***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： | ACM1901班 |
| 学 号： | U201915035 |
| 姓 名： | 邹雅 |
| 电 话： | 15058667378 |
| 邮 件： | 1542527211@qq.com |
| 完成日期： | 2022-10-19 |

目 录

[1 课程实验概述 1](#_Toc117154271)

[1.1 课程要求 1](#_Toc117154272)

[1.2 课程设计 1](#_Toc117154273)

[2 开天辟地的篇章：最简单的计算机 2](#_Toc117154278)

[2.1 思考题 2](#_Toc117154279)

[2.2 单步执行 2](#_Toc117154280)

[2.3 打印寄存器状态 3](#_Toc117154281)

[2.4 扫描内存 4](#_Toc117154282)

[2.5 表达式求值 4](#_Toc117154283)

[2.6 表达式生成器 6](#_Toc117154284)

[2.7 设置监视点 7](#_Toc117154285)

[2.8 删除监视点 8](#_Toc117154286)

[2.9 打印监视点 8](#_Toc117154287)

[2.10 必答题 9](#_Toc117154288)

[3 简单复杂的机器：冯诺依曼计算机系统 11](#_Toc117154294)

[3.1 PA 2.1 11](#_Toc117154295)

[3.2 PA 2.2 12](#_Toc117154296)

[3.3 PA 2.3 14](#_Toc117154297)

[3.4 必答题 17](#_Toc117154298)

[4 穿越时空的旅程：批处理系统 18](#_Toc117154303)

[4.1 PA 3.1 18](#_Toc117154304)

[4.2 PA 3.2 20](#_Toc117154305)

[4.3 PA 3.3 21](#_Toc117154306)

[4.4 必答题 26](#_Toc117154307)

[总结与感想 28](#_Toc117154312)

# 课程实验概述

## 课程要求

本课程设计的总体目标是基于设计好的模拟器代码框架，实现一个简化的 RISC-V 模拟器。要求可解释执行 RISC-V 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。

## 课程设计

### 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。

### 图灵机

实现简易调试器、表达式求值、监视点与断点。

### 冯诺依曼计算机

运行第一个C程序；丰富指令集；实现I/O指令。

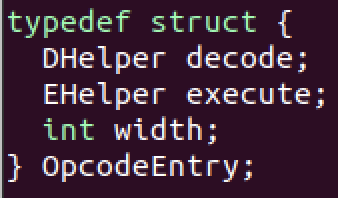
### 异常控制流

实现系统调用；实现文件系统；运行仙剑奇侠传。

# 开天辟地的篇章：最简单的计算机

## 思考题

1. 寄存器采用匿名联合体。使用联合体的原因是其中的数据共享同一块内存。
2. 在cmd\_c()函数中, 调用cpu\_exec()的时候传入了参数-1。-1的十六进制表示就是0xffffffff, 然后这是一个无符号的类型, 所以是最大的无符号数。因此传入 -1可以让函数执行时间达到最长。
3. opcode\_table是一个OpcodeEntry类型的结构体数组。



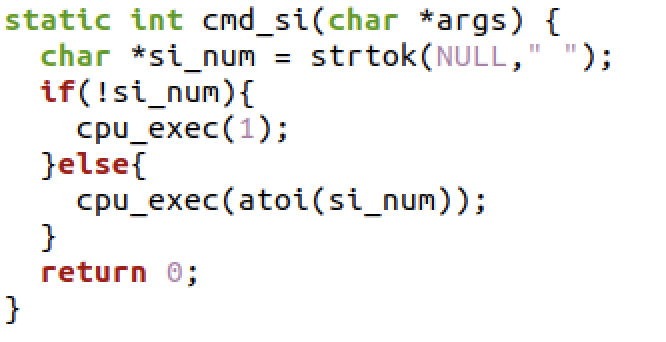
1. 程序的运行会通过操作系统上某个进程的形式来进行，因此某个程序的结束应该是以进程的终止状态来指示的。而进程主动终止的方式有主动调用exit()函数退出和执行到main函数返回命令时链接程序自动添加exit()函数两种方式。除了主动终止，进程收到kill命令以及出现异常执行的情况也会被终止。
2. NEMU是如何支持多种客户ISA的?

把ISA有关的部分抽象成一个接口, 其他ISA无关的部分调用这个接口, 不同ISA这个接口的实现不相同, 但是接口长得是一样的。在Makefile里面可以看到是通过$(ISA)来在编译的时候确定我们到底include哪个架构的头文件。

1. 我们可以把内存看作一段连续的存储空间, 而内存又是字节编址的, 在C语言中我们就很自然地使用一个uint8\_t类型的数组来对内存进行模拟. NEMU默认为客户计算机提供128MB的物理内存。但对于riscv32的物理地址从0x80000000开始. 因此我们需要记录其物理内存的起始地址, 将来CPU访问内存时, 我们会将CPU将要访问的内存地址映射到pmem中的相应偏移位置. 例如如果riscv32的CPU打算访问内存地址0x80001234, 我们最终会让它访问pmem[0x1234]. 这种机制有一个专门的名字, 叫地址映射。

## 单步执行

cpu\_exec()模拟了CPU的工作方式: 不断执行指令。传入的参数表示需要执行的指令条数。因此单步调试可以通过该函数来实现。重要的是我们需要在识别出此时是单步调试命令之后，识别出需要执行的指令数。通过strtok()函数来实现，如果没有解析到数字，则默认执行一条指令。编写单步调试的代码如下：

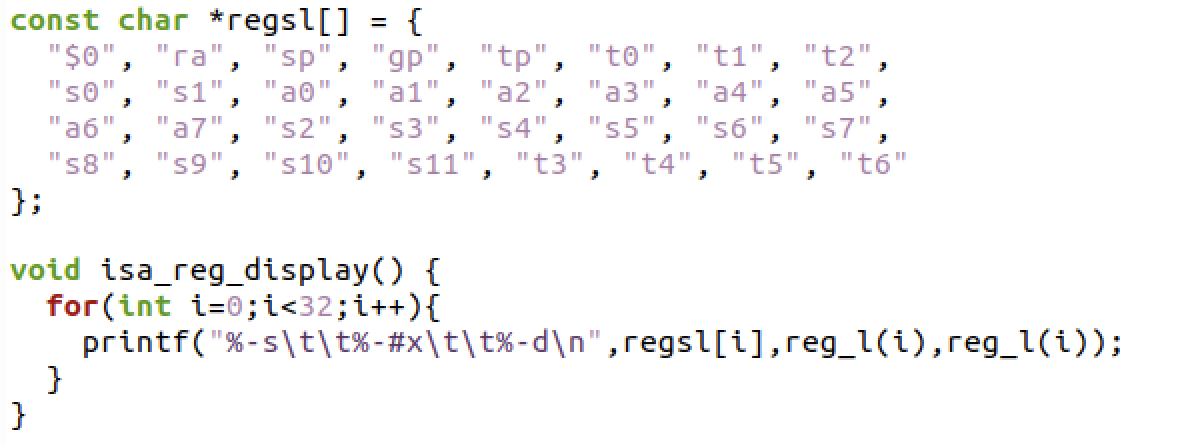


进入nemu界面进行单步调试测试，可以看到执行结果如下：

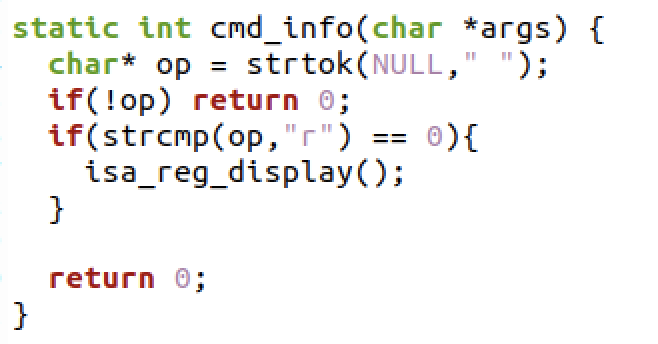


## 打印寄存器状态

寄存器的结构是ISA相关的，而简易调试器需要屏蔽ISA差异，所以在对应的nemu/src/isa/$ISA/reg.c文件中，需要为打印寄存器实现API为void isa\_reg\_display(void)。而在该函数上方，我们可以看到字符串数组regsl已经把对应的数组名字实现好给我们了，我们只需要调用printf进行打印即可，打印格式参考gdb。



由于info指令除了需要打印寄存器之外，我们还需要实现打印监视点，所以需要strtok()函数来拆分出后面的指令，如果是r则打印寄存器，只需要调用isa\_reg\_display()函数即可。打印监视点的功能在后面继续实现。

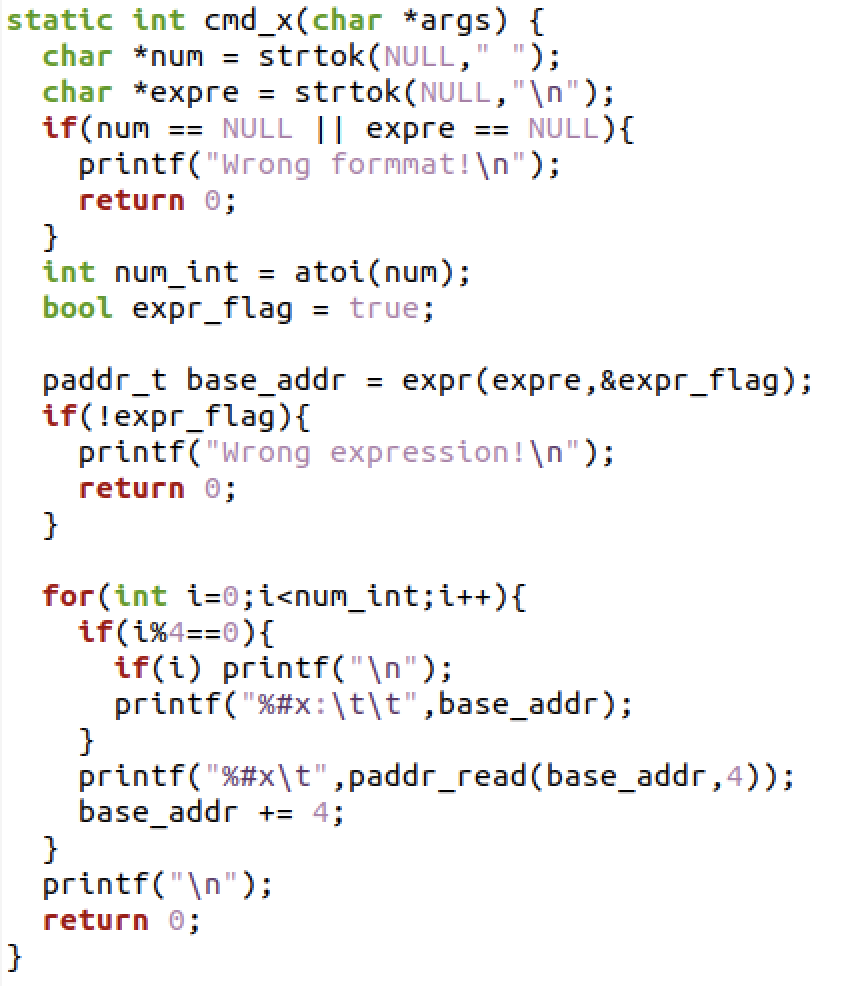


在nemu中进行测试结果如下：

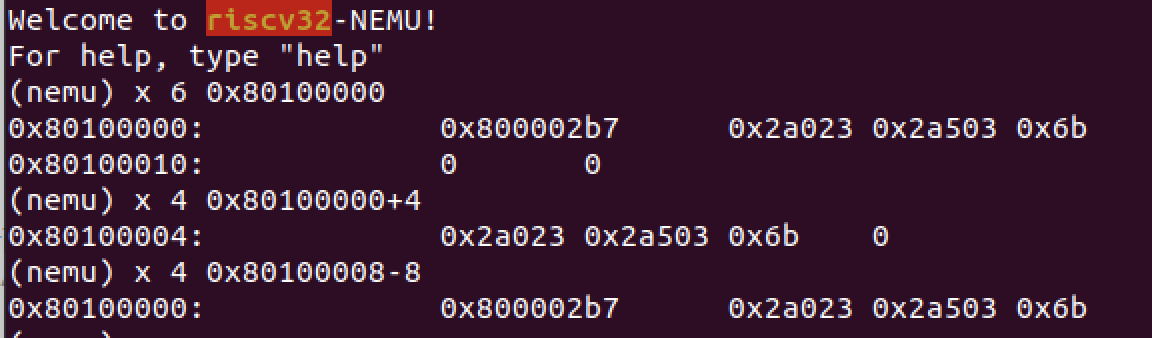


## 扫描内存

在这里我们先假设表达式求值函数已被实现完成。在解析出打印的内存个数num和表达式expre之后，把表达式求值并保存为paddr\_t类型，接着直接调用paddr\_read函数进行读取即可。



