***2020***



**系统能力综合训练 课程设计报告**

|  |  |
| --- | --- |
| 题 目： | riscv模拟器设计 |
| 专 业： | 计算机科学与技术 |
| 班 级： |  |
| 学 号： |  |
| 姓 名： |  |
| 电 话： |  |
| 邮 件： |  |
| 完成日期： | 2021-01-12 |

目 录

[1 课程设计概述 1](#_Toc61307740)

[1.1 课设目的 1](#_Toc61307741)

[1.2 课设任务 1](#_Toc61307742)

[1.3 实验环境 1](#_Toc61307743)

[2 PA1 2](#_Toc61307744)

[2.1 故障分析 2](#_Toc61307745)

[2.2 思考题 2](#_Toc61307746)

[2.3 必答题 3](#_Toc61307747)

[2.4 结果展示 5](#_Toc61307748)

[3 PA2 6](#_Toc61307749)

[3.1 故障分析 6](#_Toc61307750)

[3.2 思考题 6](#_Toc61307751)

[3.3 必答题 6](#_Toc61307752)

[3.4 结果展示 9](#_Toc61307753)

[4 PA3 11](#_Toc61307754)

[4.1 故障分析 11](#_Toc61307755)

[4.2 思考题 11](#_Toc61307756)

[4.3 必答题 12](#_Toc61307757)

[4.4 结果展示 13](#_Toc61307758)

[5 设计总结与心得 15](#_Toc61307759)

[参考文献 16](#_Toc61307760)

# 课程设计概述

## 课设目的

理解“程序如何在计算机上运行”的根本途径是实现一个完整的计算机系统. 本课程引入南京大学计算机科学与技术系《计算机系统基础》课程的小型项目(Programming Assignment, PA)，将指导学生实现一个功能完备(但经过简化)的模拟器NEMU (NJU EMUlator)，可解释执行X86、riscv或mips32执行代码，最终在该模拟器上运行开源游戏“仙剑奇侠传”,让学生探究“程序在计算机上运行”的机理。

## 课设任务

PA项目包括一个准备实验(配置实验环境)以及4部分连贯的实验内容，具体时间和任务分布如下，该项目提供了总体框架代码，学生只需要完成框架代码中的部分代码，故总体代码量只有2000多行，但阅读代码框架是一项较大的挑战。



## 实验环境

* 框架：ics2020
* 指令集：riscv32
* OS：Ubuntu-18.04.1
* QEMU：qemu-system-riscv32 version 5.1.0
* 交叉编译工具：g++-riscv64-linux-gnu, binutils-riscv64-linux-gnu
* 编辑器：VS Code

# PA1

## 故障分析

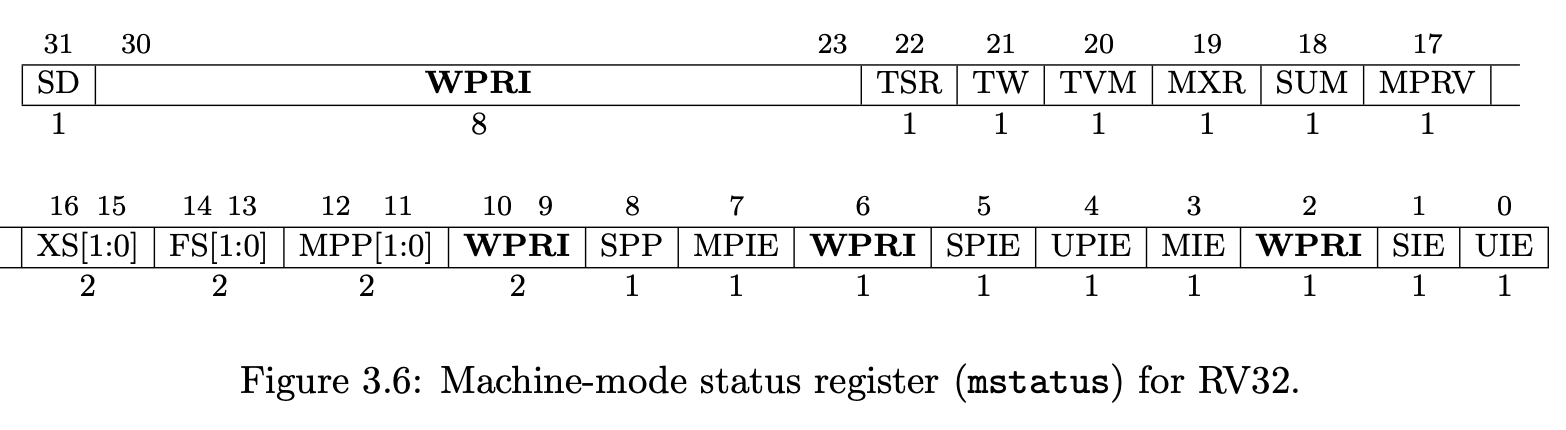
* 表达式求值十六进制出错。原因：没有注意到正则表达式匹配的顺序，将十六进制的匹配规则放到十进制之前即可解决问题。
* 表达式求值 make\_token 中 token 字符串拷贝没有考虑长度。通过观察 log 找到问题原因和 bug 位置。将 strcpy 函数换为 strncpy 函数，问题解决。

