

# 华中科技大学

## 2024

### 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:	CS2001 班
学 号:	U202015315
姓 名:	钟健维
电 话:	15272865964
邮 件:	1556109344@qq.com
完成日期:	2024-1-14



## 目 录

1 课程实验概述 .....	1
2 实验方案设计 .....	2
3 实验结果与结果分析 .....	22
4 电子签名 .....	23

# 1 课程实验概述

## 1.1 课设目的

从零开始实现一个完整的计算机系统，理解程序如何在计算机上运行，实现一个 riscv32 架构模拟器 NEMU，最终能在 NEMU 上运行游戏“仙剑奇侠传”。

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

## 1.3 实验环境

CPU 架构: x64 + windows + VirtualBox + Ubuntu64 + 操作系统: gnu/linux  
编译器: gcc、riscv-none-embed-gcc + 编程语言: c 语言

## 2 实验方案设计

### 2.1 PA1 - 开天辟地的篇章: 最简单的计算机

PA1 的目标是实现一个简易的调试器 (monitor)，包括以下功能

```
(nemu) help
help - Display informations about all supported commands
c     - Continue the execution of the program
q     - Exit NEMU
si    - Single step execution ( si [N] )
info  - Print information of registers or watchpoints ( i
nfo r || w )
p     - Evaluate expression ( p expr )
x     - Scan memory ( x N expr )
w     - set watchpoint ( w expr )
d     - delete watchpoint ( d N )
```

图 2.1.1 简易调试器的功能及参数

#### 2.1.1 PA1.1 实现单步执行，打印寄存器状态，扫描内存

##### 实现过程

1. 单步执行：在 nemu/src/monitor/debug/ui.c 中定义单步执行函数 static int cmd\_si(char\*args)，第一次调用 strtok 函数时需要传入字符串和 delim 符，找到分割符后替换为 0 并返回指向开头的指针，然后继续调用 strtok 解析该字符
2. 打印寄存器：通过 info r 来实现，在 ui.c 中定义 cmd\_info (char \* args) 函数，通过传入的 args 值打印寄存器状态，并调用 isa\_reg\_display ( ) 函数实现寄存器打印
3. 扫描内存：在 memory 相关的代码中找到 paddr\_read ( ) 函数，处理输入参数即可

##### 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 2.1.2 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

```

图 2.1.3 打印寄存器（仅展示部分寄存器内容）

```

t5  = 0x00000000
t6  = 0x00000000
(nemu) x 3 $pc
80100008
0x80100008 : 0x0002a503
0x8010000c : 0x0000006b
0x80100010 : 0x00000000
(nemu) █

```

图 2.1.4 扫描内存

## 2.1.2 PA1.2 实现算术表达式求值

### 实验过程：

1. 在算术表达式中各种 token 类型中添加规则，在识别出 token 后将 token 信息依次记录到 tokens 数组中（注意先后顺序的不同还有转义符号的使用）
2. 递归求值：表达式行为都是从 expr 函数开始，执行计算的函数是 eval
3. 在 ui.c 中定义 cmd\_p(char\* args) 函数，调用 expr.c 函数计算表达式的值，输出结果同时用十进制和十六进制表示（expression error 表示错误表达式）

### 实验结果

```

(nemu) x 3 $pc
80100008
0x80100008 : 0x0002a503
0x8010000c : 0x0000006b
0x80100010 : 0x00000000
(nemu) p $pc+18
decimal: -2146435046
hex      : 0x8010001a
(nemu) p $pc*2
decimal: 2097168
hex      : 0x200010
(nemu) p $pc/0
expression error
(nemu)

```

Ln 10, Col 6   Tab Size: 4   UTF-8   LF

图 2.1.5 表达式求值

### 2.1.3 PA1.3 实现所有要求（实现监视点）

#### 实验过程

1. 扩充监视点结构体：在 watchpoint.h 中 WP 结构体中添加 char expr[32](表达式名称) 和 uint32\_t value（表达式值）
2. 定义监视点创建函数和删除函数，WP\* new\_wp(uint32\_t value, char\* expr), WP\* free\_wp(int NO)
3. 定义显示监视点函数，void wp\_display(), 遍历 head 链表打印结点信息

#### 实验结果

```

(nemu) w $t1
watchpoint $t1 set success
(nemu) w 1+2
watchpoint 1+2 set success
(nemu) info w

```

NO	EXPR	DECIMAL	HEX
1	1+2	3	0x3
0	\$t1	0	0x0

```

(nemu) d 1
watchpoint 1 1+2 3 delete success
(nemu) info 2
Please enter help to find the use of the command info
(nemu) info w

```

NO	EXPR	DECIMAL	HEX
0	\$t1	0	0x0

```

(nemu)

```

图 2.1.6 监视点功能展示



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

1. ISA 是 riscv32
2. GDB 调试花费的时间为 270000s, 简易调试器花费  $500 \times 0.9 \times 20 = 90000s$ , 相对节省 180000s
3. Riscv32 指令格式为 R,I,S,B,U,J 6 类指令
4. LUI 指令 (`lui rd, imm`) 表示将 20 位立即数左移十二位, 并将低十二位置零, 结果写入 rd 寄存器中
5. Mstatus 用于保存全局中断以及其他状态

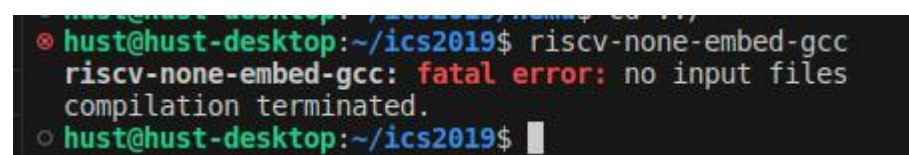
## 2.2 PA2 -简单复杂的机器：冯诺伊曼计算机系统

在 riscv-32 架构下增添新的指令集

### 2.2.1 PA2.1 在 NEMU 中运行第一个 C 程序 dummy

实验过程：

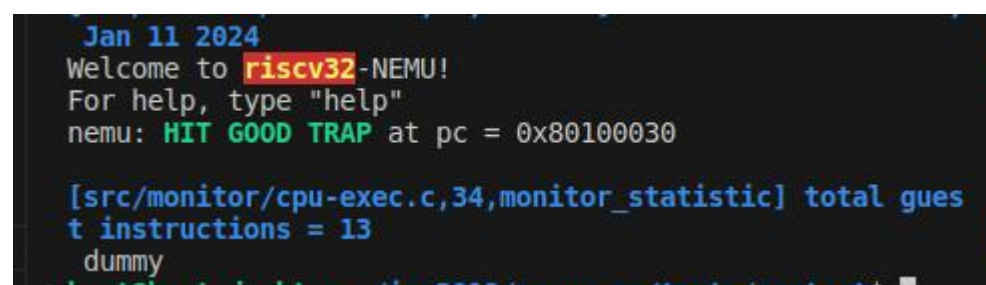
1. 准备交叉编译环境，并在 ~/.bashrc 文件中添加路径