测试结果如下图所示：

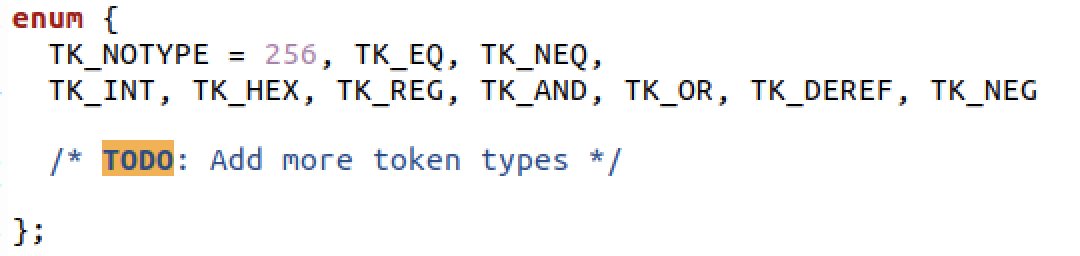


## 表达式求值

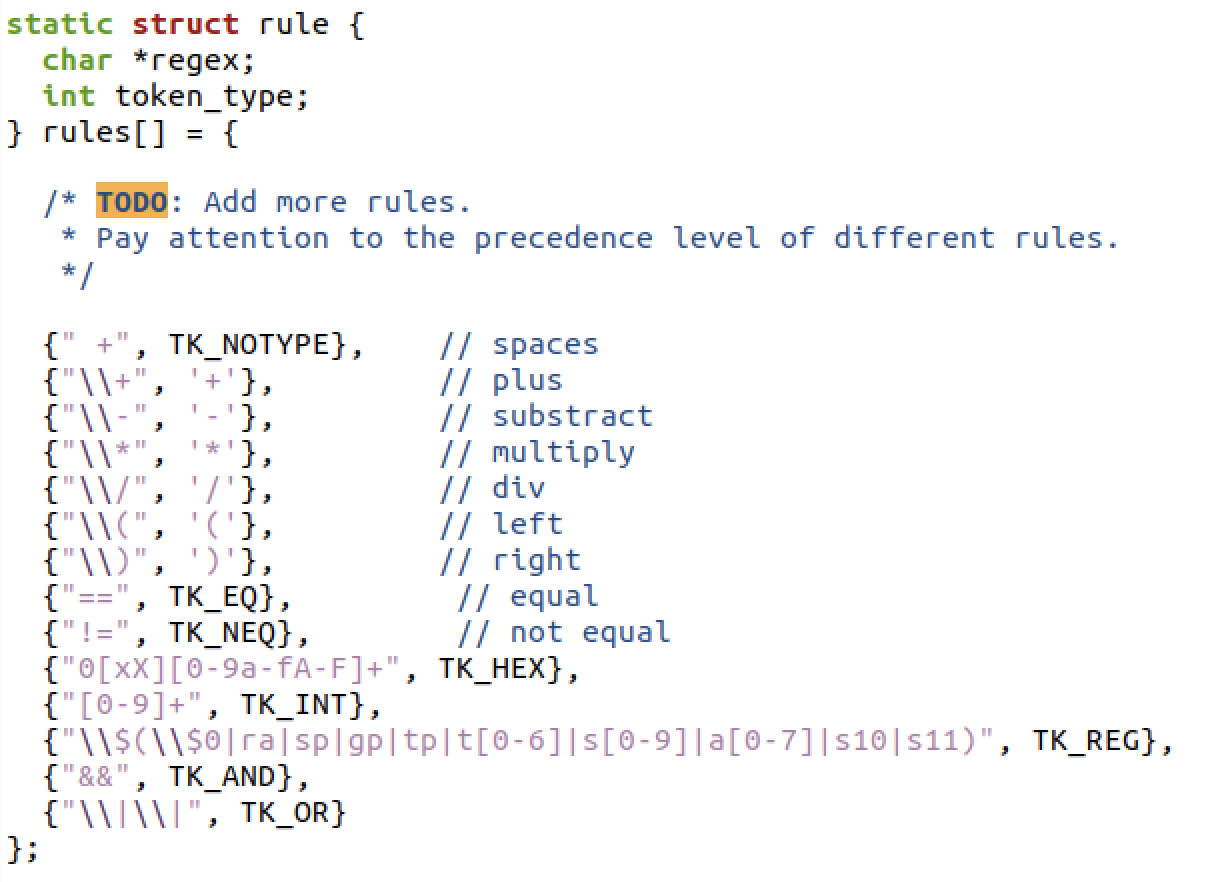
首先要实现算术表达式的词法分析，我们需要实现的功能如下：

* 为算术表达式中的各种token类型添加规则, 你需要注意C语言字符串中转义字符的存在和正则表达式中元字符的功能.
* 在成功识别出token后, 将token的信息依次记录到tokens数组中

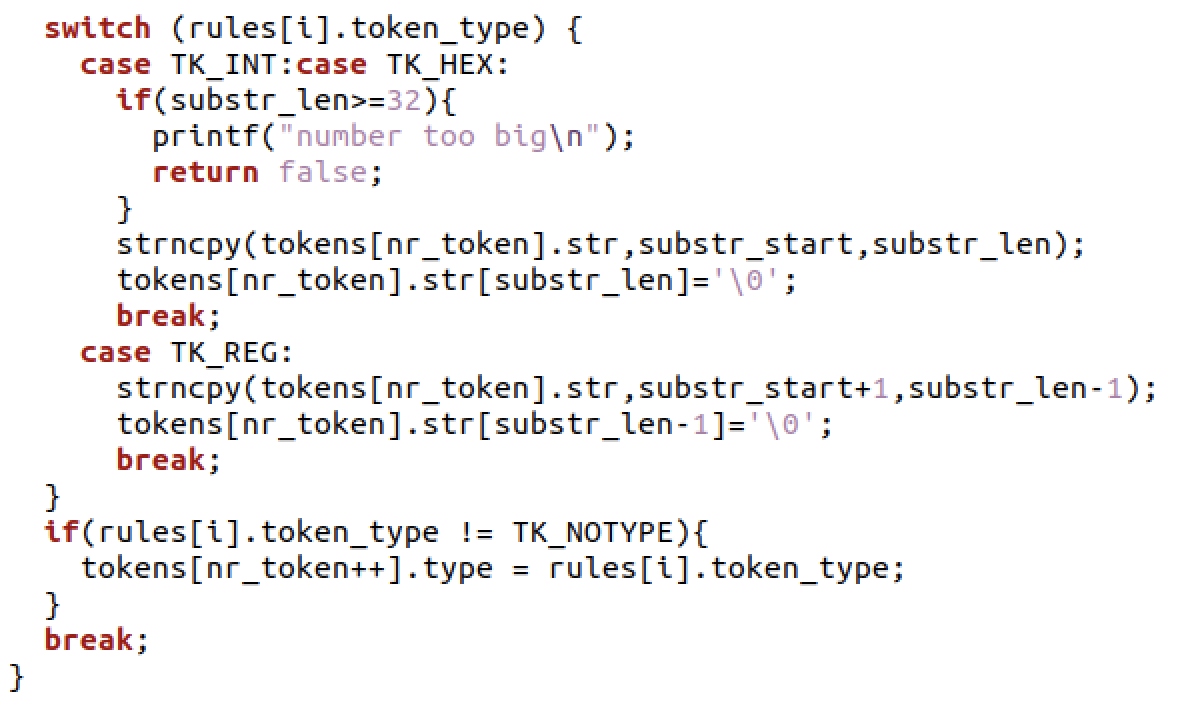
添加枚举类型如下：



添加正则表达式如下：



static bool make\_token(char \*e)函数用来将表达式中的符号识别出来，记录这些符号并放入tokens数组并用nr\_token来标记token的数量，而对于某些符号则要额外增加操作。根据当前符号的类型需要完成一些操作。除了无类型之外都需要赋类型；对整数值和寄存器类型需要记录具体值。需要补充的代码如下：



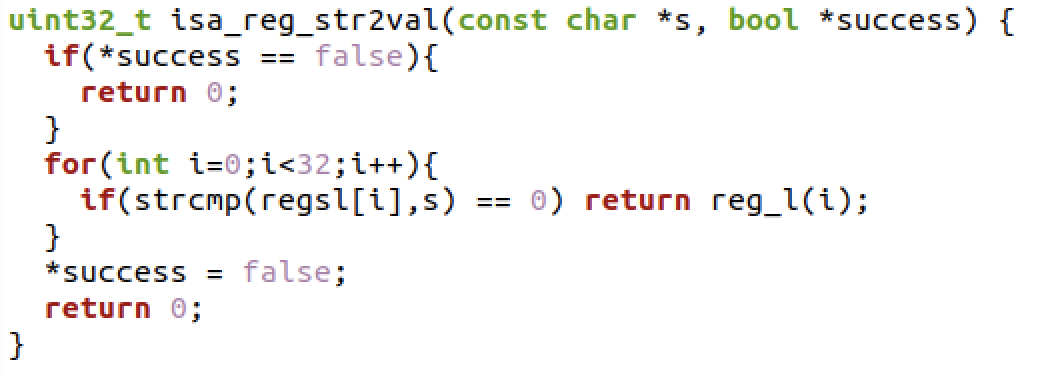
bool check\_parentheses(int p,int q)来判断表达式是否被一对匹配的括号包围着。当出现以下两种情况时，说明表达式错误根本无法求值：

1.在循环的过程中，出现了多余的右括号，也就是count < 0。

2.检查到最后的时候，count != 0，也就是括号总数不匹配。

uint32\_t eval(int p,int q,bool \*success)函数递归计算p、q之间的表达式的值。其主体框架在文档中已经有详细叙述。在这里略去具体实现。

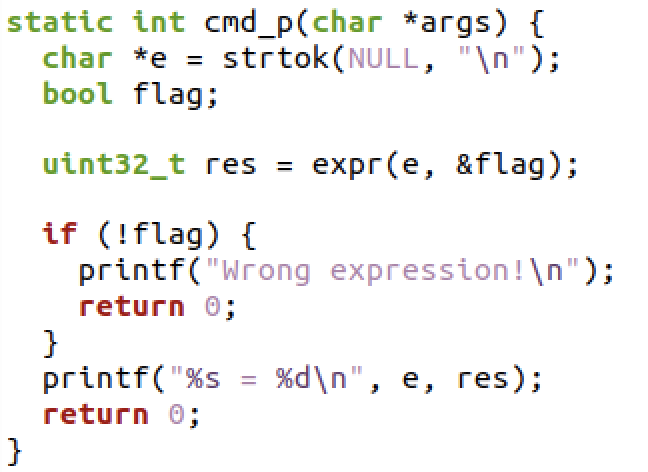
为了能读取出寄存器的值，需要调用isa\_reg\_str2val()函数来读取，为此在对应riscv32指令集下实现该函数如下：