## 思考题

* 在cmd\_c()函数中, 调用cpu\_exec()的时候传入了参数 -1, 你知道这是什么意思吗?
* **答：**cpu\_exec的参数 n 为 64 位无符号整型，表示 cpu 连续执行的指令数量，传入的参数 -1 在经过类型转换后会变为最大的无符号整数，表示让 cpu 不停地执行指令
* 如果printf不换行, 可能会发生什么? 你可以在代码中尝试一下, 并思考原因, 然后STFW对比你的想法.
* **答：**fwrite() 的实现中有缓冲区, printf() 打印的字符不一定会马上通过 write() 系统调用输出, 但遇到 \n 时可以强行将缓冲区中的内容进行输出.
* 我们在表达式求值中约定, 所有运算都是无符号运算. 你知道为什么要这样约定吗? 如果进行有符号运算, 有可能会发生什么问题?
* **答：**因为在表达式求值中可能会出现 0x80000000 这样的“负数”，将所有运算定义成无符号运算，可以简化地址的比较等运算
* 如果把断点设置在指令的非首字节(中间或末尾), 会发生什么? 你可以在GDB中尝试一下, 然后思考并解释其中的缘由.
* **答：**程序不会停下来，因为运行过程中 pc 寄存器的值不会与设置的断点相等
* 你已经对NEMU的工作方式有所了解了. 事实上在NEMU诞生之前, NEMU曾经有一段时间并不叫NEMU, 而是叫NDB(NJU Debugger), 后来由于某种原因才改名为NEMU. 如果你想知道这一段史前的秘密, 你首先需要了解这样一个问题: 模拟器(Emulator)和调试器(Debugger)有什么不同? 更具体地, 和NEMU相比, GDB到底是如何调试程序的?
* **答：**GDB 中的操作底层都是通过 ptrace 系统调用来实现的。当进程调用了 ptrace 跟踪某个进程之后：
  + 调用 ptrace 的进程会变成被跟踪进程的父进程;
  + 被跟踪进程的进程状态被标记为 TASK\_TRACED;
  + 发送给被跟踪子进程的信号 (SIGKILL 除外) 会被转发给父进程, 而子进程会被阻塞;
  + 父进程收到信号后, 可以对子进程进行检查和修改, 然后让子进程继续执行;
* 当我们用 GDB 设置断点时, GDB 会把断点处的指令修改成 int 3, 同时把断点信息及修改前的指令保存起来. 当被调试子进程运行到断点处时, 便会执行 int 3命令, 从而产生 SIGTRAP 信号. 由于 GDB 已经用 ptrace 和调试进程建立了跟踪关系, 此时的 SIGTRAP 信号会被发送给 GDB, GDB 通过和已有的断点信息做对比 (通过指令位置) 来判断这次 SIGTRAP 是不是一个断点.

