello World from Navy-apps for the 6th time!
Hello World from Navy-apps for the 7th time!
Hello World from Navy-apps for the 8th time!
Hello World from Navy-apps for the 9th time!
Hello World from Navy-apps for the 10th time!
Hello World from Navy-apps for the 11th time!
Hello World from Navy-apps for the 12th time!
Hello World from Navy-apps for the 13th time!
Hello World from Navy-apps for the 14th time!
Hello World from Navy-apps for the 15th time!

```

图 2.28 hello 程序测试

2.3.3 PA3.3 运行仙剑奇侠传并展示批处理系统，提交完整的实验报告

实验涉及修改的文件：

- nanos-lite/src/loader.c
- nanos-lite/src/syscall.c
- nanos-lite/src/proc.c
- nexus-am/am/src/riscv32/nemu/cte.c
- navy-apps/libs/libos/src/nanos.c
- nanos-lite/Makefile
- nanos-lite/src/fs.c
- nanos-lite/src/device.c
- navy-apps/libs/libos/src/nanos.c
- nexus-am/am/src/riscv32/nemu/cte.c
- nanos-lite/src/syscall.c

实验内容：

- (1) 修改 Makefile 文件：update: update-ramdisk-single src/syscall.h 修改为 update: update-ramdisk-fsimg src/syscall.h。然后 make clean 将之前运行生成的 ramdisk.img 清除，重新 make 生成新的 ramdisk.img。
- (2) 文件记录表 Finfo：在 fs.c 文件中的 Finfo 结构体添加 open_offset 成员，用来记录文件被打开之后的读写指针。
- (3) 实现基本文件操作：基本的文件操作有 fs_open、fs_read、fs_write、fs_lseek 和 fs_close 这五种操作。在终端使用 man open 可以查看相关的 manual page 来帮助实现 open 命令，其余命令也一样。不过要注意的是，实现的是简易文件系统，若 fs_open 没有找到文件的话需要用断言 assert 结束程序运行。使用 ramdisk_read() 函数和 ramdisk_write() 函数实现对 ramdisk.img 的读写，不过在 Finfo 结构体中存有不同读写函数的指针，故在使用 ramdisk_read() 函数和 ramdisk_write() 函数之前，先判断对应读写指针是否存在，若存在则调用对应读写指针进行读写。由于文件大小是固定的，需要注意偏移量不要越过文件的边界，故当偏移量超过文件边界时需要将其长度限制在文件边界处。对于 fs_write 函数中的 stdout 和 stderr 需要使用 _putc() 函数进行输出，其余的 fs_xxx 函数则直接忽略 stdin, stdout 和 stderr 这三个特殊文件操作。
- (4) 为文件系统添加系统调用：这次需要添加的系统调用有 SYS_open、SYS_read、SYS_close、SYS_lseek，添加过程参考 PA3.2(2)(4)。
- (5) 让 loader 使用文件：之前是让 loader.c 文件中的 loader(PCB *pcb, const char *filename) 函数直接调用 ramdisk_read() 来加载用户程序，但是程序多了之后就不好管理了。故调用上述实现的 fs_xxx 函数实现 loader() 函数。修改完 loader() 函数后就可以在 proc.c 文件中的 init_proc() 函数中调用 naive_upload(PCB *pcb, const char *filename) 函数来打开程序了，运行不同的程序只需要修改 filename 即可。
- (6) 实现完整的文件系统：fs_xxx 函数上述已经实现，修改 naive_upload(NULL, "/bin/text") 再次运行即可看到终端输出 PASS!!! 的信息。
- (7) 把串口抽象成文件：在 device.c 实现 serial_write(const void *buf, size_t offset, size_t len) 函数，使用 _putc() 函数将 buf 的信息一个一个字符输出