```
hust@hust-desktop:~/ics2019$ riscv-none-embed-gcc  
riscv-none-embed-gcc: fatal error: no input files  
compilation terminated.  
hust@hust-desktop:~/ics2019$
```

2. 补充 auipc、addi、jal、jalr 指令
3. 译码阶段：在 decode.h 中声明辅助函数，在 decode.c 中定义 U 型，I 型和 J 型指令的译码辅助函数。
4. 执行阶段：调用 rtl 函数实现即可
5. 完善 exec.c 文件，根据定义的译码和执行阶段辅助函数填写 opcode-table

实验结果



```
Jan 11 2024  
Welcome to riscv32-NEMU!  
For help, type "help"  
nemu: HIT GOOD TRAP at pc = 0x80100030  
  
[src/monitor/cpu-exec.c,34,monitor_statistic] total guess  
t instructions = 13  
dummy  
hust@hust-desktop:~/ics2019/nemu-3.4.0/testcases/cputest$
```

图 2.2.1 执行 dummy 程序

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

实验过程：

1. 开启 diff-test，并添加校准指令：定义 DIFF\_TEST 宏以及 isa\_diffset\_checkregs(CPU\_state \*ref\_r, vaddr\_t pc) 函数
2. 在 opcode\_table 中添加 cputest 中需要的指令，并定义各自的译码和执行辅助函数



实验结果:

```
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 2.2.2 一键回归测试

## 2.2.3 PA2.3 运行打字小游戏

实验过程:

1. 串口设备程序, 框架代码已实现
2. 时钟设备: 需要保存 nemu 的启动时间以及当前运行时刻的返回值并计算当前时间与启动时间的差值

3. 键盘设备程序：调用 `inl` 函数读取 `KBD_ADDR` 获取键盘数据
4. VGA 设备：当 `vga` 设备接收写命令时，调用 `update_screen` 函数输出

实验结果：

```
t instructions = 2188
make[1]: Leaving directory '/home/hust/ics2019/nemu'
● hust@hust-desktop:~/ics2019/nexus-am/tests/amtest$ make
ARCH=native mainargs=H run
# Building amtest [native] with AM_HOME {/home/hust/ics2019/nexus-am}
# Building lib-am [native]
# Building lib-klib [native]
+ CC src/stdio.c
+ CC src/string.c
+ AR -> build/klib-native.a
# Creating binary image [native]
+ LD -> build/amtest-native
/home/hust/ics2019/nexus-am/tests/amtest/build/amtest-native
Usage: make run mainargs=*
H: display this help message
d: scan devices
h: hello
i: interrupt/yield test
k: readkey test
m: multiprocessor test
p: x86 virtual memory test
t: real-time clock test
v: display test
○ hust@hust-desktop:~/ics2019/nexus-am/tests/amtest$
```

图 2.2.3 -H 全部可执行的程序

```
● hust@hust-desktop:~/ics2019/nexus-am/tests/amtest$ make
ARCH=native mainargs=h run
# Building amtest [native] with AM_HOME {/home/hust/ics2019/nexus-am}
# Building lib-am [native]
# Building lib-klib [native]
# Creating binary image [native]
+ LD -> build/amtest-native
/home/hust/ics2019/nexus-am/tests/amtest/build/amtest-native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
Hello, AM World @ native
○ hust@hust-desktop:~/ics2019/nexus-am/tests/amtest$
```

图 2.2.4 -h hello.c 程序

```
hust@hust-desktop:~/ics2019/nexus-am/tests/amtest$ make
ARCH=native mainargs=t run
# Building amtest [native] with AM_HOME {/home/hust/ics2019/nexus-am}
# Building lib-am [native]
# Building lib-klib [native]
# Creating binary image [native]
+ LD -> build/amtest-native
/home/hust/ics2019/nexus-am/tests/amtest/build/amtest-native
2024-1-14 11:55:06 GMT (1 second).
2024-1-14 11:55:07 GMT (2 seconds).
2024-1-14 11:55:08 GMT (3 seconds).
2024-1-14 11:55:09 GMT (4 seconds).
2024-1-14 11:55:10 GMT (5 seconds).
2024-1-14 11:55:11 GMT (6 seconds).
2024-1-14 11:55:12 GMT (7 seconds).
2024-1-14 11:55:13 GMT (8 seconds).
2024-1-14 11:55:14 GMT (9 seconds).
hust@hust-desktop:~/ics2019/nexus-am/tests/amtest$
```