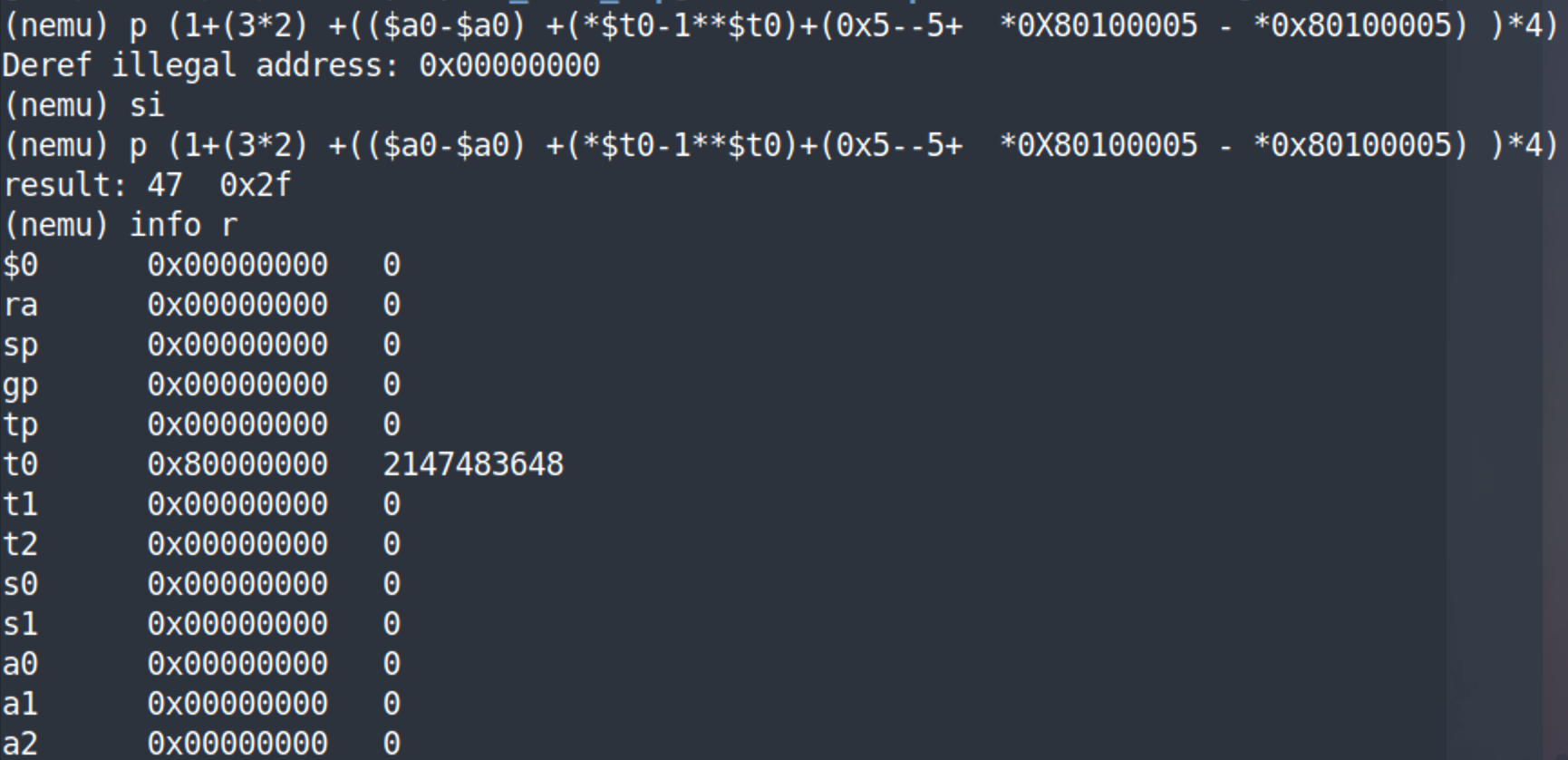
## 必答题

* 我选择的ISA是 riscv32.
* 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?
* **答：**使用 GDB 需花费 75h，使用简易调试器可节省 50h。
* RTFM
* 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:
  + riscv32有哪几种指令格式?
  + **答：**六种基本指令格式：R：寄存器-寄存器操作；I：短立即数操作、访存load操作；S:访存store操作；B:条件跳转；U:长立即数操作；J:无条件跳转
  + LUI指令的行为是什么?
  + **答：**高位立即数加载(Load Upper Immediate)。将符号位扩展的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?