即可。修改 fs.c 文件中的 file_table 表，将 stdout 和 stderr 的 write 指针修改为 (WriteFn)serial_write。修改 fs_write() 函数，将 fd 为 1 和 2 的情况调用该文件对象的 write 指针指向的函数进行输出，即实际上是调用 serial_write() 函数输出。

- (8) 把设备输入抽象成文件：Nanos-lite 和 Navy-apps 约定将输入设备（时钟和键盘）抽象成文件 /dev/events。在 device.c 实现 events_read(void *buf, size_t offset, size_t len) 函数，该函数将键盘和时间的信息写进 buf 中。调用 read_key() 函数读取键盘信息存在 key 变量中，若 key & KEYDOWN_MASK 不等于 0，表示键盘按下，其中 KEYDOWN_MASK=0x8000 是键盘掩码。首先判断键盘是否按下，然后将信息写入到 buf 中，假如没有键盘活动，就将时间信息写入 buf 中。在 fs.c 文件中填写 file_table 表，将 {"/dev/events", 0, 0, 0, (ReadFn)events_read, (WriteFn)invalid_write} 填入表格中。然后修改 init_proc() 函数中的 naive_upload(NULL, "/bin/events")，编译运行即可看到程序输出时间和按键事件。
- (9) 把 VGA 显存抽象成文件：Nanos-lite 和 Navy-apps 约定把显存抽象成文件 /dev/fb，在 device.c 实现 fb_write(const void *buf, size_t offset, size_t len)，其中坐标的计算方式为偏移量除以或模屏幕宽度分别得到 y 和 x 的坐标，调用 draw_rect() 函数绘制，返回长度 len。然后在 fs.c 文件中填写 file_table 表，将 {"/dev/fb", 0, 0, 0, (ReadFn)invalid_read, (WriteFn)fb_write} 填入表格中。Nanos-lite 和 Navy-apps 约定把刷新操作通过写入设备文件 /dev/fbsync 来触发，在 device.c 实现 fbsync_write(const void *buf, size_t offset, size_t len) 函数，直接调用 draw_sync() 函数即可，返回值为 len，在 fs.c 文件中将 {"/dev/fbsync", 0, 0, 0, (ReadFn)invalid_read, (WriteFn)fbsync_write} 填入 file_table 表中。Nanos-lite 和 Navy-apps 约定屏幕大小的信息通过 /proc/dispinfo 文件来获得，在 device.c 实现 dispinfo_read(void *buf, size_t offset, size_t len) 函数，在 init_device() 函数中，将屏幕的宽和高写进 dispinfo 数组中，然后在 dispinfo_read() 函数中将 dispinfo 数组写进 buf 中，在 fs.c 文件中将 {"/proc/dispinfo", 128, 0, 0, (ReadFn)dispinfo_read, (WriteFn)invalid_write} 填入 file_table 表中。然后修改 init_proc() 函数中的 naive_upload(NULL, "/bin/bmptest")，再次编译运行可以看到 logo N。
- (10) 在 NEMU 中运行仙剑奇侠传：从链接 <https://course.cunok.cn/pa/pal.tbz> 下载仙剑奇侠传的数据文件，使用命令 tar -jxvf pal.tbz 解压压缩包，并将 pal 文件夹拷贝到 navy-apps/fsimg/share/games/ 目录下，然后修改 naive_upload(NULL, "/bin/pal")，重新编译运行即可进入仙剑奇侠传游戏中。
- (11) 展示你的批处理系统：实现批处理系统还需要实现 SYS_execve 系统调用，在 nanos.c 文件中实现即可，然后在 cte.c 识别该系统调用，然后在 syscall.c 中处理该系统调用，在 SYS_execve 处调用 naive_upload(NULL, a[0]) 加载菜单选择的程序，在 SYS_exit 处调用 naive_upload(NULL, "/bin/init") 重新加载开始菜单。将 {"/dev/tty", 0, 0, 0, (ReadFn)invalid_read, (WriteFn)serial_write} 添加到 fs.c 文件中的 file_table 表中，然后在 proc.c 文件中修改 naive_upload(NULL, "/bin/init")。重新编译运行，即可看到开始菜单，共有 8 个程序可选择，输入对应按钮则会打开对应的程序。

实验结果:

```
Welcome to riscv32-NEMU!