图 2.2.5 -t rtc.c 程序测试

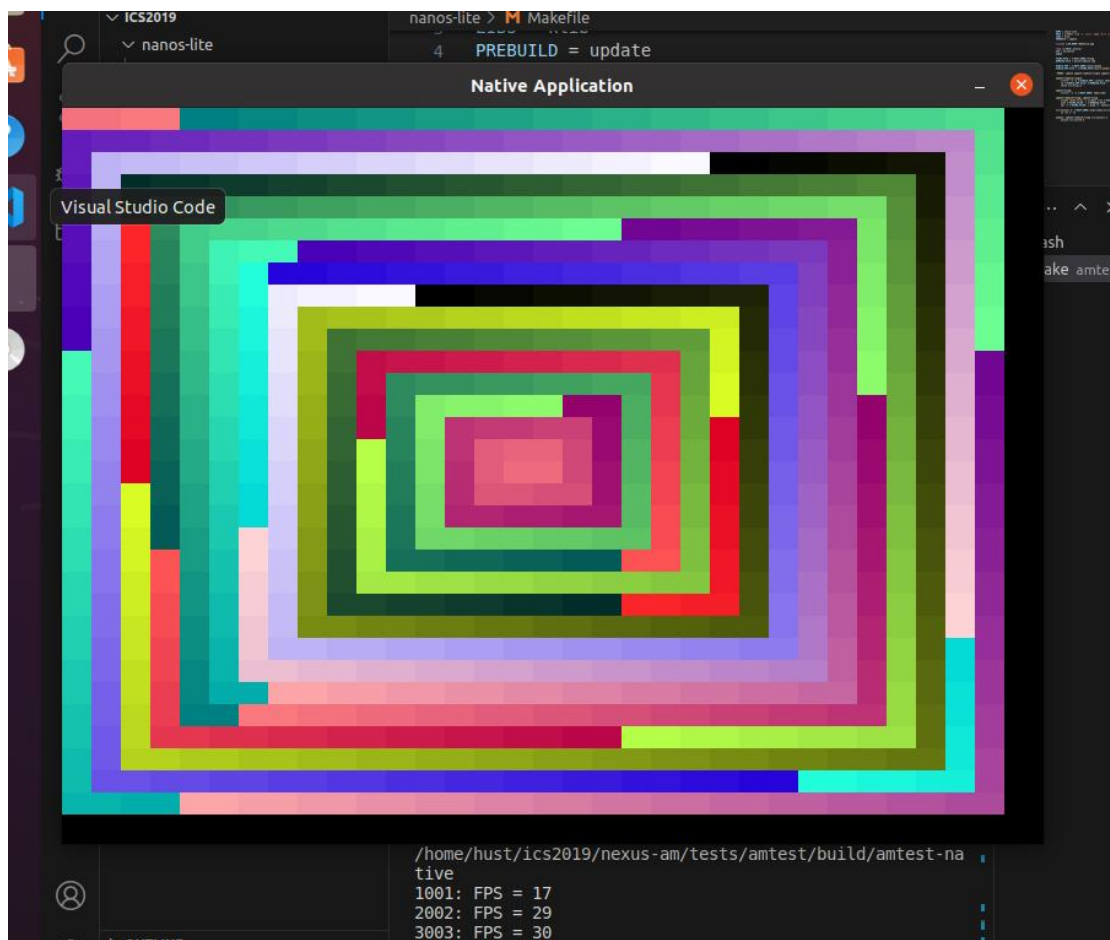


图 2.2.6 -v video.c 程序测试



```
hust@hust-desktop:~/ics2019/nexus-am/tests/amtest$ make
ARCH=native mainargs=k run
# Building amtest [native] with AM_HOME {/home/hust/ics2019/nexus-am}
# Building lib-am [native]
# Building lib-klib [native]
# Creating binary image [native]
+ LD -> build/amtest-native
/home/hust/ics2019/nexus-am/tests/amtest/build/amtest-native
Try to press any key...
Get key: 44 S down
Get key: 44 S up
Get key: 43 A down
Get key: 43 A up
Get key: 46 F down
Get key: 43 A down
Get key: 46 F up
Get key: 30 W down
Get key: 43 A up
Get key: 46 F down
Get key: 30 W up
Get key: 46 F up
Get key: 43 A down
Get key: 43 A up
Get key: 59 V down
Get key: 59 V up
Get key: 31 E down
Get key: 47 G down
Get key: 47 G up
Get key: 31 E up
Get key: 43 A down
Get key: 43 A up
Get key: 43 A down
Get key: 30 W down
Get key: 43 A up
Get key: 30 W up
hust@hust-desktop:~/ics2019/nexus-am/tests/amtest$
```