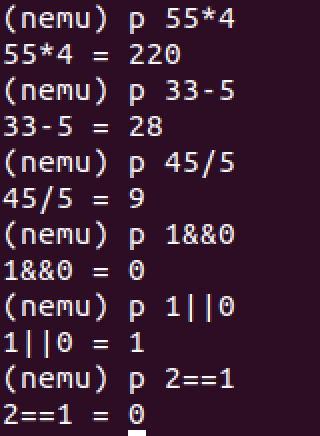
为了后续的监视点功能，还需要多实现解引用以及！符号功能。Expr函数的框架在文档里已经有列出，在这里列出具体实现如下：



为了在nemu中实现p指令，完成cmd\_p函数如下，可以看出核心代码值需要解析出表达式并调用即可。



简单的测试结果如下图：

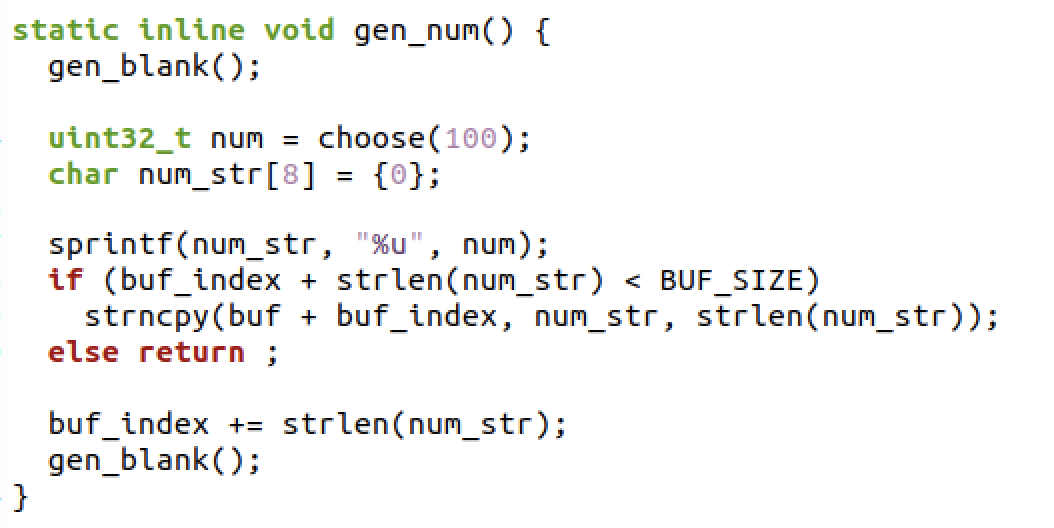


## 表达式生成器

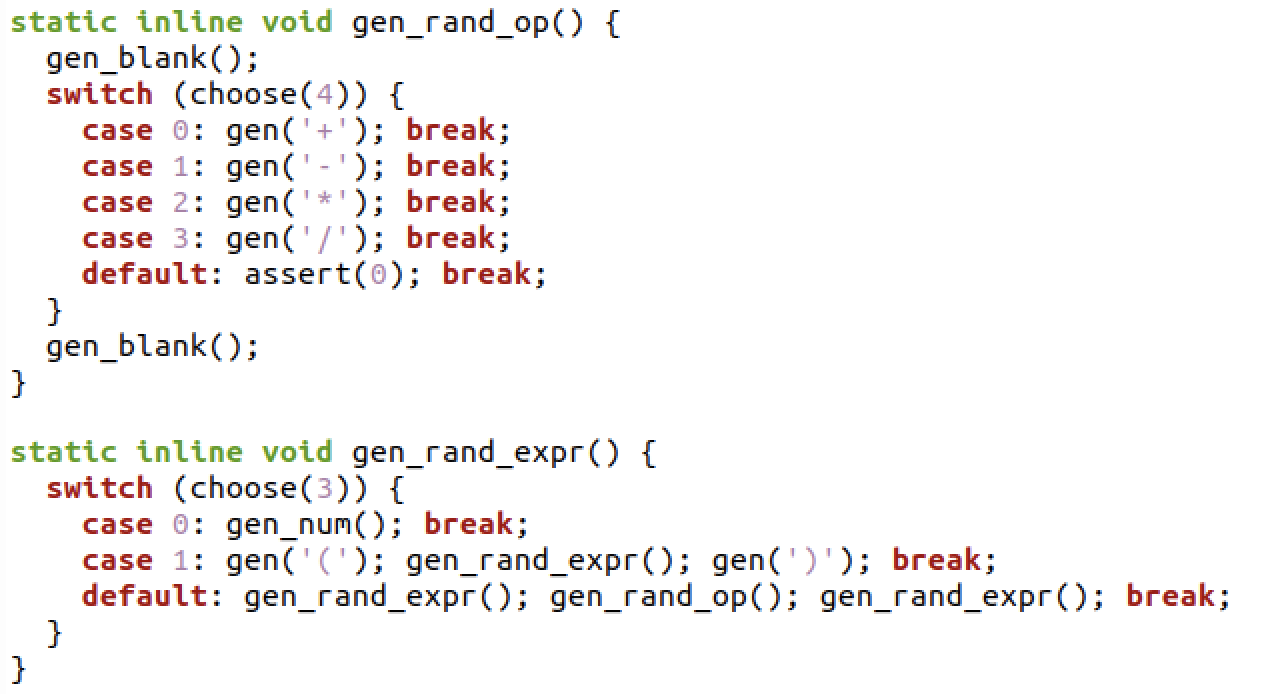
保证表达式进行无符号运算只需要将返回值设置为无符号类型即可。随机插入空格可以通过在生成数字前后和生成运算符前后增加随机生成若干个括号的函数实现。如果生成的表达式有除0行为, 直接assert并返回。

gen\_rand\_expr()的框架已经在文档中给出，具体实现也大致一致。按照框架实现gen()函数是一个内连函数，将传入的参数字符c放入buf数组中。实现choose(uint32\_t n)函数是生成一个0到n（不包括n）的随机数。实现gen\_blank()函数是随机生成小于一定数量的连续空格。实现gen\_num()函数是随机生成小于某个值的整数。实现gen\_rand\_op()函数是随机生成一个操作符。

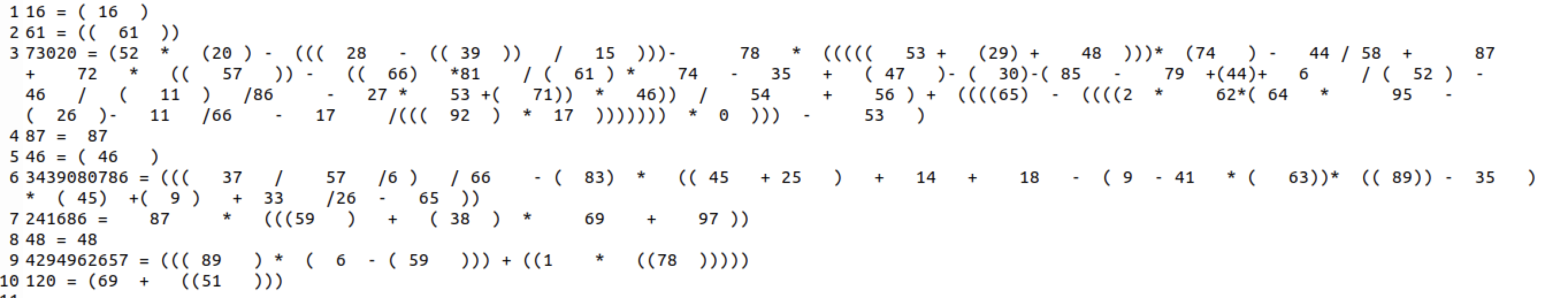
列出gen\_num()函数具体代码如下：



列出gen\_rand\_op()函数和gen\_rand\_expr()函数具体代码如下：



编译后运行，查看input输出文件如下所示。为了能更清晰阅读，在结果和表达式之间加了等号，可以看出已经实现了生成表达式并求值的功能。

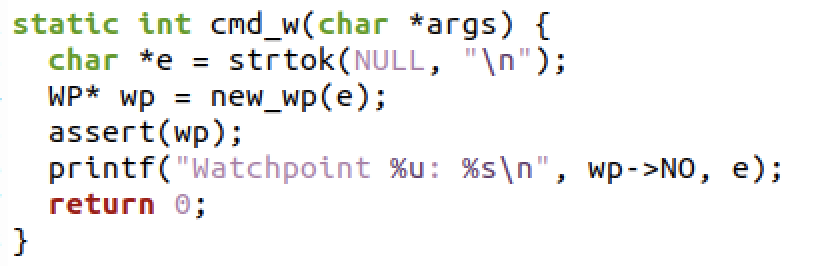


## 设置监视点

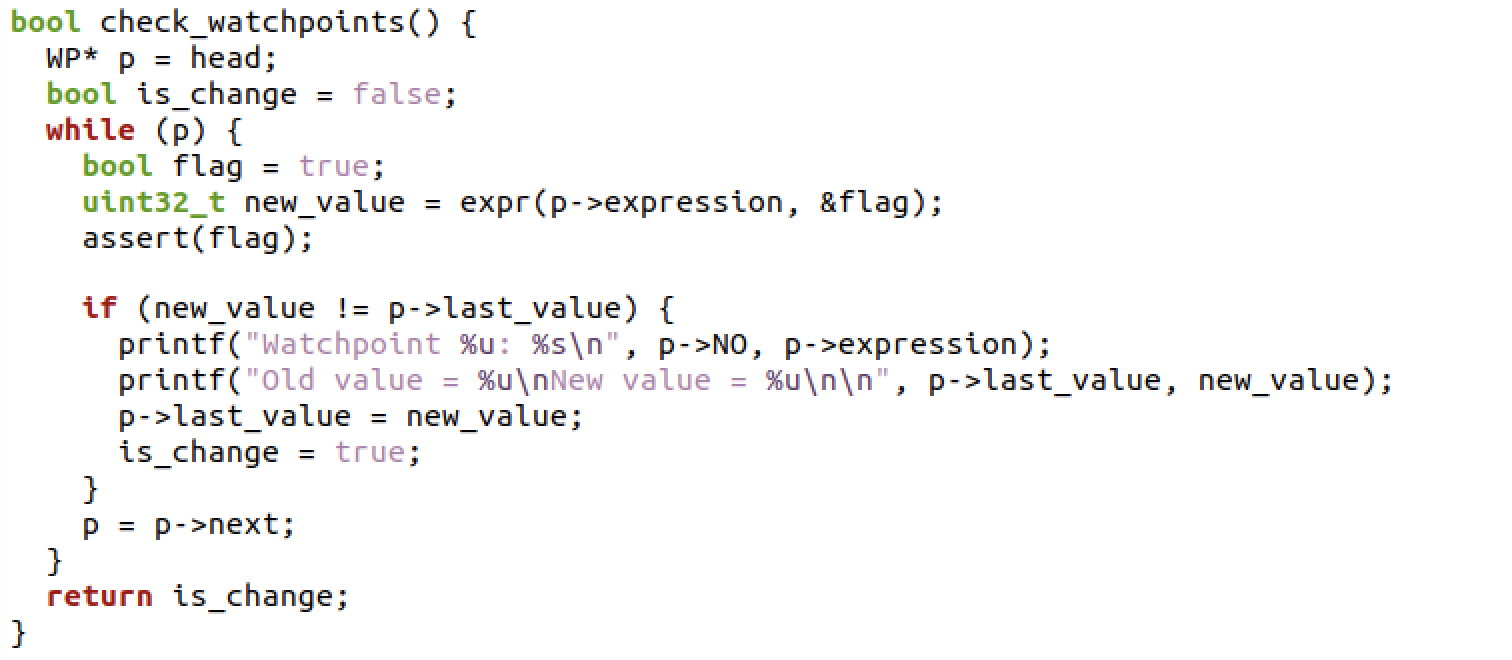
实现监视点watchpoint结构体需要新增两个属性，分别是char数组类型表达式expression和uint32\_t类型监视点旧值。

接着实现监视点池的管理，new\_wp()函数实现返回一个空闲的监视点，free\_wp()函数将传入的序号的监视点归还到free\_链表中。具体实现省略，就是简单的链表操作。

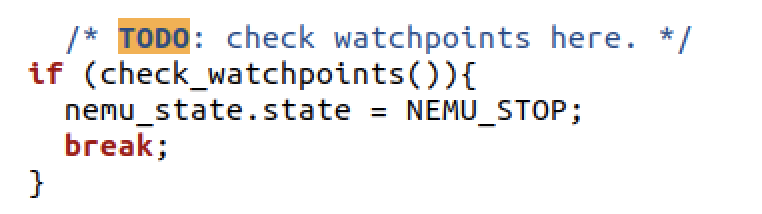
接着完成设置监视点的函数，主要就是调用了new\_wp()函数。



为了能实现在每句指令执行完后可以判断监视点的值是否改变，需要增加一个check\_watchpoints()函数来判断监视点的值是否改变了。



实现完check\_watchpoints()函数后，需要在cpu\_exec()函数中添加如下代码进行判断。若发生了变化, 程序就因触发了监视点而暂停下来, 需要将nemu\_state.state变量设置为NEMU\_STOP来达到暂停的效果.