* 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pa0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"?) 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?
  + git diff --stat pa0
  + find nemu -name "\*.[c|h]" | xargs cat | grep -v ^$ | wc -l
* RTFM 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?
  + -Wall，开启gcc的所有警告。
  + -Werror，将所有警告视为错误进行处理。

## 结果展示



# PA2

## 故障分析

* 少数 jalr 指令跳转出错。通过 diff-test 定位到错误，发现少执行了一条指令，跳转的地址比正常地址加了四。原因：没有考虑到源寄存器与目的寄存器相同的情况，先修改了目的寄存器的值，导致源寄存器值被修改，跳转到了错误的地址。
* 实现 printf 时可变参数的使用出现问题。将 va\_list 参数传递给另一个函数时编译器给出警告，类型不匹配，导致编译不通过。原因：va\_list 为数组类型，在参数传递过程中会导致类型弱化为指针类型。在参数传递前使用 va\_copy 复制，将拷贝值当作参数传递。并在函数返回后将传入的参数拷贝回来。
* 实现 diff-test 后 qemu 无法启动。原因：新版 qemu 运行需要添加额外参数 -bios none 即可运行。
* 运行时钟测试时输出过快。原因：am 实现时钟寄存器时没有初始化记录初始时间，导致启动时间计算不正确。