For help, type "help"
[/home/hust/hust/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/hust/ics2019/nanos-lite/src/main.c,15,main] Build time: 18:27:08, Oct 18 2022
[/home/hust/hust/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = , end = , size = -.
/,*,...)-. bytes
[/home/hust/hust/ics2019/nanos-lite/src/device.c,81,init_device] Initializing devices...
[/home/hust/hust/ics2019/nanos-lite/src/irq.c,30,init_irq] Initializing interrupt/exception handler...
[/home/hust/hust/ics2019/nanos-lite/src/proc.c,31,init_proc] Initializing processes...
[/home/hust/hust/ics2019/nanos-lite/src/loader.c,52,naive_uoload] Jump to entry = 0x830002f4
PASS!!!
nemu: HIT GOOD TRAP at pc = 0x80100db4
```

图 2.29 text 程序测试

```
Welcome to riscv32-NEMU!
For help, type "help"
[/home/hust/hust/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/hust/ics2019/nanos-lite/src/main.c,15,main] Build time: 19:12:07, Oct 18 2022
[/home/hust/hust/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = , end = , size = -./*,...)) bytes
[/home/hust/hust/ics2019/nanos-lite/src/device.c,82,init_device] Initializing devices...
[/home/hust/hust/ics2019/nanos-lite/src/irq.c,30,init_irq] Initializing interrupt/exception handler...