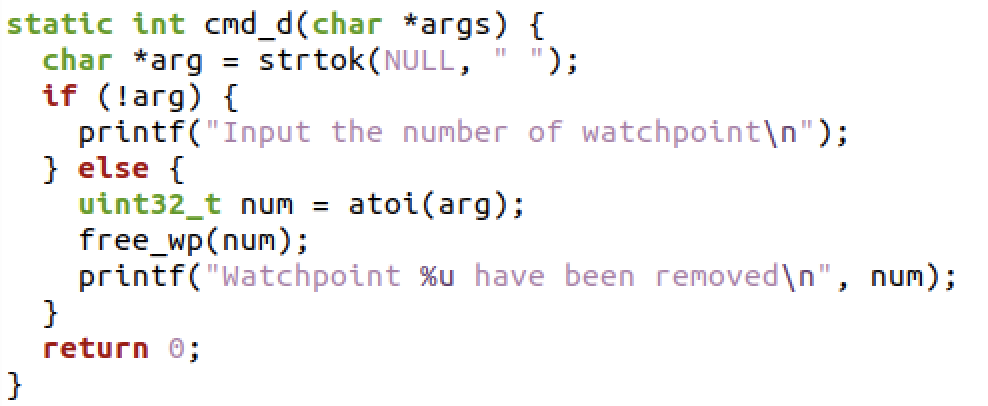


最后进行测试，达到如下效果：



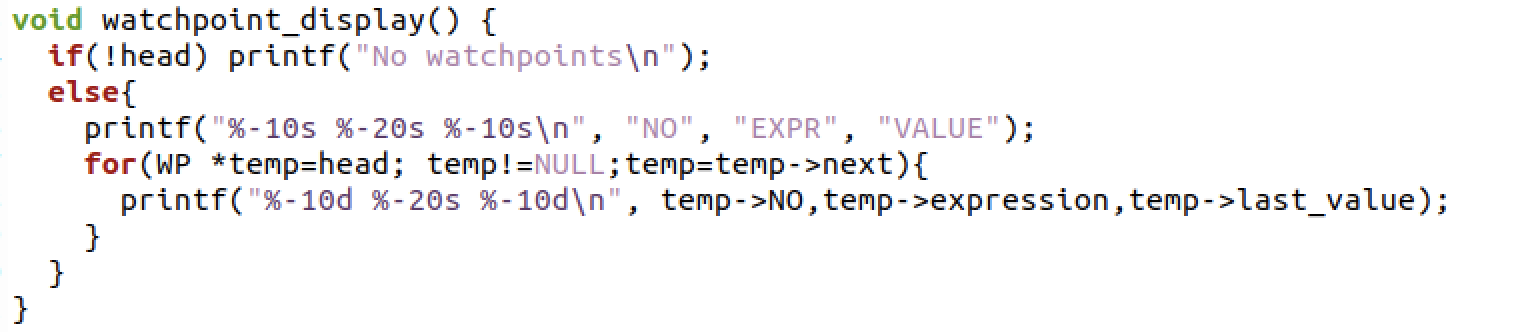
## 删除监视点

删除监视点的功能比较好实现，只需要将需要删除的监视点标号解析出来并且调用free\_wp()函数即可。

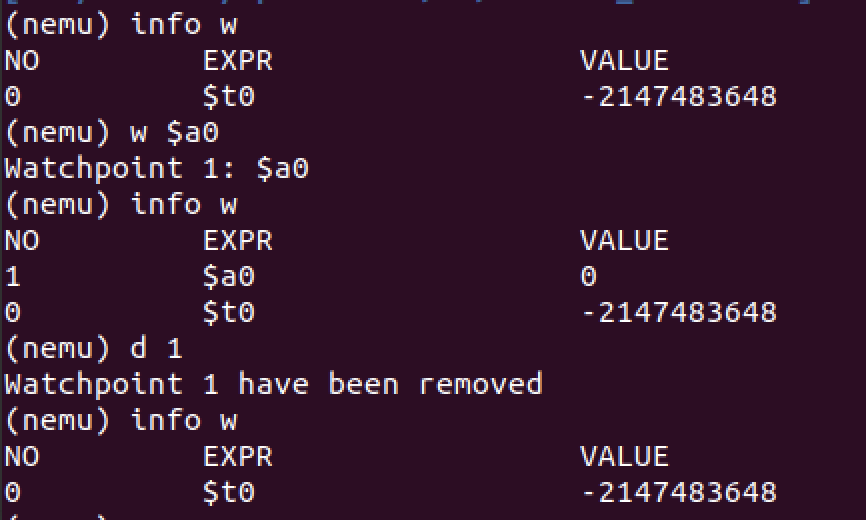


## 打印监视点

为了能够打印监视点，实现一个把head链表中的监视点打印一遍的函数watchpoint\_display()，并且在cmd\_info()函数解析到符号w时调用。具体实现如下：



打印监视点以及删除监视点的测试结果可以查看下图：



## 必答题

### ISA

我选择的ISA是riscv32.

### 理解基础设施

这个学期我将会在调试上花费：

500\*90%\*30\*20=270000s=90 小时

简易调试器可以帮助节省：

500\*90%\*(30-10)\*20 = 180000s = 50 小时

### 查阅手册

#### riscv32指令格式

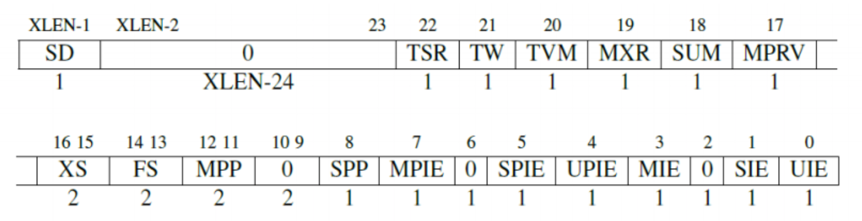
RISC-V 指令具有六种基本指令格式:  
R 类型指令:用于寄存器 - 寄存器操作;  
I 类型指令:用于短立即数和访存 load 操作; S 类型指令:用于访存 store 操作;  
B 类型指令:用于条件跳转操作;  
U 类型指令:用于长立即数操作;  
J 类型指令:用于无条件操作;

#### LUI指令的行为

LUI是高位立即数加载指令，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

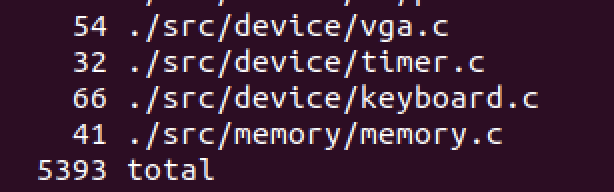
mstatus控制状态寄存器保存全局中断使能，以及许多其他的状态。其结构如下图所示：



### shell命令

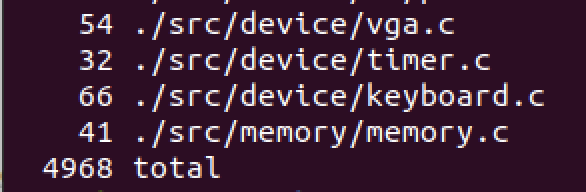
在nemu目录下用以下命令可以获取全部的代码行数：

find . -name “\*.[hc]” | xargs wc -l

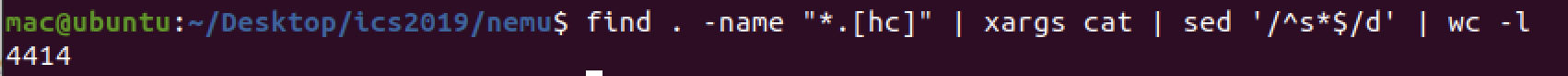


可以看到总代码行数是5393.

切换到pa0分支，输入同样的命令可以看到框架代码总共是4968行：



在pa1分支下，去除空行后，总共代码是4414行。



### 使用man

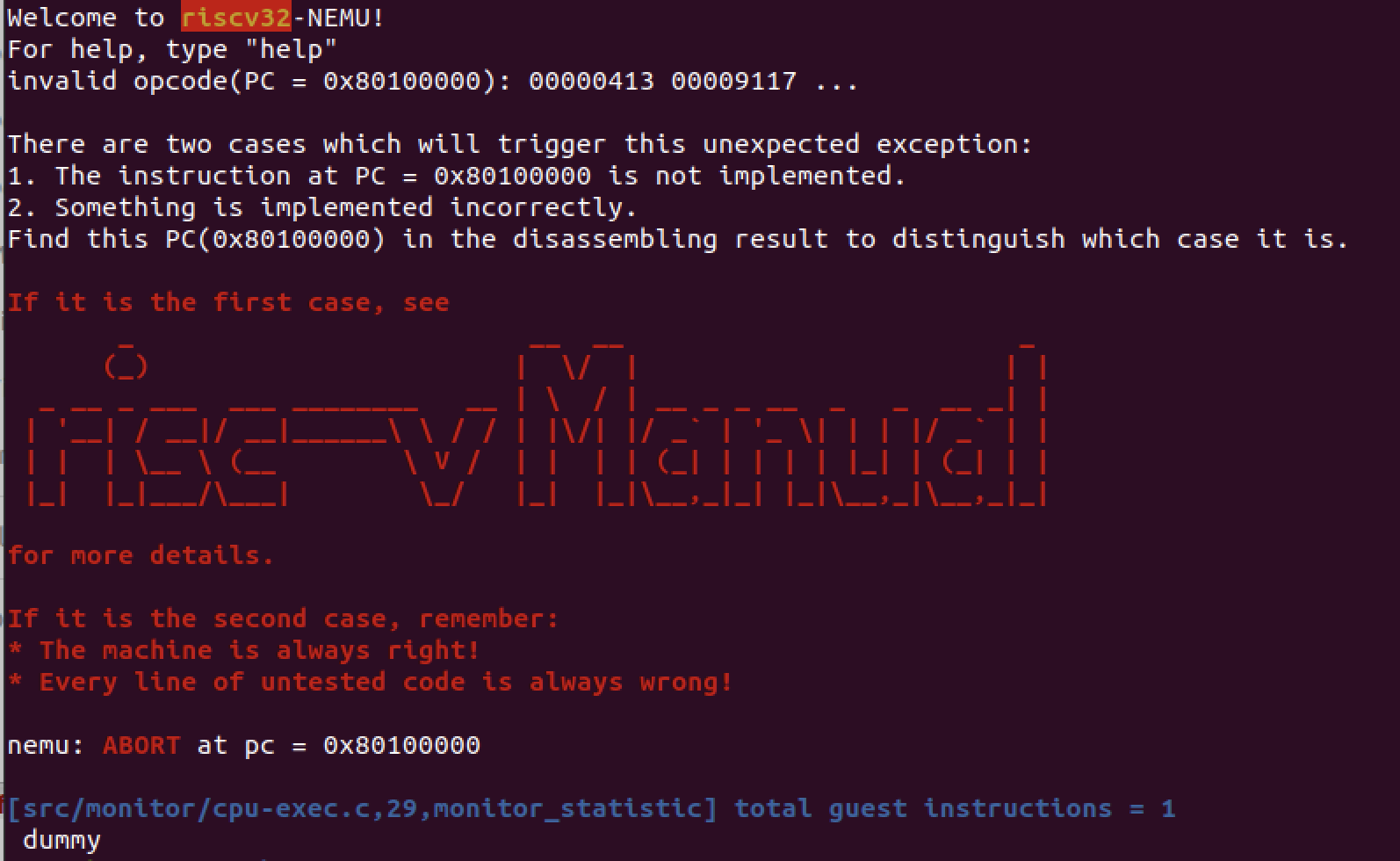
-Wall 的作用是打开 gcc 所有警告。  
-Werror 的作用是要求 gcc 将所有警告当成错误处理。