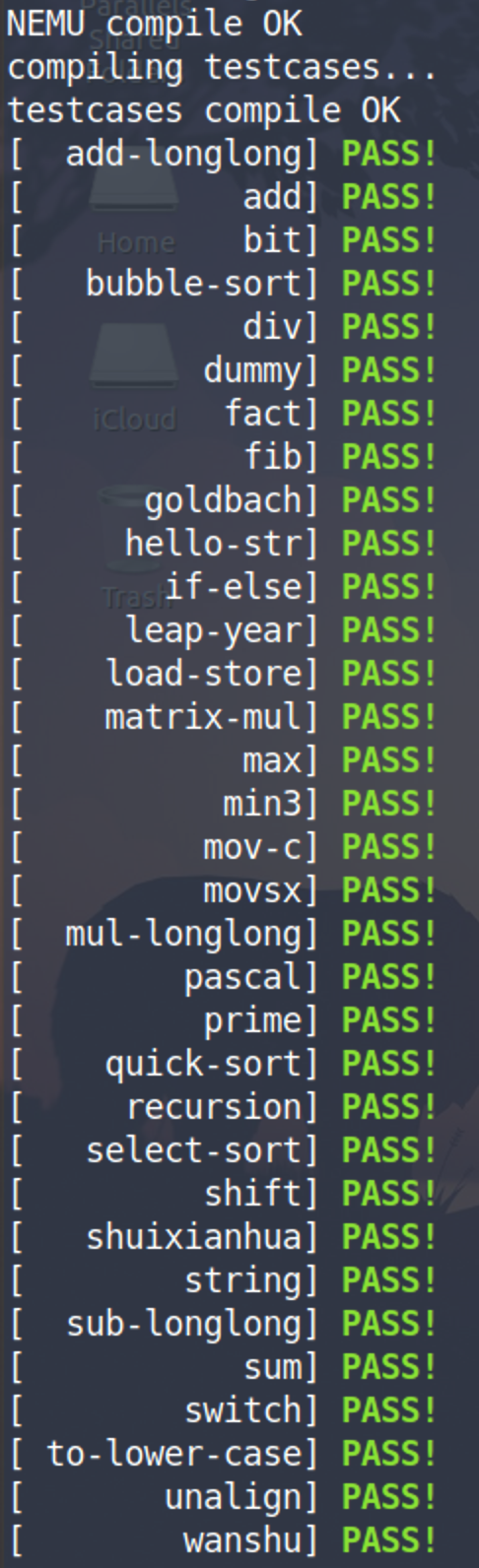
## 思考题

* mips32和riscv32的指令长度只有32位, 因此它们不能像x86那样, 把C代码中的32位常数直接编码到一条指令中. 思考一下, mips32和riscv32应该如何解决这个问题?
* **答：**RISC-V 使用 I-type 格式的指令和 U-type 格式的指令加载立即数，I-type 格式包含 12 位立即数， U-type 格式包含 20 位立即数，而且这 20 位是加载到寄存器的高 20 位的。所以，一条 I-type 指令加上一条 U-type 指令就可以加载 32 位的常量，包括 32 位的立即数或内存地址。
* 请你通过RTFSC理解这个参数是如何从make命令中传递到hello程序中的, $ISA-nemu和native采用了不同的传递方法, 都值得你去了解一下.
* **答：**nemu 在 trm.c 文件中将 MAINARGS 宏定义作为参数传递给了 main 函数，并在 nemu.mk 中添加了-DMAINARGS="$(mainargs)" 的 CFLAGS ，让编译器将宏展开为我们传入 makefile 的 mainargs 变量。
* native 中通过 getenv 获取 mainargs 环境变量，然后将其传入 main 函数。