[/home/hust/hust/ics2019/nanos-lite/src/proc.c,31,init_proc] Initializing processes...
[/home/hust/hust/ics2019/nanos-lite/src/loader.c,52,naive_uoload] Jump to entry = 0x83000180
Start to receive events...
receive event: kd A
receive event: kd D
receive event: ku D
receive event: kd A
receive event: ku A
receive event: kd D
receive event: ku D
receive time event for the 1024th time: t 55
receive time event for the 2048th time: t 91
receive time event for the 3072th time: t 133
receive time event for the 4096th time: t 172
receive event: kd D
receive time event for the 5120th time: t 212
receive event: kd A
receive event: ku D
receive time event for the 6144th time: t 269
receive time event for the 7168th time: t 307
receive event: ku A
```

图 2.30 events 程序测试

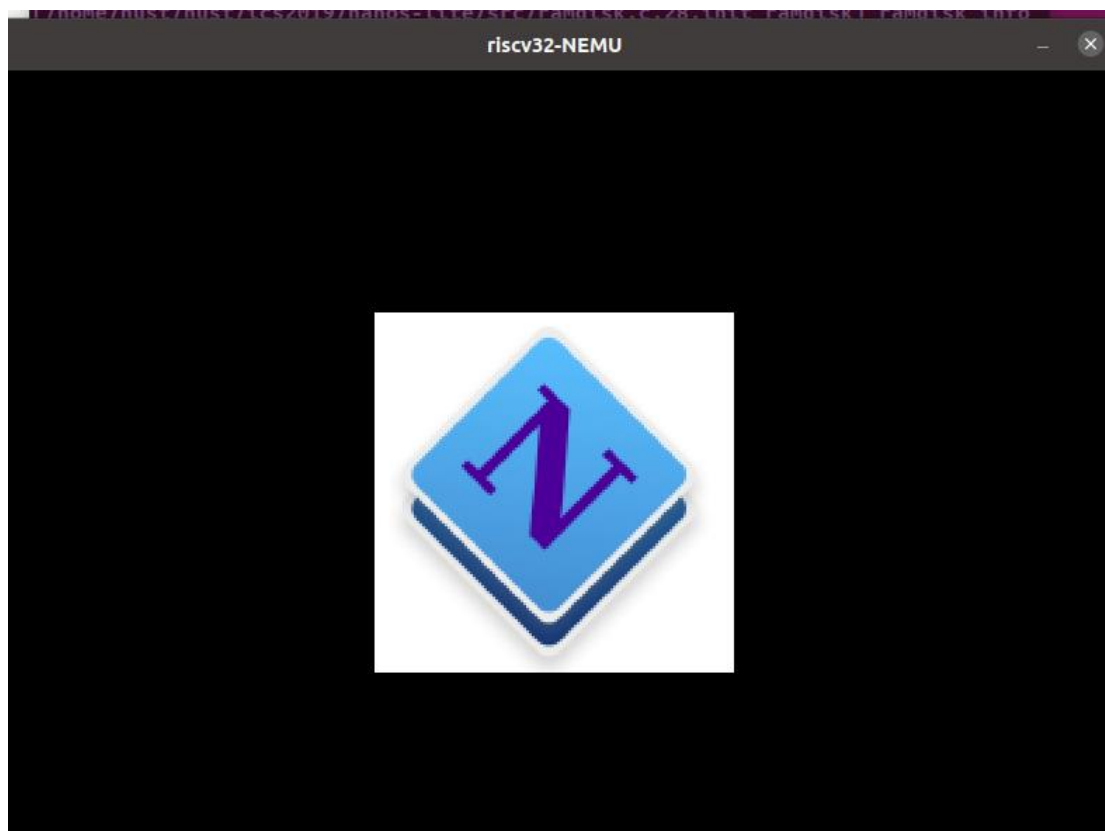


图 2.31 bmptest 程序测试

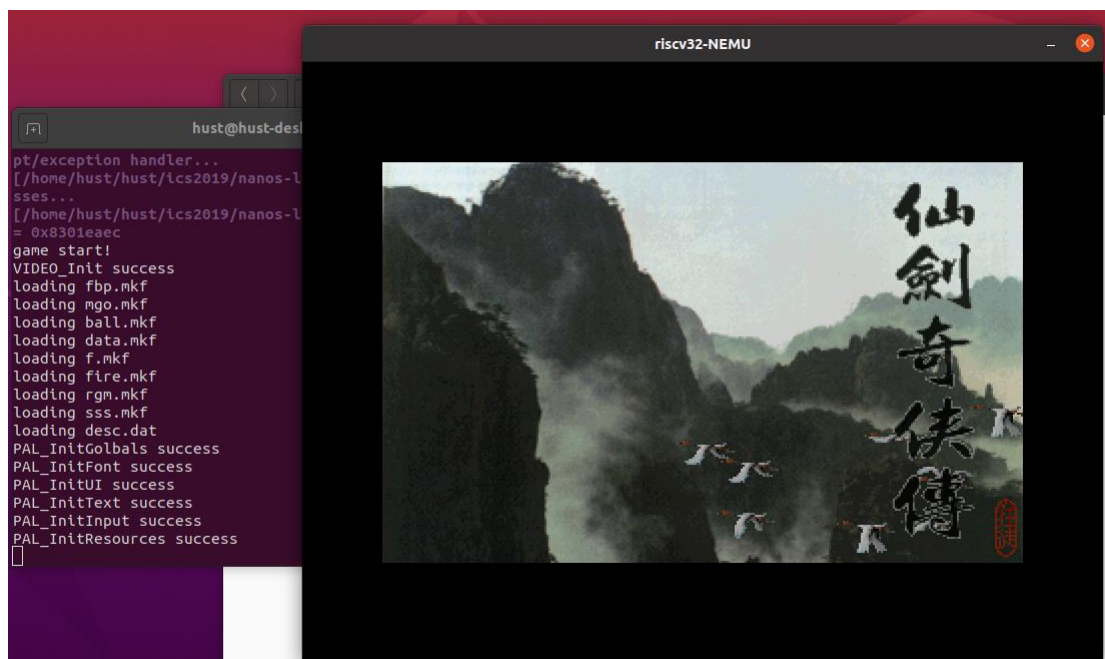


图 2.32 仙剑奇侠传游戏初始化画面



图 2.33 仙剑奇侠传游戏对话画面



图 2.34 仙剑奇侠传菜单功能

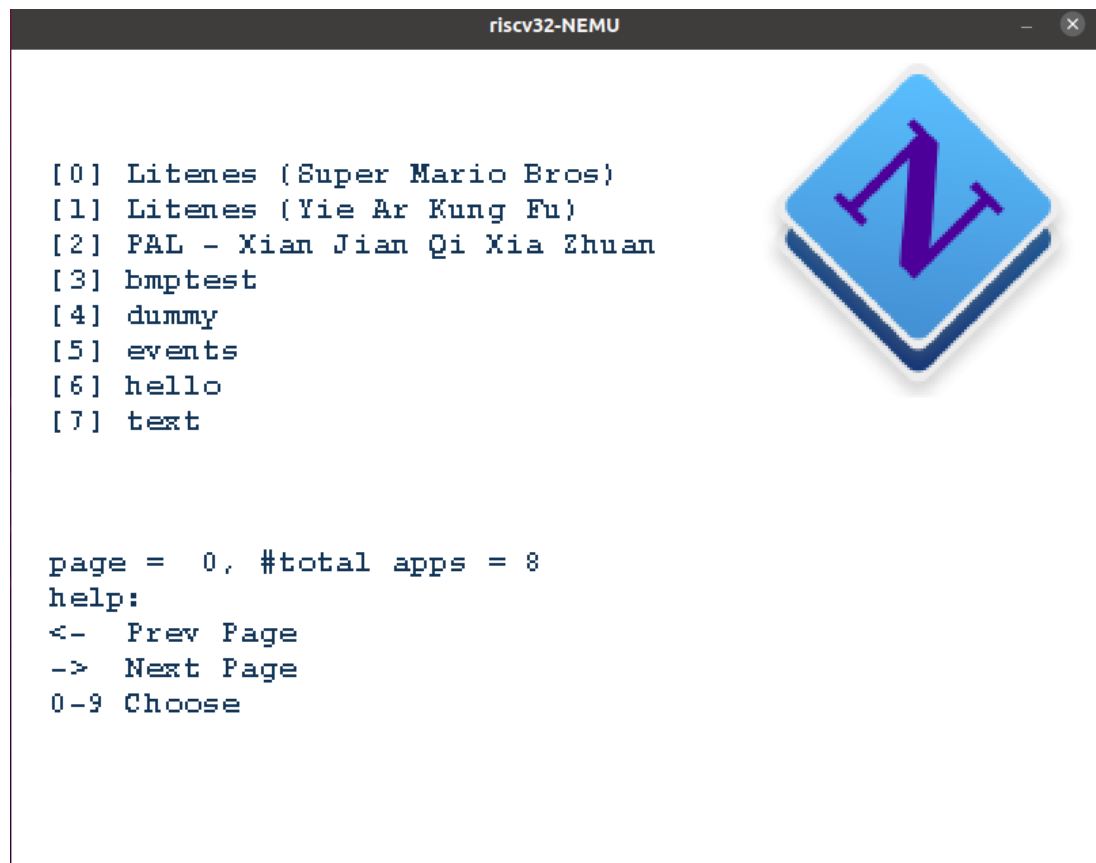


图 2.35 开始菜单

2.3.4 PA3 必答题

□ 必答题(需要在实验报告中回答) - 理解上下文结构体的前世今生

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。

图 2.36 必答题 1

(1) 理解上下文的前世今生