使用-Wall、-Werror的好处是可以防止代码的不规范，有利于发现代码中不严谨之处，保证代码在编译期是正确的，从而减少运行时因忽略 warning 所导致的错误。

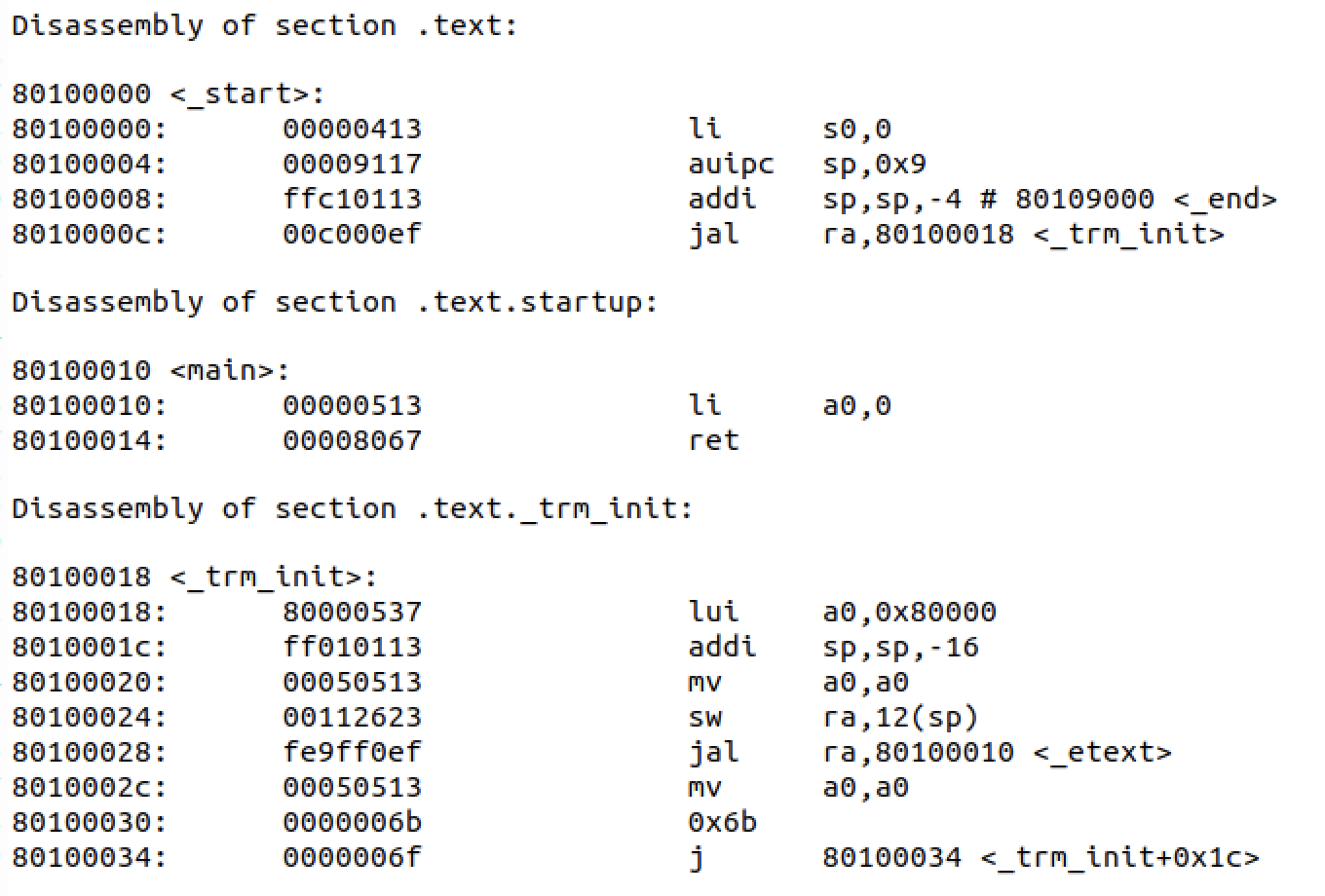
# 简单复杂的机器：冯诺依曼计算机系统

## PA 2.1

编译dummy程序, 并启动NEMU运行它。可以看到在未完成指令编写之前，会出现错误提示：

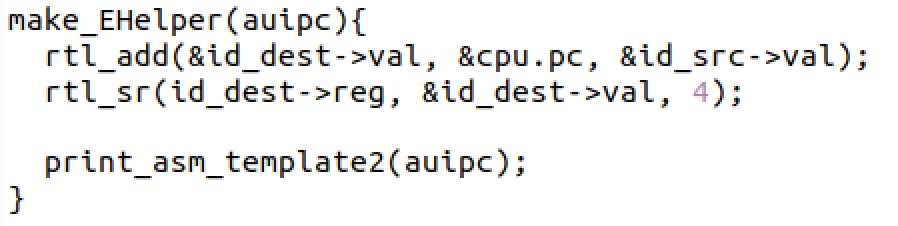


在第一次运行完之后，可以通过产生的build文件夹中的dummy-riscv32-nemu.txt 文件中的反汇编代码来查看所需要实现的指令：

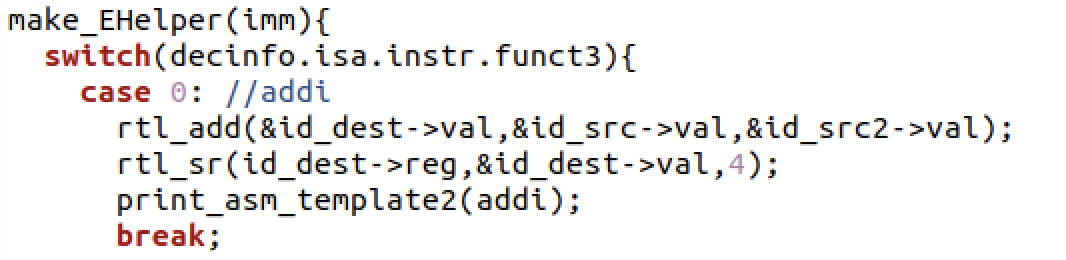


可以看到我们需要实现的指令有li、auipc、addi、jal、ret、lui、mv、sw、j，其中lui和sw指令已经实现完成，而ret指令经过比对是jalr指令。

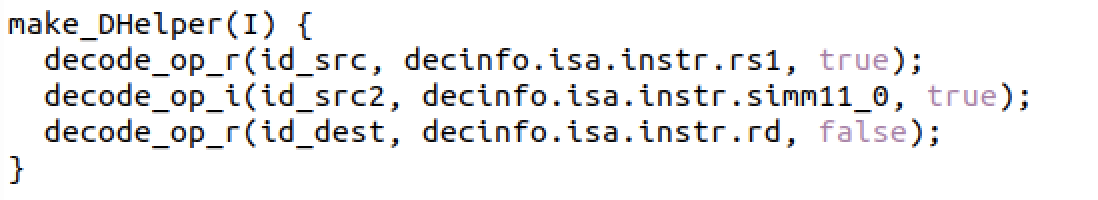
auipc指令是将高位立即数加pc，其指令格式和lui一致，按照lui指令执行辅助函数来编写auipc的执行辅助函数如下：



li指令是一条riscv32伪指令，把立即数imm加载到rd寄存器中。根据反汇编代码来看，li、addi、mv三条指令的opcode是一致的，都是I型指令，再根据funct3都为零来看，其实都是一条指令，也就是addi。其执行辅助函数实现如下：

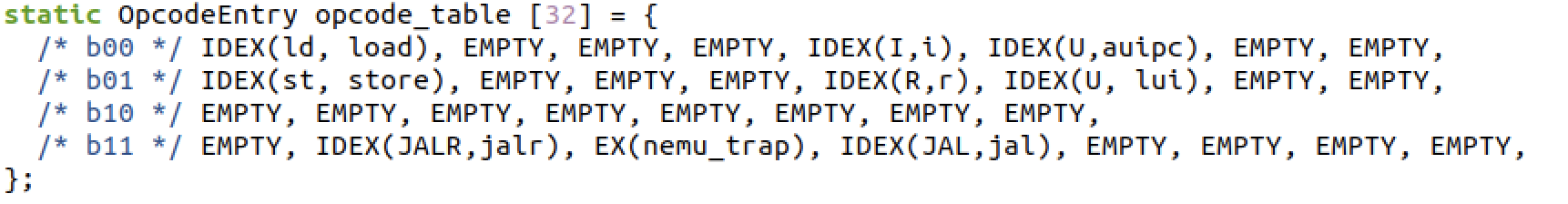


除了执行辅助函数，我们还需要实现译码辅助函数。实现I型指令的译码辅助函数如下：

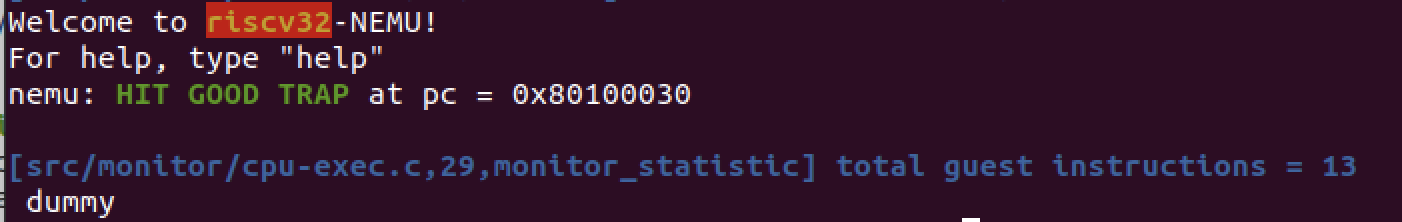


接着实现jal和jalr指令的执行辅助函数和译码辅助函数，在此省略具体实现代码。

接下来把实现的函数在分别的头文件中声明，并且实现opcode\_table. opcode\_table数组其实就是我们之前提到的译码查找表, 这一张表通过操作码opcode来索引, 它记录了每一个opcode对应指令的译码辅助函数, 执行辅助函数, 以及操作数宽度.