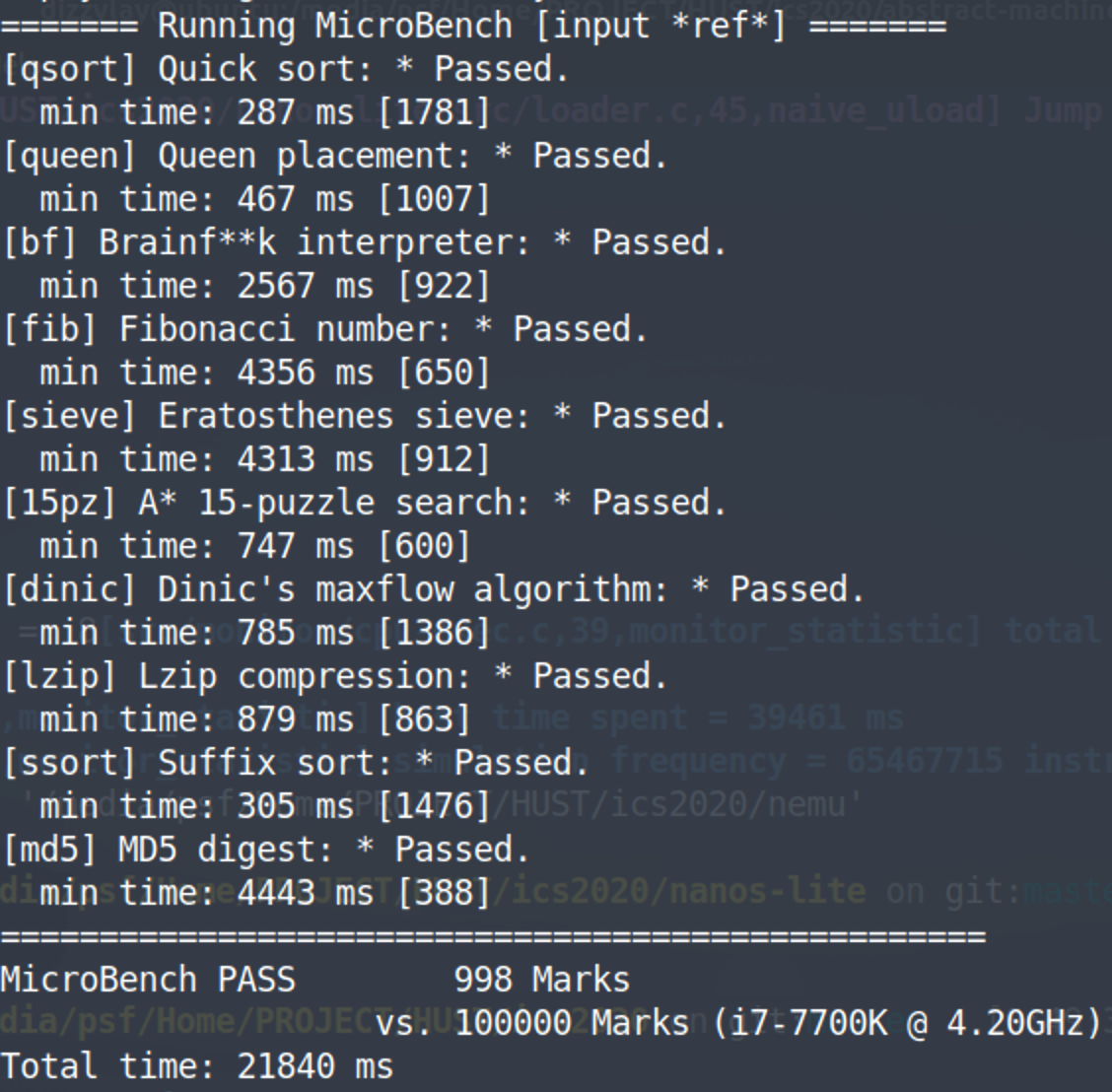
## 必答题

* RTFSC 请整理一条指令在NEMU中的执行过程.
* **答：**以 load 为例，nemu 先根据 pc 寄存器在内存中将指令取出，然后根据指令的 opcode ，使用 I 型译码辅助函数对指令译码，然后调用 load 的执行辅助函数，根据funct3字段确定 load 指令的宽度和是否有符号，并调用相应的 rtl 指令。执行完毕后，更新 pc 寄存器。
* 程序如何运行 理解打字小游戏如何运行.
* **答：**游戏会通过循环不断获取用户的按键，然后根据按键判断是否命中当前的字符，若命中，则改变对应字符的速度为负。在屏幕上绘制时通过速度判断一个字符是否命中，并显示相应的颜色。速度大于0表示还未命中，等于0表示miss，小于0表示命中。
* 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?
* **答：**去掉 static 可以正常编译，因为编译器会将函数调用内联成对应的代码。
* 去掉 inline 会有编译警告部分函数未使用。
* 都去掉将会有 error ，部分函数在多个文件中重复定义。因为 rtl 指令定义在头文件中，会被多个源文件引用。
* 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?
  + **答：** 变量实体的数量取决于 common.h 被多少个源文件 include ，因为 static 保证变量仅在源文件内可见。
  1. 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.
  + **答：**数量与上题相同，因为 static 定义的未初始化变量为弱符号，源文件中多次定义只会有一个生效。
  1. 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.)
  + **答：**编译会出现错误，因为初始化的全局变量是强符号，只能定义一次。
* 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
* **答：**makefile 通过 make 指令传入的 ISA 变量确定要编译的指令架构，并筛选出相应的源文件，确定头文件目录、中间文件生成目录以及编译目标文件的名字。然后执行编译和链接的过程。

## 结果展示







# PA3

## 故障分析

* 执行 ecall 指令内陷处理完毕调用sret返回时，pc 又回到了 ecall 指令，导致无限递归。原因：处理中断时没有在合适的地方对 pc 进行加四，导致返回地址错误。