- `c` 指向的上下文结构在 `nexus-am/am/include/arch/riscv32-nemu.h`。
- 由 `trap.S` 得出。
- 在 `trap.S` 中 `_Context` 结构体的成员会被先后压栈，成员赋值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 `_Context` 上下文结构体，确定了成员的顺序；`trap.S` 通过汇编程序对上下文结构体进行赋值；讲义则说明了流程；实现的指令 `ecall`、`sret`、`csrrw`、和 `csrrs` 实现了异常调用，异常返回等，使得自陷操作能够正常执行。

□ 必答题(需要在实验报告中回答) - 理解穿越时空的旅程

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。

图 2.37 理解穿越时空的旅程

(2) 理解穿越时空的旅程

- 在文件 `nexus-am/am/src/riscv32/nemu/cte.c` 中，通过内联汇编代码 `li a7, -1`（即将 `a7` 设为-1），然后执行 `ecall` 指令。
- 在文件 `nemu/src/isa/riscv/exec/system.c` 中，`ecall` 调用了 `raise_intr()` 函数，参数为中断编号和当前的 `pc` 值。
- 在文件 `nemu/src/isa/riscv32/intr.c` 中定义了 `raise_intr()` 函数，该函数将异常地址和中断号分别保存在 `sepc` 和 `scause` 寄存器中，调用 `rtl_j()` 函数跳转到 `stvec` 寄存器所存中断处理程序入口地址处。
- 在文件 `nexus-am/am/src/riscv32/nemu/cte.c` 中的 `_cte_init()` 函数中中断处理程序为 `__am_asm_trap()` 函数，`raise_intr()` 函数将虚拟机内部正在运行的程序转移到了其中断服务程序处继续执行。
- `__am_asm_trap()` 函数以汇编代码的形式定义在 `trap.S` 中，在进行一系列压栈操作后，转移到函数 `__am_irq_handle()` 函数处执行。
- 随后进行事件分发，在文件 `cte.c` 中 `am_irq_handle()` 函数根据栈的内容

的 `cause`（中断号）对事件进行打包，调用 `user_handler()` 函数对事件进行处理。`user_handler()` 函数在 `_cte_init()` 函数中已经进行了初始化。

- 在文件 `nanos-lite/src/irq.c` 中，`do_event()` 函数对传入的事件进行解析。