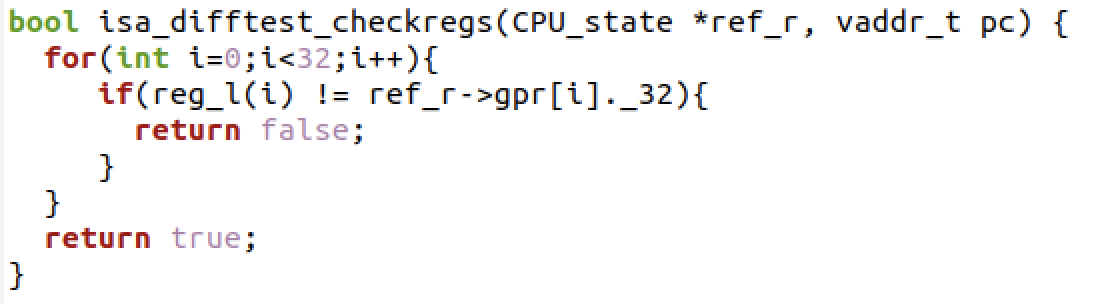
最后编译运行可以看到结果：



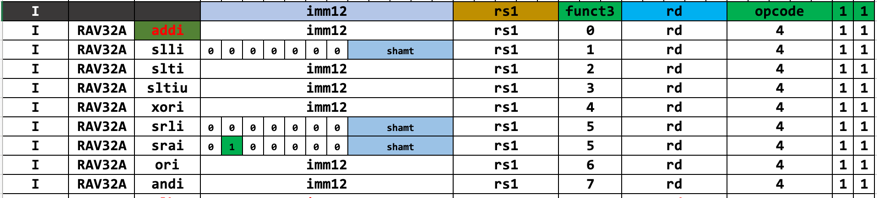
## PA 2.2

在完成dummy运行的基础之上，为了能够实现更多的指令运行，我们需要实现其余的riscv32指令集中的I型、R型和B型指令。

为了能够更好的调试，防止error的进一步传播以及回溯找到fault，我们实现diff-test.c文件。实现isa\_difftest\_checkregs()函数如下：



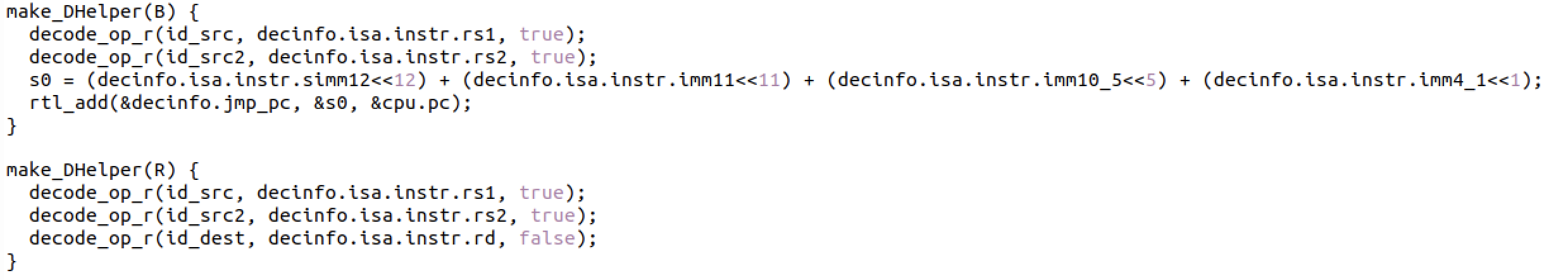
其中I型指令是在addi指令基础上进行扩展即可，目前实现的I型指令如下图：



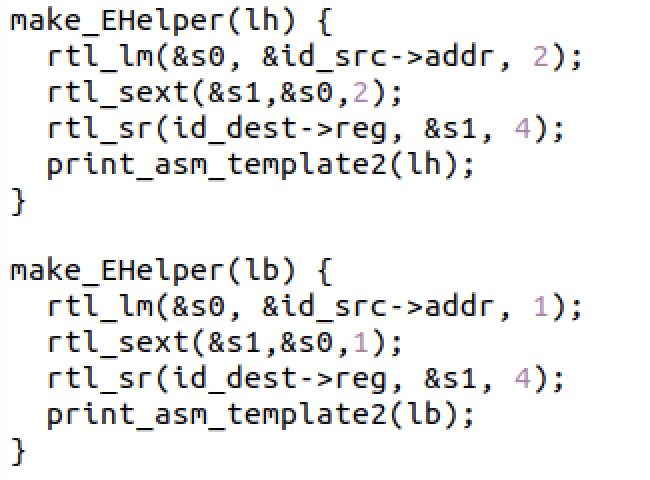
可以看出I型指令是通过funct3的值来进行区别的，而srli和srai还要再进一步判断funct7。按照指令的语意实现并调用对应的rtl基础指令即可。R型、B型指令也同理。

对于B型指令，其执行辅助函数定义在control.c中，通过rtl\_jrelop传入不同的指令参数来实现。

实现R型指令和B型指令的译码辅助函数如下：



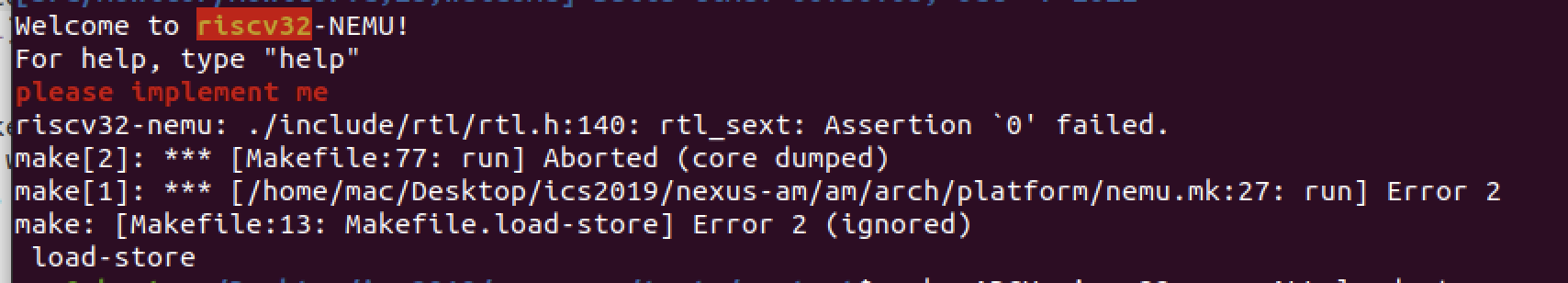
查看ldst.c中的代码，发现load的执行辅助函数中未实现lh和lb指令，对此需要完成这两个指令的执行。



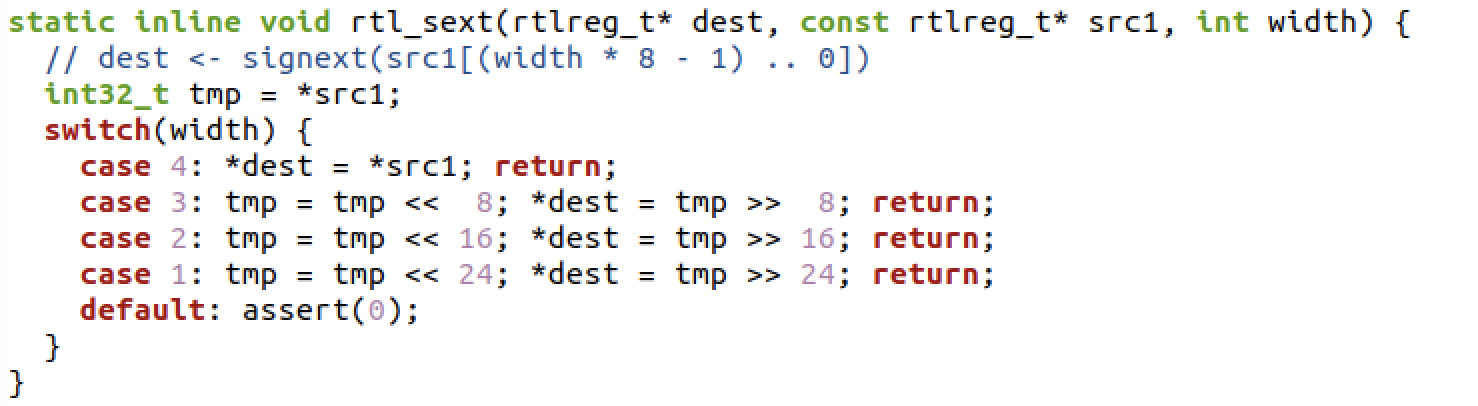
增加了lh、lb的执行辅助函数之后，需要在all-instr.h中进行声明。除此之外，还需修改load\_table。在opcode等于0时，funct3等于0时为lb指令，funct3等于1时为lh指令。故而修改了load\_table数组之后，可以正确调用lh、lb指令。



实现完ldst.c的代码之后再进行编译运行，结果发现出现了错误：

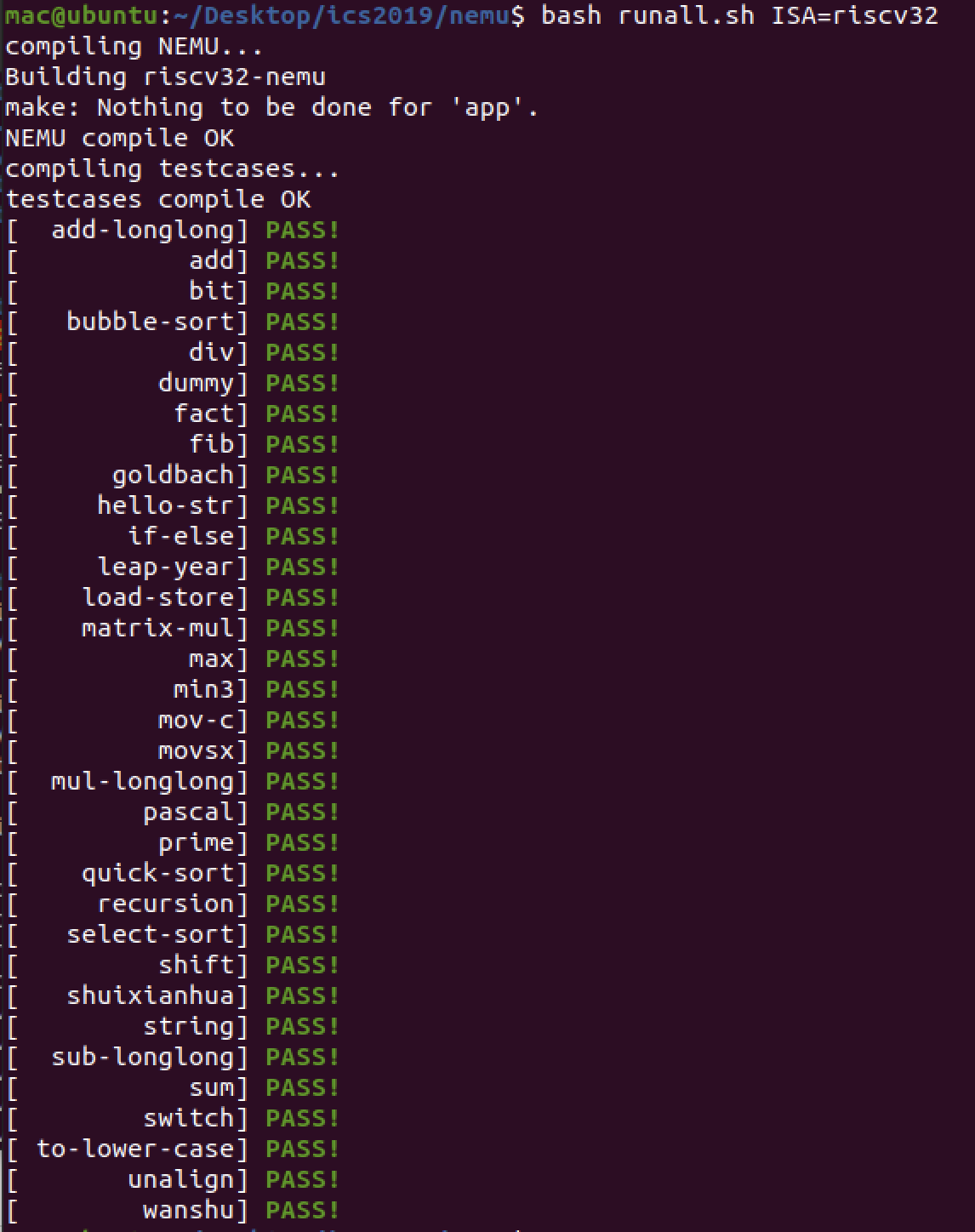


根据错误提示，我们可以知道是因为有函数没有实现，调用了TODO()而导致的。查看具体错误是在./include/rtl/rtl.h中的rtl\_sext函数未被实现，而在实现lh、lb的执行辅助函数中都已经被调用，所以我们需要实现该函数：