图 2.2.7 -k keyboard.c 程序测试

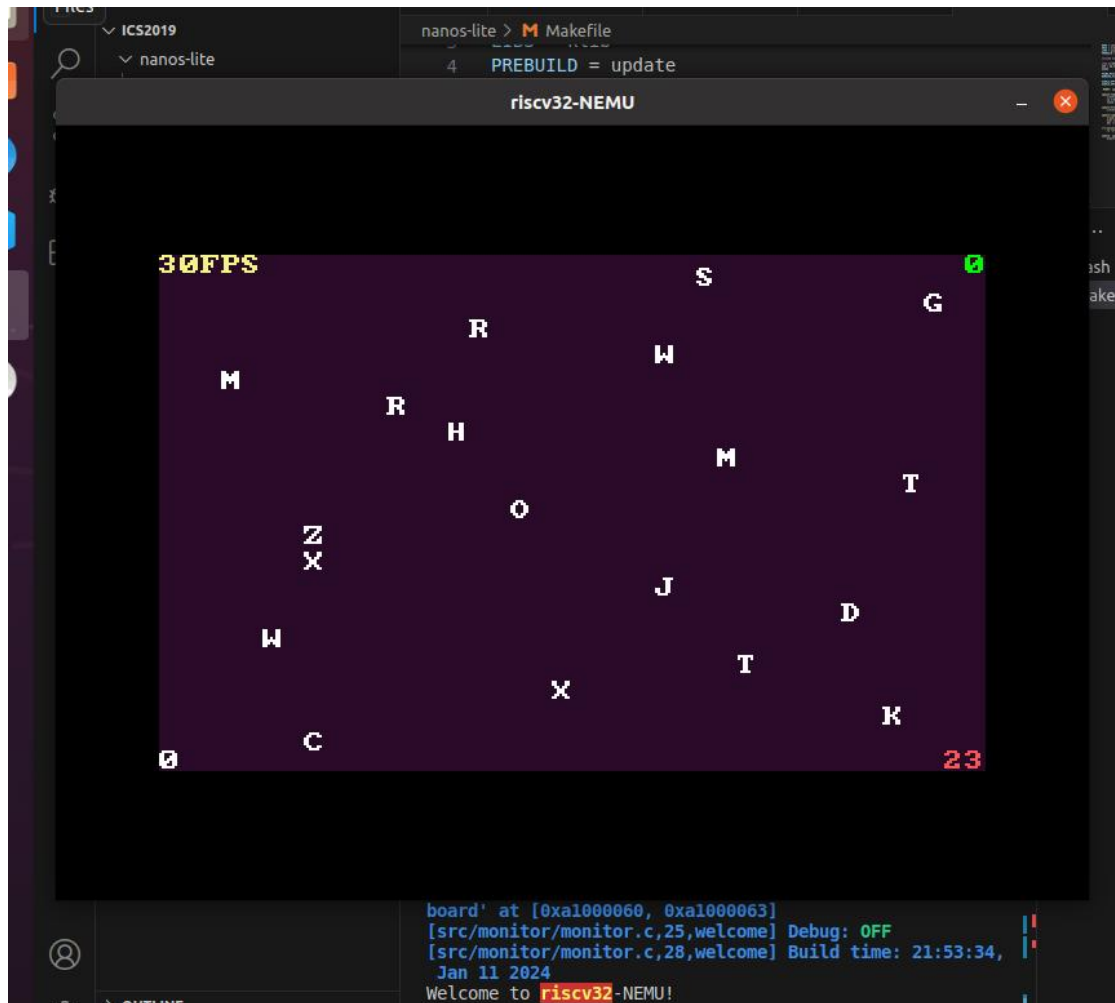


图 2.2.8 typing 测试

nanos-lite

4 PREBUILD = update

riscv32-NEMU

## 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

The diagram illustrates the Project-N ecosystem architecture, organized into several layers:

- 应用 (Application):** Includes '仙剑奇侠传' (Sword and Fairy), 'NES 模拟' (NES emulator), and 'NEMU 模拟器' (NEMU emulator). Below these are 'hello' and '编译器' (Compiler).
- 抽象层 (Abstraction Layer):** Includes '运行库: libos, libc, libndt (图形)' (Runtime: libos, libc, libndt (graphics)).
- 操作系统 (Operating System):** Includes 'Nanos 操作系统' (Nanos OS).
- 抽象层 (Abstraction Layer):** Includes 'AM 抽象计算机 = TRM + IOE + [ASYE] + [PTE]' (AM abstract computer = TRM + IOE + [ASYE] + [PTE]).
- ISA (Instruction Set Architecture):** Includes 'mips32 (NOOP-SoC)', 'NEMU X86 模拟器 (qemu)', and 'Mips32 (qemu)'. Below these are 'FPGA' and 'Linux'.

71

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

```

board' at [0xa1000060, 0xa1000063]
[src/monitor/monitor.c,25,welcome] Debug: OFF
[src/monitor/monitor.c,28,welcome] Build time: 21:53:34, Jan 11 2024
Welcome to riscv32-NEMU!
  
```

图 2.2.9 slider 测试

## 2.2.4 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, 也许会对你有帮助
- **RTFM**

1. 根据 `pc` 值取指令, 由 `op` 字段找到对应的译码辅助函数和执行函数, 译码后将相关的译码信息保存到 `decinfo` 结构体中, 执行辅助函数通过 `rtl` 指令对译码结果进行对应操作, 最后更新 `pc` 值
2. 单独去掉不会报错, 同时去掉会报错。因为在其他文件中都有对这类函数的定义, 单独去掉 `inline` 时, 由于 `static` 属性, 函数限制在本文件内不会出现重定义的问题; 但单独去掉 `static` 时, 由于 `inline` 的属性, 函数在预编译的时候就会展开, 也不会出现重定义的问题
- 3.1 通过 `grep` 指令, 有 81 个
- 3.2 82 个
- 3.3 初始化或, `remake` 报错, 两个 `dummy` 同事初始化后, 产生重复的强符号
4. `Make` 后, 将 `makefile` 中第一个目标文件作为最终的目标文件。若文件不存在, 则 `remake`; 若目标文件的依赖文件也不存在, 则递归生成上层文件

## 2.3 PA3-穿越时空的旅程: 批处理系统

(实现系统调用以及文件系统)

### 2.3.1 PA3.1 实现自陷操作 `yield()` 及其过程

实验过程:

1. 重新组织 `context` 结构体, 先压栈的是 32 个通用寄存器, 然后是 `scause`, `sstatus`, `sepc`

2. 实现正确的事件分发：在 switch 中添加几条 case 语句
3. 恢复上下文
4. 实现 loader：读取 ELF\_Ehdr 模块获得程序的总体信息，判断内存加载位置以及从 ramdisk 哪个位置开始加载
5. 识别系统调用：在 cte.c 中判断异常类型的时候添加 case0~19，在 do\_system 函数中进一步判断系统调用号并进行对应的处理逻辑
6. 处理事件：补充 irq.c 中 do\_event() 函数，根据 e 事件的事件号处理不同的事件；自陷事件需要处理 EVENT\_YIELD，使用 LOG 函数打印对应的信息。

```
2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
2019/nanos-lite/src/main.c,15,main] Build time: 15:40:37, Oct 18 2022
2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = , end = , size = -./,*,..**( byte
2019/nanos-lite/src/device.c,81,init_device] Initializing devices...
2019/nanos-lite/src/irq.c,30,init_irq] Initializing interrupt/exception handler...
2019/nanos-lite/src/proc.c,31,init_proc] Initializing processes...
2019/nanos-lite/src/main.c,33,main] Finish initialization
2019/nanos-lite/src/irq.c,14,do_event] self trapping event
2019/nanos-lite/src/main.c,39,main] system panic: Should not reach here
```