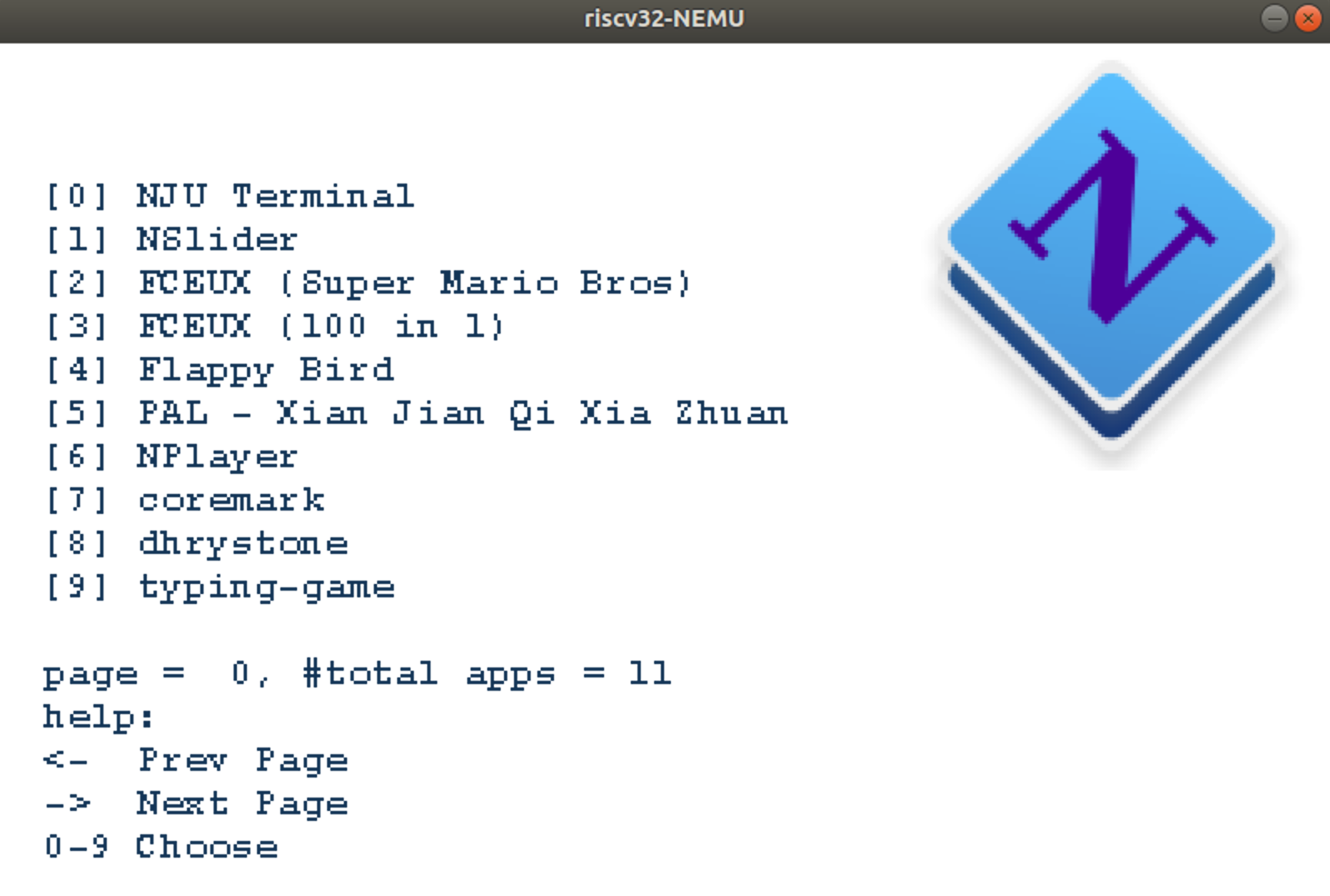
## 思考题

* 这些程序状态(x86的eflags, cs, eip; mips32的epc, status, cause; riscv32的sepc, sstatus, scause)必须由硬件来保存吗? 能否通过软件来保存? 为什么?
* **答：**我认为必须由硬件保存，因为中断包含硬件中断和软件中断，硬件中断的状态无法通过软件保存。
* x86通过软件来保存异常号, 没有类似cause的寄存器. mips32和riscv32也可以这样吗? 为什么?
* **答：**不能，mips32 和 riscv32 没有中断向量表，所有中断入口相同，软件无法确定异常号
* 自陷决定PC要不要加4的, 是硬件还是软件呢? CISC和RISC的做法正好相反, CISC都交给硬件来做, 而RISC则交给软件来做. 思考一下, 这两种方案各有什么取舍? 你认为哪种更合理呢? 为什么?
* **答：**硬件实现可以简化软件，但增加硬件复杂度；软件实现可以使硬件电路更简单。我认为通过软件实现更加灵活。
* 我们提到了代码和数据都在可执行文件里面, 但却没有提到堆(heap)和栈(stack). 为什么堆和栈的内容没有放入可执行文件里面? 那程序运行时刻用到的堆和栈又是怎么来的? AM的代码是否能给你带来一些启发?
* **答：**因为堆和栈都是程序运行时才有的概念，是程序内存中的特定区域。栈由操作系统管理，负责临时变量的分配与回收、参数传递、保存返回地址等，而堆区内存由用户程序主动分配和释放。
* 如果你在GNU/Linux下执行一个从Windows拷过来的可执行文件, 将会报告"格式错误". 思考一下, GNU/Linux是如何知道"格式错误"的?
* **答：**通过可执行文件起始4字节的magic number来判断格式，Windows 可执行文件为PE格式，linux 可执行文件为ELF格式。
* 使用readelf查看一个ELF文件的信息, 你会看到一个segment包含两个大小的属性, 分别是FileSiz和MemSiz, 这是为什么? 再仔细观察一下, 你会发现FileSiz通常不会大于相应的MemSiz, 这又是为什么?
* **答：**程序的数据段包含初始化和未初始化的变量，初始化的变量保存在可执行文件中，未初始化的变量不需要保存，因此有一个叫做 bss 的特殊 section ，在文件中大小为0。操作系统加载程序时将内存中 bss 部分的变量统一清零。
* 为什么需要将 [VirtAddr + FileSiz, VirtAddr + MemSiz) 对应的物理区间清零?
* **答：**这一区间对应程序段中的 bss section，表示未初始化的变量，需要将这些变量初始化为0。
* 对于批处理系统来说, 系统调用是必须的吗? 如果直接把AM的API暴露给批处理系统中的程序, 会不会有问题呢?
* **答：**批处理系统每次只会运行一个程序，因此不必管理计算机资源，系统调用也不是必须。但直接暴露 AM 的 API 会导致应用程序可以直接操纵硬件，可能会有恶意程序进行破坏。