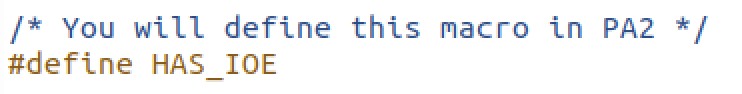
在测试文件string和hello-str中需要一系列库函数，在stdio.c中需要实现printf、vsprintf、sprintf等输出函数，在string.c中需要实现strlen、strcpy、strcmp、memcpy等字符串处理函数。字符串处理函数都比较简单。而输出函数则比较复杂，参考源码以及库stdarg之后进行编写。vsprintf 需要对传入的字符串 fmt 以及可变参数列表 va\_list 进行处理，当在 fmt 字符串中读取到%d、%x、%s 的子串时， 调用 va\_arg 函数将 va\_list 中对应类型的参数，然后转换成字符串传入输出字符 串 out 中，这样就完成了 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。

最后，使用一键回归测试来测试所有程序得到结果如下：



## PA 2.3

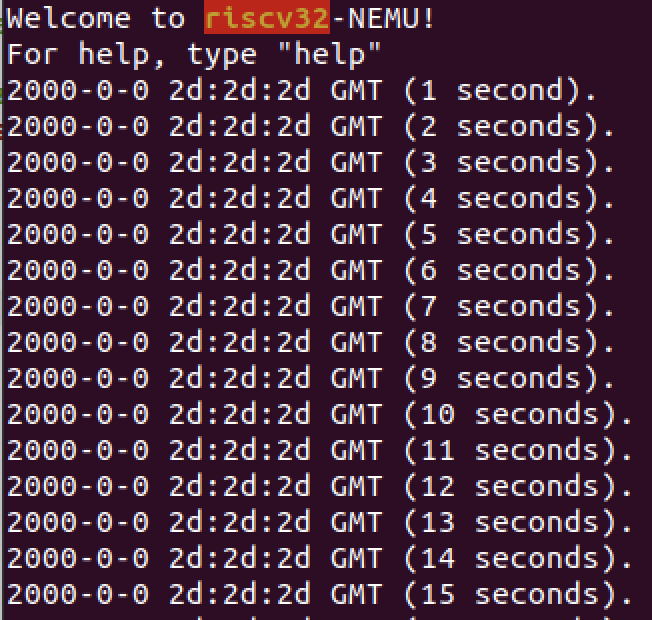
首先在./include/common.h中把HAS\_IOE宏给定义出来：



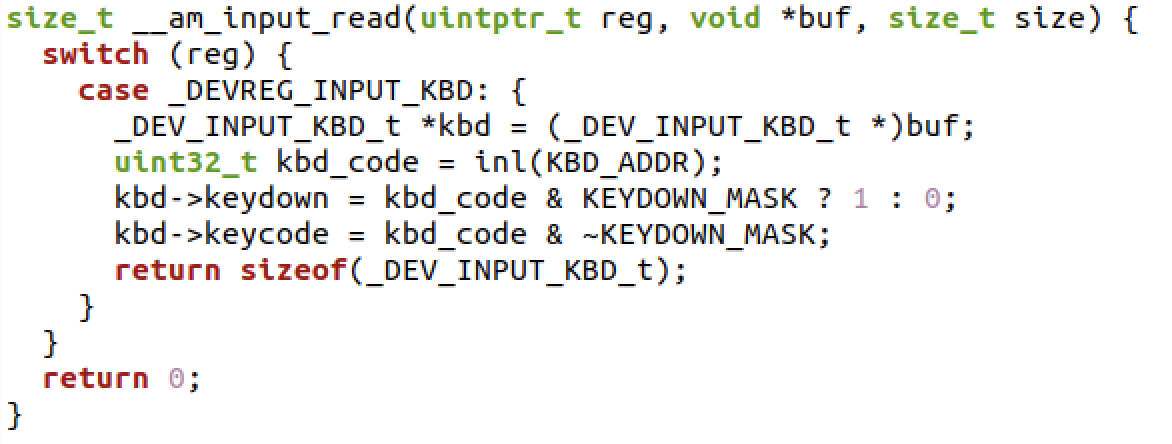
在PA2.2中已经完成printf函数的编写，之间运行打印的测试程序：



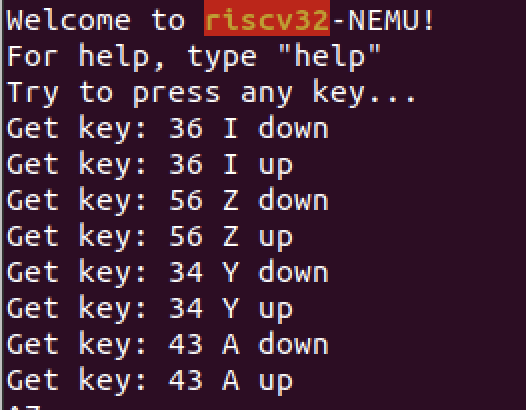
为了完成时钟的功能，我们需要在nexus-am/am/src/nemu-common/nemu-timer.c中实现\_DEVREG\_TIMER\_UPTIME的功能。在启动时\_\_am\_timer\_init函数通过 RTC\_ADDR 获取启动时间并保存至一个静态变量启动时间中，运行时钟功能时通过地址 RTC\_ADDR 获取当前时间，将 uptime->hi 设为 0，uptime->lo 设为当前时间与启动时间的差值，即可完成时钟功能。运行并测试如下所示：



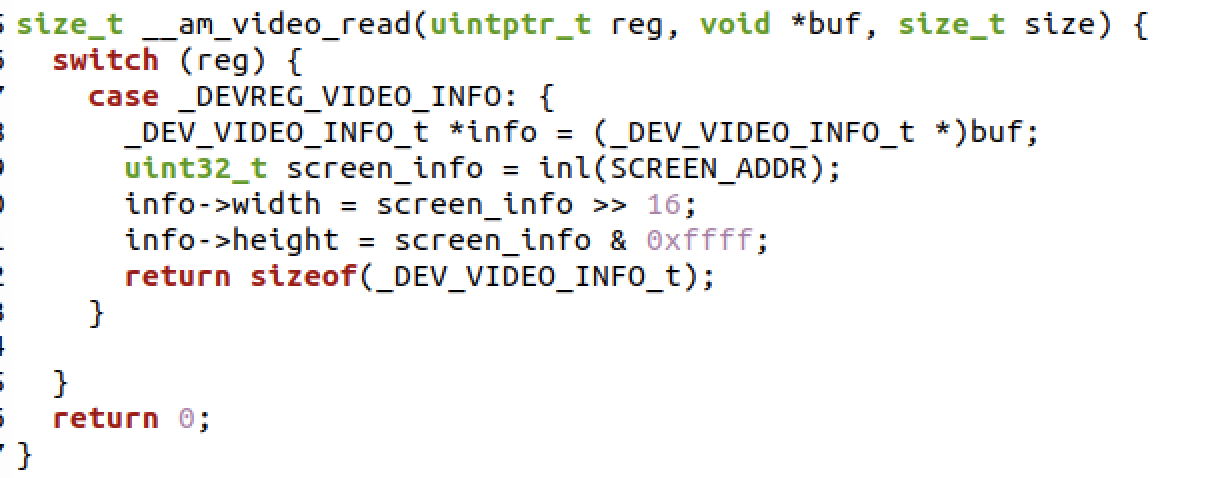
实现键盘的功能，在nexus-am/am/src/nemu-common/nemu-input.c中实现\_DEVREG\_INPUT\_KBD的功能。通过地址KBD\_ADDR可以获取键盘的按键信息，接着与KEYDOWN\_MASK进行操作运算得到kbd->keydown和kbd->keycode。



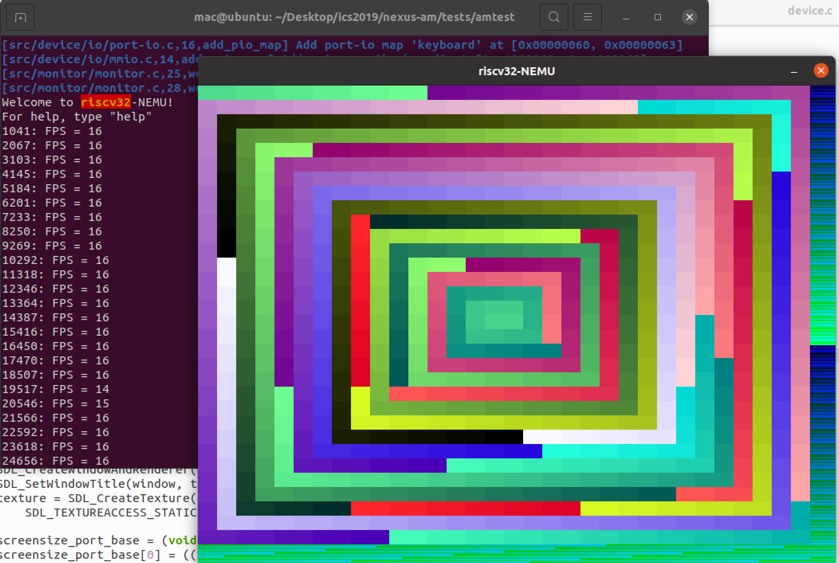
编译测试得到如下结果：



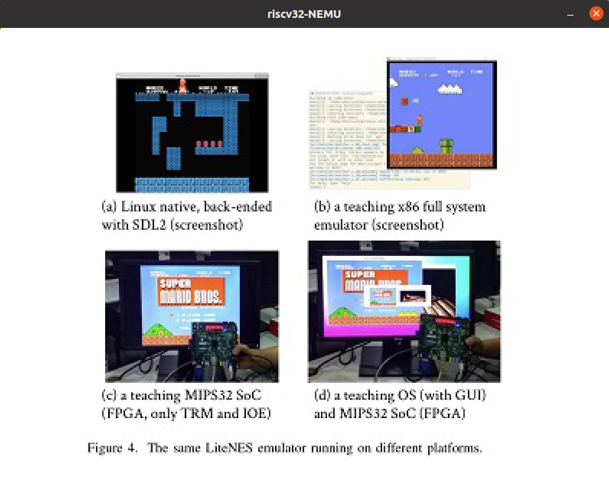
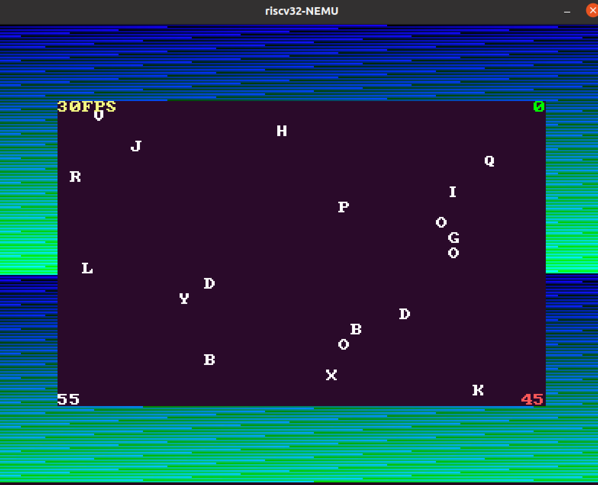
最后实现VGA部分的功能。首先需要实现\_\_am\_video\_read和\_\_am\_video\_write函数，实现两个抽象寄存器\_DEVREG\_VIDEO\_INFO和\_DEVREG\_VIDEO\_FBCTL。\_\_am\_video\_read是从屏幕地址读取出长、宽并保存。\_\_am\_video\_write函数向屏幕(x, y)坐标处绘制w\*h的矩形图像. 图像像素按行优先方式存储在pixels中, 每个像素用32位整数以00RRGGBB的方式描述颜色。\_\_am\_video\_read函数的实现代码如下所示：