图 2.3.1 自陷事件

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

#### 实验过程：

1. 实现 loader 文件：在 loader.c 文件中定义 uintptr\_t loader() 函数，将用户程序加载到正确的内存地址并执行。
2. 识别系统调用：调用 uintptr\_t syscall() 函数，系统调用函数返回该函数返回值。
3. 实现 SYS\_yield 系统调用：调用 do\_syscall 函数用来处理系统调用。
4. 在 Nanos-lite 目录下运行 hello world: 输出由系统调用实现。修改 makefile 中 SINGLE\_APP 为 hello.c 路径。

```
Hello World from Navy-apps for the 40th time!  
Hello World from Navy-apps for the 41th time!  
Hello World from Navy-apps for the 42th time!  
Hello World from Navy-apps for the 43th time!  
Hello World from Navy-apps for the 44th time!  
Hello World from Navy-apps for the 45th time!  
Hello World from Navy-apps for the 46th time!  
Hello World from Navy-apps for the 47th time!  
Hello World from Navy-apps for the 48th time!  
Hello World from Navy-apps for the 49th time!  
Hello World from Navy-apps for the 50th time!  
Hello World from Navy-apps for the 51th time!  
Hello World from Navy-apps for the 52th time!  
Hello World from Navy-apps for the 53th time!  
Hello World from Navy-apps for the 54th time!  
Hello World from Navy-apps for the 55th time!  
Hello World from Navy-apps for the 56th time!  
Hello World from Navy-apps for the 57th time!  
Hello World from Navy-apps for the 58th time!  
Hello World from Navy-apps for the 59th time!  
Hello World from Navy-apps for the 60th time!  
Hello World from Navy-apps for the 61th time!  
Hello World from Navy-apps for the 62th time!  
Hello World from Navy-apps for the 63th time!  
Hello World from Navy-apps for the 64th time!  
Hello World from Navy-apps for the 65th time!  
Hello World from Navy-apps for the 66th time!  
Hello World from Navy-apps for the 67th time!  
Hello World from Navy-apps for the 68th time!  
Hello World from Navy-apps for the 69th time!  
Hello World from Navy-apps for the 70th time!
```

图 2.3.2 hello.c 程序运行

### 2.3.3 PA3.3 运行仙剑奇侠传并展示批处理系统

#### 实验步骤:

1. 修改 Makefile 文件:执行程序指向 pal 所在路径
  2. 实现文件记录表中打开文件的读写指针
  3. 实现基本的文件操作: 包括 fs\_open, fs\_read, fs\_write, fs\_lseek 和 fs\_close 五项操作。
  4. 为 文 件 系 统 添 加 系 统 调 用 :  
SYS\_OPEN, SYS\_READ, SYS\_CLOSE, SYS\_LSEEK
  5. 实现完整的文件系统
  6. 将串口, 设备, VGA 显存抽象成文件
- 在 NEMU 中运行仙剑奇侠传: 下载仙剑奇侠传的数据文件并将 pal 文件拷贝到 navy-apps/fsimg/share/games/ 目录下, remake 后运行
7. 展示批处理系统, 开始菜单中提供 8 个可选程序, 结果如下:

#### 实验结果:



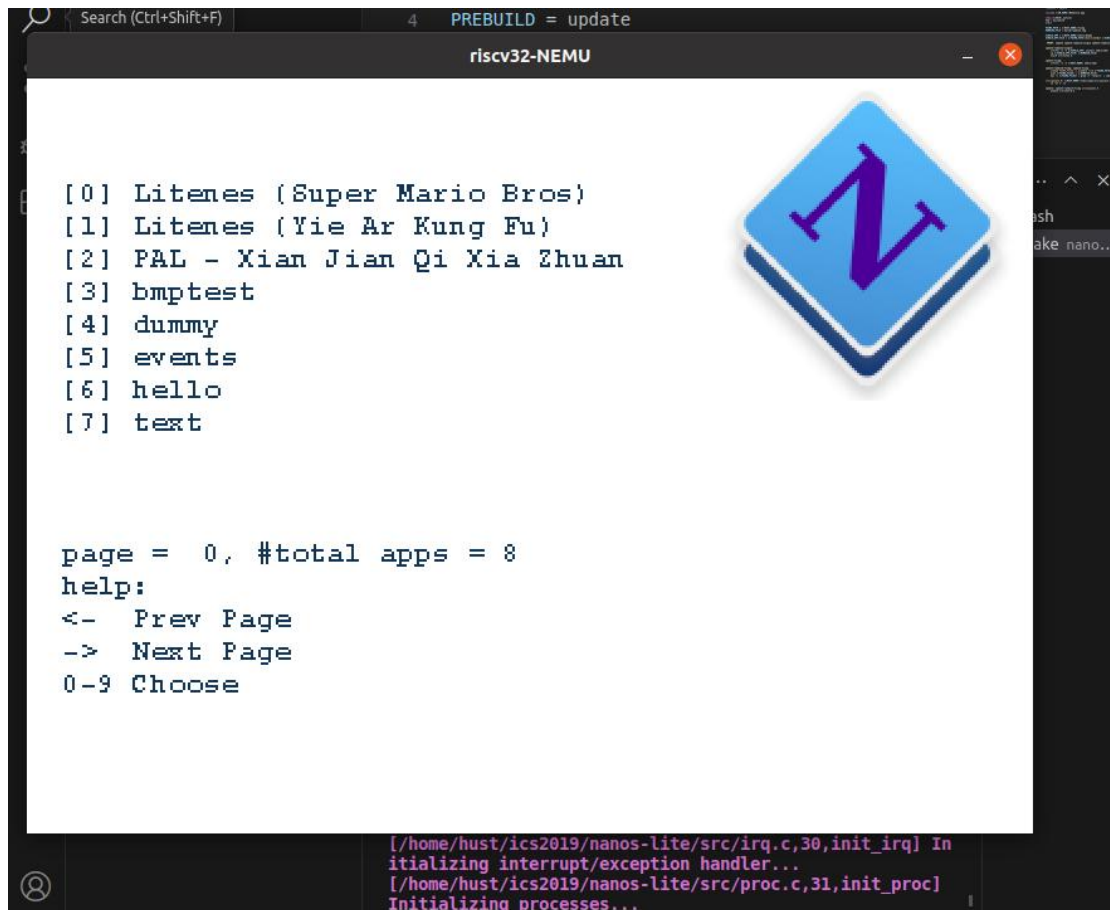


图 2.3.3 批处理菜单

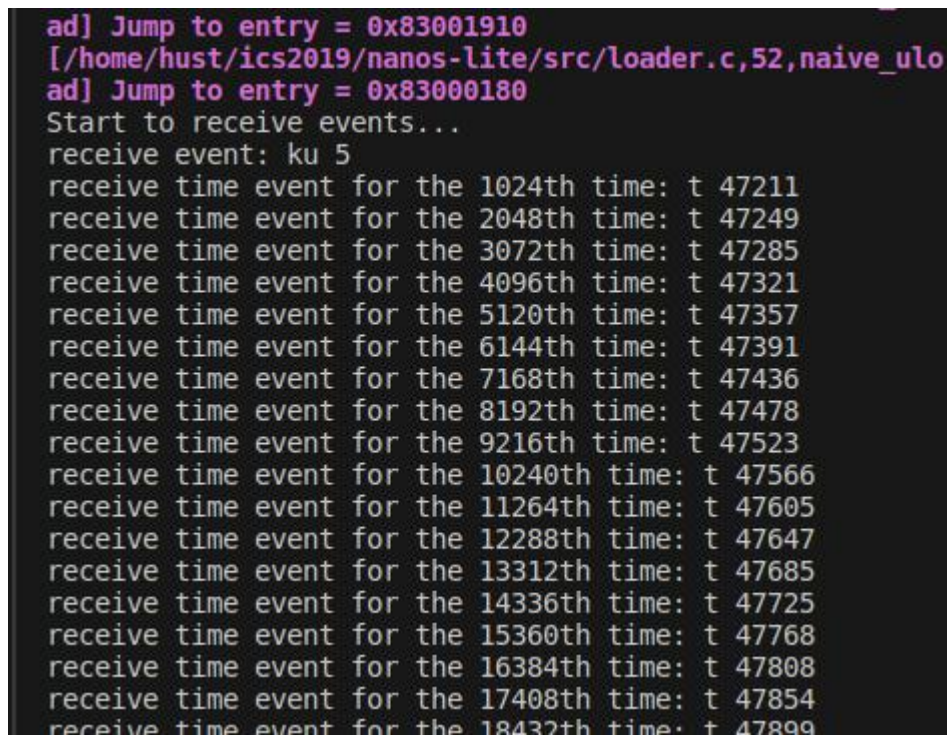