## 必答题

* 理解上下文结构体的前世今生 (见PA3.1阶段)。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?
* **答：**c 指向的上下文结构在栈上，是由 trap.S 中保存上下文时创建的，结构体成员与压栈的顺序一致。
* 理解穿越时空的旅程 (见PA3.1阶段)
* **答：**操作系统在初始化时通过 cte\_init 设置了异常入口地址和中断服务例程。当自陷指令执行时， nemu 会调用raise\_intr 函数模拟异常响应机制，保存 pc 寄存器，设置异常号然后跳转到异常入口地址。跳转后会保存上下文信息，接着调用 \_\_am\_irq\_handle 函数进行异常处理。它会把执行流切换的原因打包成事件, 然后调用在 cte\_init 中注册的事件处理回调函数, 将事件交给操作系统来处理. 这一回调函数是 nanos-lite/src/irq.c中的 do\_event() 函数. do\_event() 函数会根据事件类型再次进行分发。
* hello程序是什么, 一开始在哪里? 它是怎么出现内存中的? 为什么会出现在目前的内存位置? 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?
* **答：**hello 程序编译成二进制以后，被包含进了resource.S文件中，以 ramdisk\_start 和 ramdisk\_end 表示镜像的起始和结束位置。操作系统初始化时会执行 init\_proc 函数将 hello 程序加载到内存。hello 是 elf 格式的可执行文件，可以根据程序头部表得到各个程序段的信息，并将程序段的数据载入段头部表 p\_vaddr 属性指示的内存地址。在 riscv32.mk 文件中添加了 -Ttext-segment 链接参数，使程序的代码段被链接到 0x83000000 的高地址处。程序载入内存之后，根据程序头部表的 e\_entry 属性找到程序执行的入口并执行。hello 程序调用了 libc 库的 printf 函数，最终会调用到操作系统的 write 系统调用。然后由文件描述符调用到 serial\_write 函数，通过 AM 的 putch api 进行串口输出。
* 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/repo/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?
* **答：**仙剑奇侠传在读取 mgo.mkf 的数据时，用到了文件读写的库函数，本质上是 read 和 write 系统调用的封装。最后会将像素信息写入显示设备，这是通过 AM 提供的设备访问 api 实现的。NEMU中对内存映射 IO 进行了模拟，以显示设备为例，NEMU使用了 SDL 库来模拟，将显存中的信息绘制到屏幕上，从而在电脑上看到动画。

## 结果展示







# 设计总结与心得

本次PA实验在框架代码的基础上完成了一个riscv32模拟器，并成功运行了仙剑奇侠传，效果很棒。实验的工作量较大，前后总共花费近三周时间完成。因为之前学习操作系统时完成了jos实验，对很多概念理解到位了，因此写PA时没有感觉很吃力，但是寻找bug与调试代码仍然是一项艰巨的任务，时间的大头也都花在这上面了。做完以后最大的感慨就是riscv指令集设计真的很精巧，基础指令集足够简洁的同时，也充分考虑了指令集的扩展性。整个计算机系统就像积木一样，通过抽象的方式一层层的搭建起来，硬件层、AM层、系统层与应用层的巧妙配合，使得仙剑奇侠传这样的游戏得以运行起来。本次实验最大的收获应该是代码阅读能力与信息检索能力得到了提升，今后在面对更加庞大的项目时也不会感到畏惧了。自身的编码能力也得到了良好的锻炼，对今后的学习工作受益匪浅。

# 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.