- `do_event()` 函数结束就会回到 `trap.S` 汇编代码，恢复上下文调用 `sret` 指令。
- `sret` 指令会调用 `nemu` 中对应的辅助执行函数，从 `sepc` 中读出之前保存的 `pc` 然后加 4，然后跳转到下一指令的地址。
- 自陷操作到此全部完成。

□ 必答题(需要在实验报告中回答) - hello程序是什么, 它从何而来, 要到哪里去

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!

图 2.38 hello 程序是什么, 它从何而来, 要到哪里去

(3) hello 程序是什么, 它从何而来, 要到哪里去

- `hello` 程序在磁盘上, `hello.c` 被编译成 `ELF` 文件后, 位于 `ramdisk` 中。通过 `load()` 函数读入指定的内存并放在正确的位置。加载完成后, 操作系统从其 `ELF` 信息中获取到程序入口地址, 通过上下文切换从入口地址处继续执行, `hello` 程序便获取到 `CPU` 的控制权开始执行指令。
- 对于字符串在终端的显示, 首先调用 `printf` 等库函数, 然后通过 `SYS_write` 系统调用来输出字符, 系统调用通过调用外设的驱动程序最终将内容在外设中表现出来, 程序执行完毕后操作系统会回收其内存空间。

□ 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?

图 2.39 理解计算机系统

(4) 理解计算机系统

- 操作系统通过库函数读出画面的像素信息, 画面通过被抽象成设备文件的 `VGA` 输出, `fs_wrtie()` 函数在逐步执行中调了 `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 总结与心得

PA 是一项很有趣、很有挑战性的项目。它涉及的内容知识点很多，能够丰富学生的视野，提升学生的系统开发能力，能让学生从系统的角度思考问题。我很庆幸自己选了这门课作为系统综合能力培养课程的项目。经过这个项目，我有如下心得和体会。

3.1 PA1

PA1 的核心内容是实现一个简易调试器，是 NEMU 中一项非常重要的基础设施。实现简易调试器可以提高调试的效率，同时也可以熟悉框架代码，为之后的挑战做好铺垫。简易调试器的主要内容为 GDB 的子集，其有如下功能：帮助指令 `help`；继续运行指令 `c`；退出 `q`；单步执行指令 `si`；打印程序状态 `info`；表达式求值 `p`；扫描内存 `x`；设置监视点 `w`；删除监视点 `d`。对于本节实验来说，我认为表达式求值是重难点。