图 2.3.4 events 程序测试

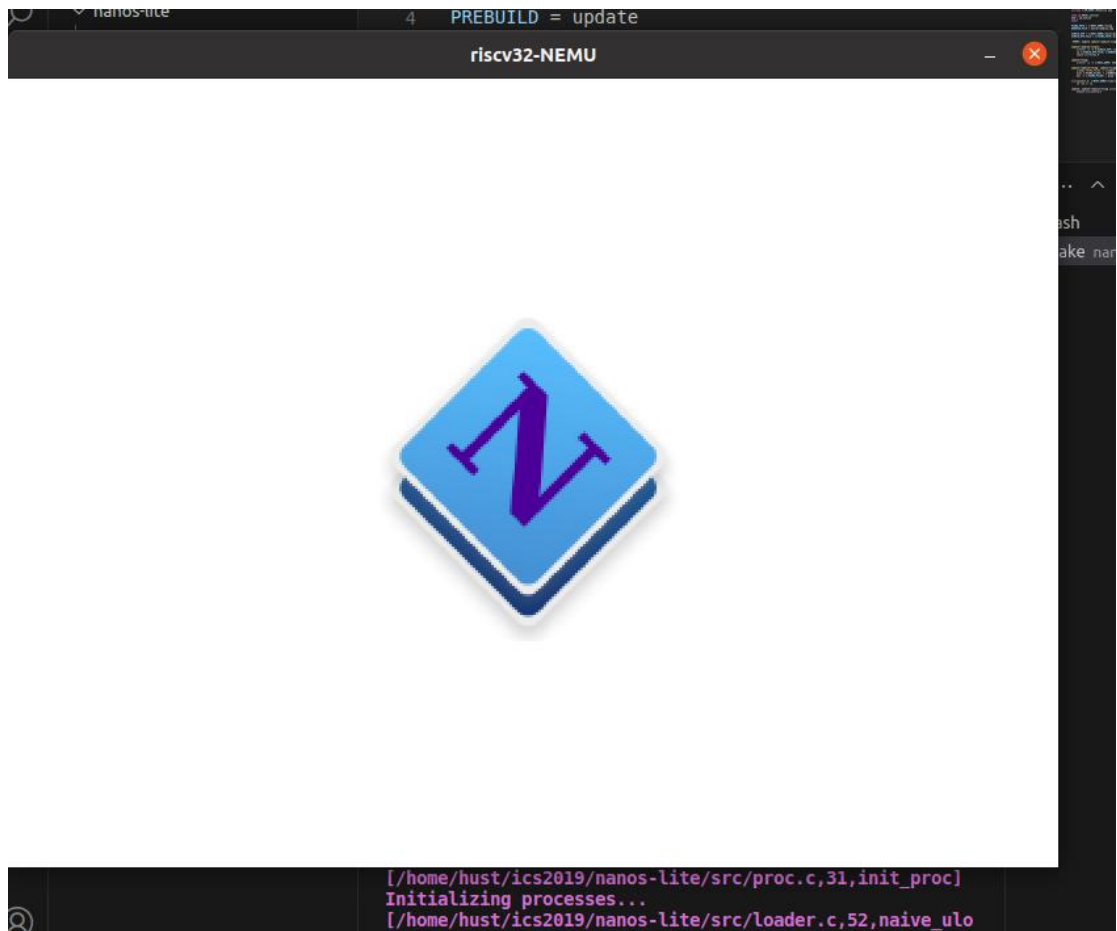
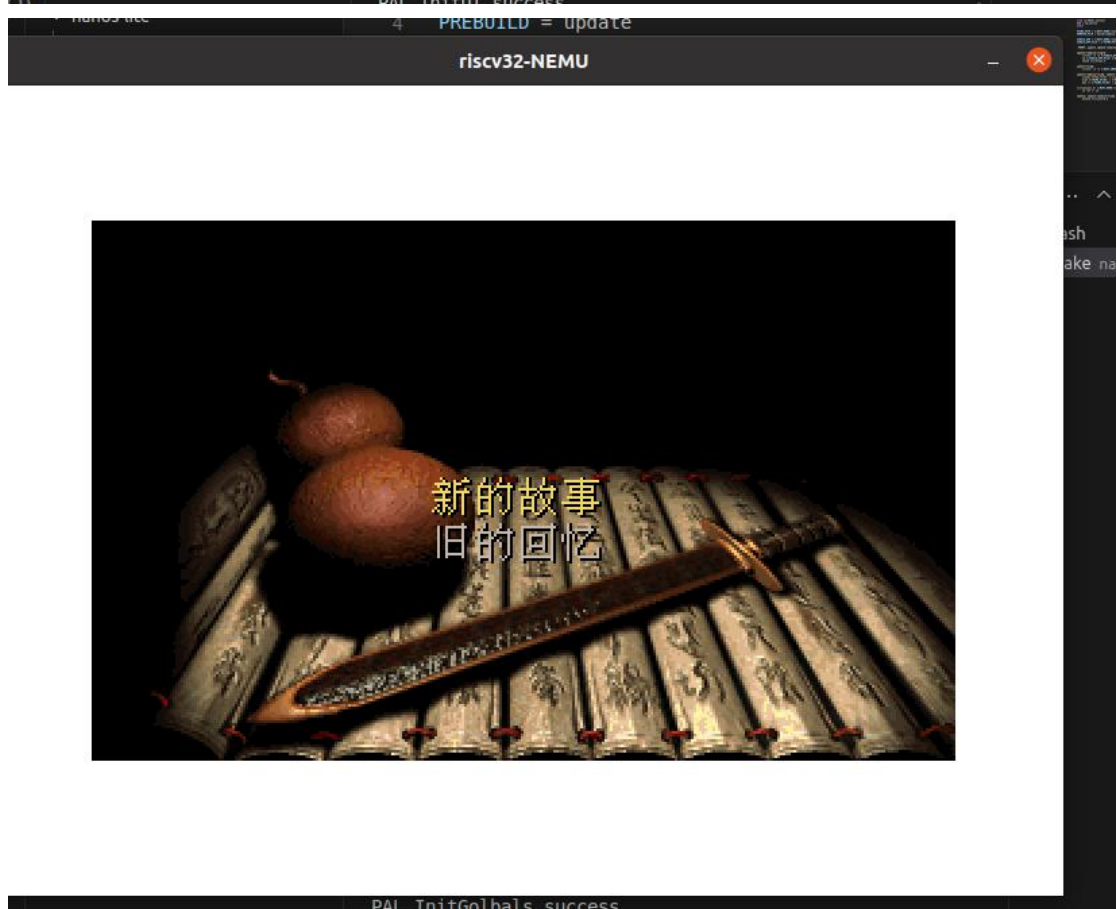


图 2.3.5 bmptest 程序测试

```
Jan 11 2024
Welcome to riscv32-NEMU!
For help, type "help"
[/home/hust/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/hust/ics2019/nanos-lite/src/main.c,15,main] Build time: 14:31:38, Jan 14 2024
[/home/hust/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = , end = , size = -./,*,...)* bytes
[/home/hust/ics2019/nanos-lite/src/device.c,82,init_device] Initializing devices...
[/home/hust/ics2019/nanos-lite/src/irq.c,30,init_irq] Initializing interrupt/exception handler...
[/home/hust/ics2019/nanos-lite/src/proc.c,31,init_proc] Initializing processes...
[/home/hust/ics2019/nanos-lite/src/loader.c,52,naive_loader] Jump to entry = 0x83001910
[/home/hust/ics2019/nanos-lite/src/loader.c,52,naive_loader] Jump to entry = 0x830002f4
PASS!!!
[/home/hust/ics2019/nanos-lite/src/loader.c,52,naive_loader] Jump to entry = 0x83001910
```

图 2.3.6 text 程序测试





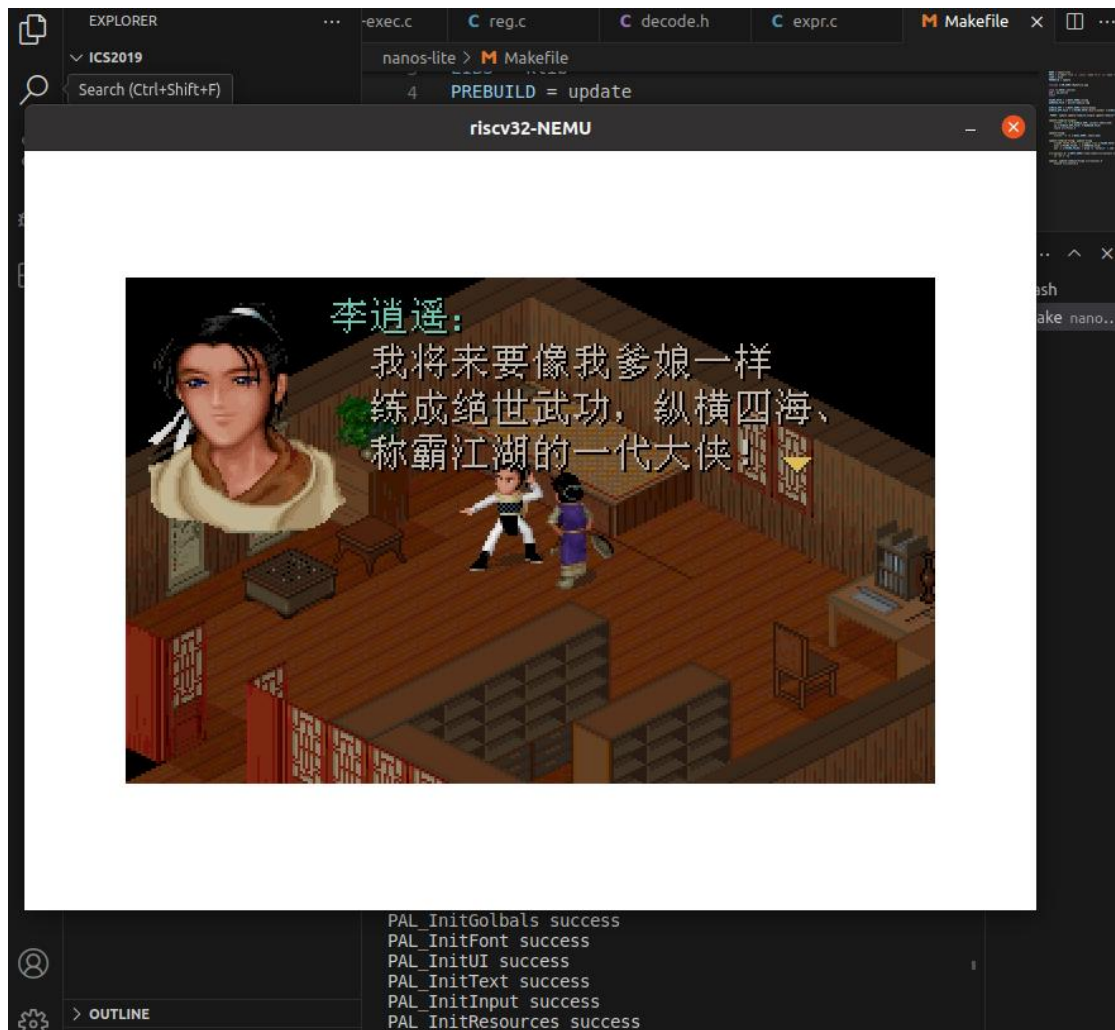


图 2.3.7 仙剑奇侠传游戏运行

#### 2.3.4 PA3 必答题

#### □ 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, 它从何而来, 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. 上下文结构保存在 `nexus-am/am/include/arch/riscv32-nemu.h`, `trap.S` 文件的前半部分负责组织上下文结构体。这个上下文结构体作为函数的参数, 保存在栈上。`trap.S` 的行为和正常的 C 程序调用函数前准备参数的过程是一样的, 根据 RISC-V 的 calling convention, `a0` 寄存器保存了第一个参数的值。我们后面要调用的函数 `__am_irq_handle` 的参数是 `Context *c`, 因此将当前的 `sp` 值传进 `a0`, 进入函数后指针 `c` 就指向了在栈上保存的上下文结构体的首地址。

2. `Nanos-lite` 调用 `yield()` 之后, 执行了两条汇编指令 (这两条汇编指令是用内联汇编的方式直接嵌入的), 其中第一条指令向 `a7` 寄存器写入 `-1`, `a7` 寄存器是约定中传递中断类型的寄存器。第二条指令 `ecall` 则是“中断”指令。`__am_asm_trap` 函数在 `trap.S` 中定义, 是用内联汇编写的。该函数首先组织上下文结构体, 组织方式见上一道必答题。然后跳转进入处理函数 `__am_irq_handle`。函数 `__am_irq_handle` 会根据上下文结构体的内容打包出一个事件结构体 `ev`。根据上下文结构体中的 `mcause` 寄存器的值, 可以识别事件的类型, 如自陷事件的事件号为 `-1`。打包完事件结构体后, `__am_irq_handle` 会将上下文结构体和事件结构体一起传给处理函数。这个处理函数是在 `cte_init` 中传进来的 `do_event`。`do_event` 函数检查事件结构体中的事件类型, 当看到是 `EVENT_YIELD` 时, 它判定这个中断没有必要重复, 因此给 `mepc` 的值 `+4`, 这个 `+4` 操作直接修改了上下文结构体中的 `mepc` 的保存值。这时进入 `trap.S` 的后半部分, 后半部分将栈上保存的寄存器内容恢复, 然后调用 `mret` 指令。在硬件层面, `nemu` 识别出 `mret` 指令后, 直接将 `pc` 恢复为 `mepc` 的值。至此, 时空穿越的旅程结束。

3. 由于此时还没有实现文件系统, 所以 `ramdisk.img` 中只有 `hello` 程序。`nanos-lite` 中的内联汇编程序 `resources.S` 将 `ramdisk.img` 的内容加载到了内存中。在 C 程序中我们可以很方便地定位 `ramdisk.img` 被加载到了内存中的哪个位置。从硬件层面来说, 由于现在还没有引入虚拟内存, `ramdisk.img` 被放在了地址 `0x83000000` 处, `Makefile` 的 `LNK_ADDR` 实现了这一点。在调用了 `naive_oload` 函数后, 系统会进入 `hello` 程序的汇编代码 (elf 文件中有入口地址), `hello` 程序输出字符的时候, 会使用 `write` 系统调用 (直接使用 `write` 或者通过 `printf` 库函数使用 `write`), `nanos-lite` 中的 `fs_write` 函数会使用 `AM` 提供的接

口来输出（在实现了文件系统之后，`fs_write` 会直接使用 `stdout` 文件对应的写函数 `serial_write`）。

4. 程序的行为总体上是读出仙鹤的像素信息，然后不断更新到屏幕上并修改仙鹤的位置。在软件层面，程序会使用 `fseek` 和 `fread` 库函数来定位和读取像素信息，使用 SDL 库函数来输出。`fseek` 和 `fread` 函数最终会使用 `lseek` 和 `read` 系统调用，SDL 库函数基于我们对系统调用进一步封装的 NDL 库实现，NDL 库会使用系统调用 `lseek` 和 `write` 来实现像素的定位和写入（注意 VGA 被抽象成了文件，读写像素不需要专门的 I/O 相关函数。）`lseek` `read` 和 `write` 最终会调用 `libos` 中的 `_read`, `_write` 和 `_lseek`，传递对应的系统调用号和相关参数，使用 `ecall` 指令触发中断。Nanos-lite 中会识别系统调用的类型，并调用 `fs_read` `fs_write` 和 `fs_lseek`。`fs_lseek` 比较简单，只是修改文件当前的 `offset`，`fs_read` 会使用 `/bin/pal` 对应的读函数 `ramdisk_read` 来从“硬盘”中读取信息（事实上已经被加载到内存中）`ramdisk_read` 使用的是 AM 的函数 `memcpy`；`fs_write` 会使用 `/dev/fb` 文件对应的写函数 `fb_write`。`fb_write` 会使用 AM 提供的 VGA 相关的抽象寄存器和一段像素空间来写入信息。AM 和 `nemu` 之间有一套内存 I/O 的映射规定，`nemu` 会根据这些 I/O 空间的信息进行硬件层面的修改。`memcpy` 是 `klib` 中的函数，可以直接访问 `nemu` 的大数组（内存），实现信息的读取

### 3 实验结果与结果分析

PA1~3 的实验结果均在各阶段结尾展示，可见上图。分析如下：

1. PA1 的目标实现一个简易的调试器，难点主要在于运算符的识别和监视点功能的实现
2. PA2 的目标时候显示 riscv 指令，重点在于理解 riscv 架构下各指令的功能和结构，在实践过程中需要大量重复地翻阅相关文献。同时也可以感受到相较于 mip 架构，riscv 结构下指令的精简程度
3. PA3 的目标任务是实现系统调用和文件系统，需要深入理解系统调用以及自陷操作的过程

## 4 电子签名

V202015315  
钟健组