- 难点一：tokens 的识别，即编写相应的正则表达式识别 token。
- 难点二：*运算符的识别，*具有乘法和解引用两功能，首先遍历一遍所有 token，然后将原本识别为乘法的 token 标记为解引用类型。解引用的识别在讲义上有涉及。
- 难点三：-运算符的识别，即负数还是减号的识别，识别流程和难点二一样。
- 难点四：求值运算函数 `eval()` 的编写。

监视点功能也是需要注意的模块，实现监视点功能，需要熟悉链表的结构，由于监视点是存在某个数组中的，所以要注意数组空间的余量。

3.2 PA2

PA2 的核心内容是实现 `riscv` 指令和 `klib` 库函数。整个 PA2 的开发流程都离不开指令的设计，所以一定要深刻理解指令的结构，所以在开始实验前，我建议先看看相关的书籍，然后好好理解以下 `nemu` 目录下的框架，最后才着手开始编写代码。对于本实验，我推荐先完成 `diff-test`，再开始指令的设计是一个不错的选择。对于本实验的重难点有：

- 难点一：涉及的指令有很多，需要一边查指令集一边实现架构下的指令。本次实验的 `cputest` 目录下有很多测试程序，需要实现足够多的指令来通过这些测试。未实现的指令可以通过某些程序的反汇编代码来得到。
- 难点二：设计指令涉及译码辅助函数和执行辅助函数，需要深刻理解这两种类型的函数。
- 难点三：`kib` 库的实现，需要我们实现 `stdio.c` 和 `string.c` 库，可以使用“`man+函数名`”参考标准库函数的实现。
- 难点四：输入输出设备，主要是 VGA 设备的设计。

总的来说，PA2 是比较困难得一个环节，需要我们多花时间阅读相关书籍和多花时间阅读框架代码，只有理解了框架代码才会顺利得完成内容。

3.3 PA3

PA3的核心内容是实现系统调用和文件系统。本节内容还是比较困难的，主要是涉及到了汇编、联合编译以及 elf 文件格式等内容。本节的重难点如下：

- 难点一：添加自陷指令，并添加相应的 CSR 寄存器来存方异常状态和处理器状态，理解自陷指令工作的流程。
- 难点二：理解系统调用的流程；
- 难点三：loader 函数的编写，即了解 elf 文件的格式，使该函数能够加载文件。
- 难点四：文件系统的实现，linux 系统下万物为文件，本次实现的模拟器也是如此，将设备等抽象为文件，通过不同的读写函数实现对设备文件的读写。

对于 PA3 来说，编写的代码量还是比较少的，最主要的还是需要熟悉框架代码，只有熟悉了框架代码，在做实验的时候才会如鱼得水。

4 电子签名

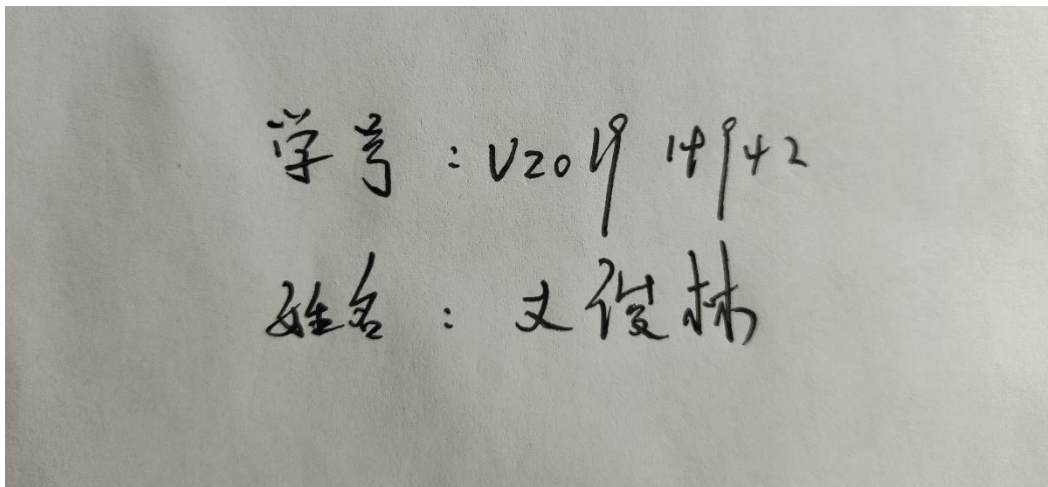


图 4.1 电子签名