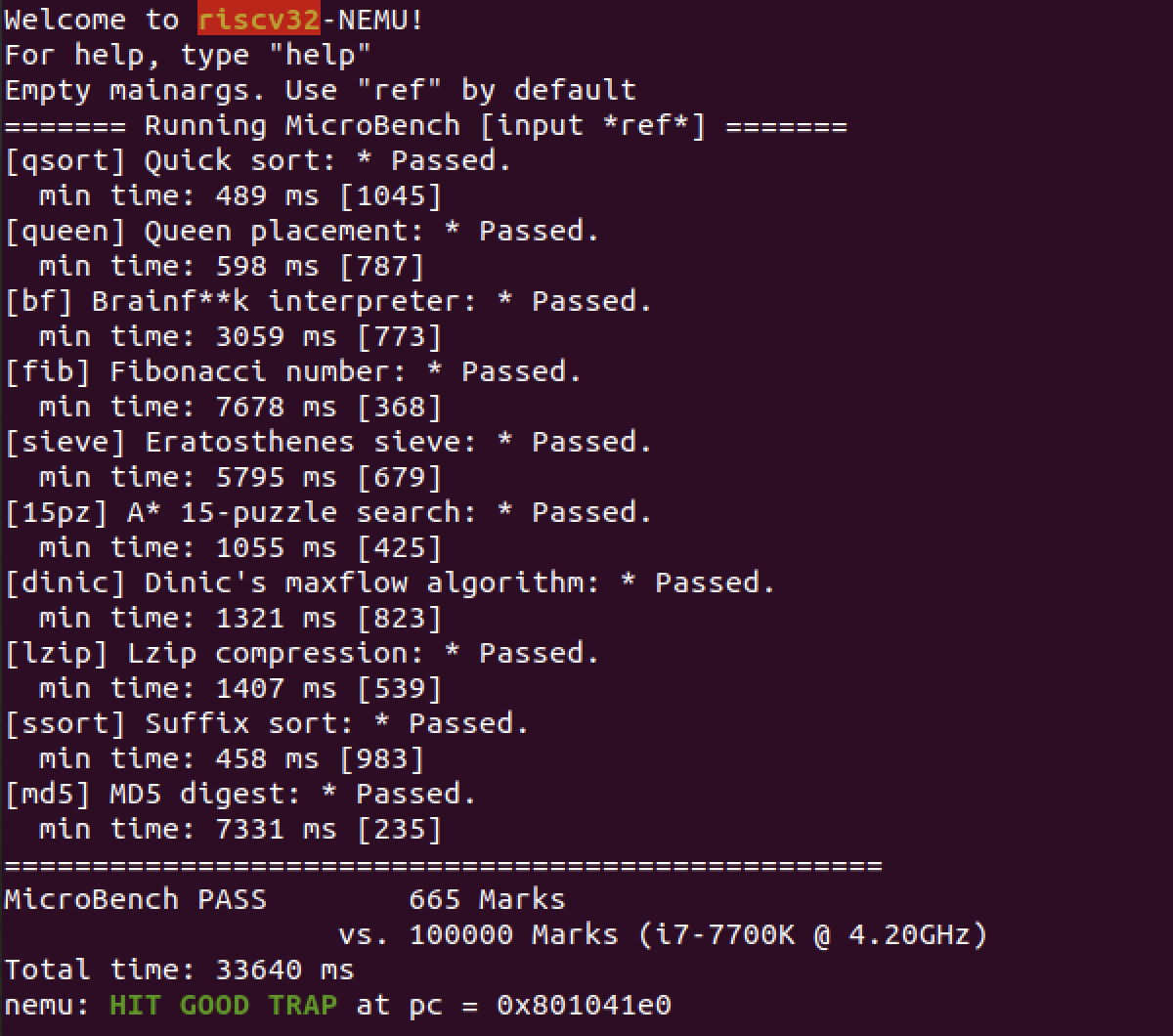
在./src/device/vga.c中的vga\_io\_handler需要被实现，我们只需要判断当前是否是写模式，如果是则调用update\_screen()函数即可。最后测试如下，运行时可以看到颜色条在逆时针旋转：



实现以上代码之后就可以在riscv32-nemu下运行幻灯片播放和打字小游戏了，运行结果如下图所示：



运行跑分程序，可以看到最终结果为665分:



## 必答题

### 整理一条指令在NEMU中的执行过程.

isa\_exec()函数拿到参数decinfo.seq\_pc且被调用，通过instr\_fetch函数获得当前pc对应内存位置中的指令。接着调用idex译码执行函数，其中调用的参数根据之前获得的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。

随后调用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；调用执行辅助函数执行指令，通过相关的基础rtl指令进行相关操作，计算、加载、保存等。

最后调用update\_pc函数来更新pc的值。

### 编译与链接——RTL指令函数

RTL指令函数单独去掉 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。

以rtl\_li函数为例。两者都去掉时，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，就会出现错误。

### 编译与链接

1. 添加后，使用 grep 命令查看，共有 1 个 dummy 实体。

2. 继续添加后，重新编译运行，使用 grep 命令，共有 2 个 dummy 实体。

static关键字表示变量只有在当前文件可以被识别，volatile表示这处代码不会被编译器优化， 而 debug.h 通过include包含了 common.h ，所以重新声明的变量不会覆盖之前的dummy。

3. 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。

### 了解Makefile

在终端中输入 make 命令后，会将 makefile 文件中第一个目标文件作为最终的目标文件。

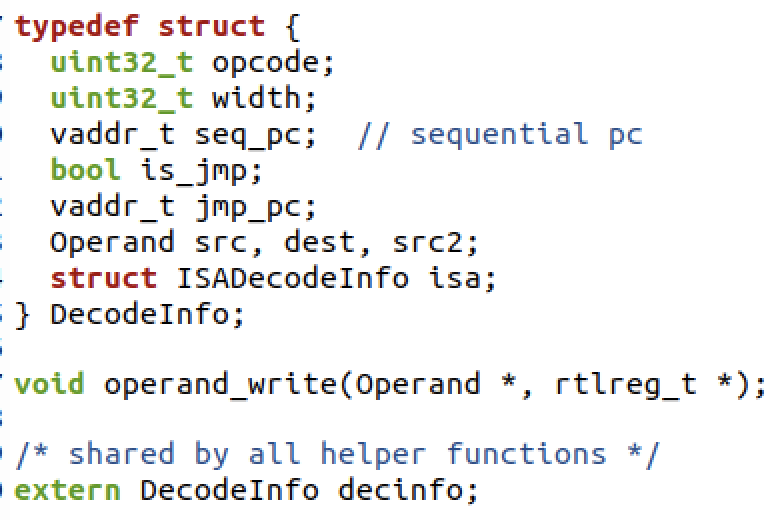
如果文件不存在，或是文件所依赖的后面的.o 文件的修改时间比这个文件晚，就会重新编译；如果目标文件依赖的.o 文件也不存在，就根据这个.o 文件的生成规则生成，读取所有需要编译的.c的文件并将其编译为.o文件。

编译好所有的文件后进行链接，生成最后的目标文件。

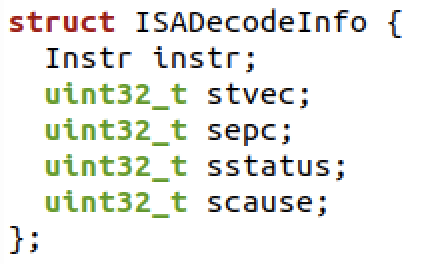
# 穿越时空的旅程：批处理系统

## PA 3.1

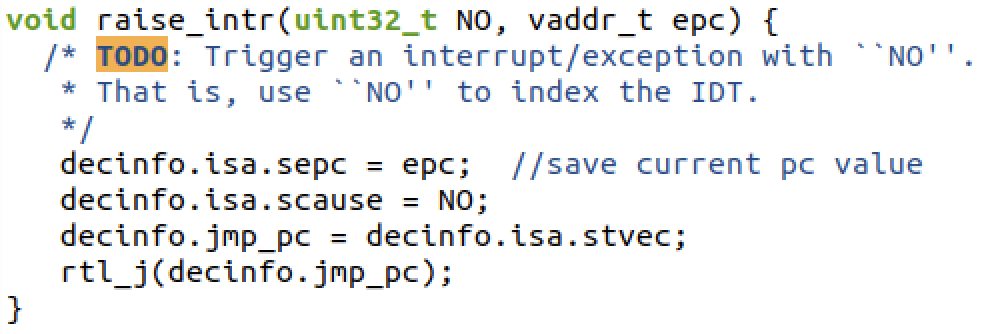
观察decinfo的结构体类型DecodeInfo，可以看到其中具体和ISA实现相关的结构体ISADecodeInfo，因此我们知道控制状态寄存器应该是保存在ISADecodeInfo 结构体中的。



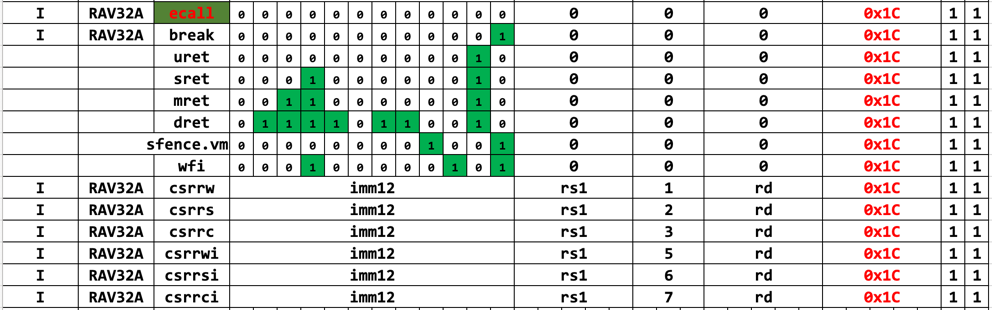
我们需要四个控制状态寄存器，分别是stvec寄存器来存放异常入口地址、sepc寄存器来存放触发异常的PC、sstatus寄存器来存放处理器的状态、scause寄存器来存放触发异常的原因。扩展riscv32架构的ISADecodeInfo结构体如下图所示：



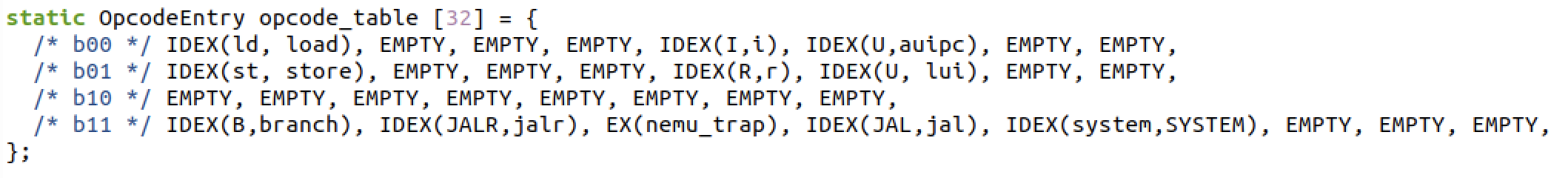
接着需要在 intr.c文件中进行 raise\_intr 函数的编写。raise\_intr 函数用于模拟异常响应机制:将当前 PC值保存到 sepc 寄存器，在 scause 寄存器中设置异常号，从 stvec 寄存器中取出异常入口地址，调用RTL基础跳转指令跳转到异常入口地址。raise\_intr 函数实现如下：



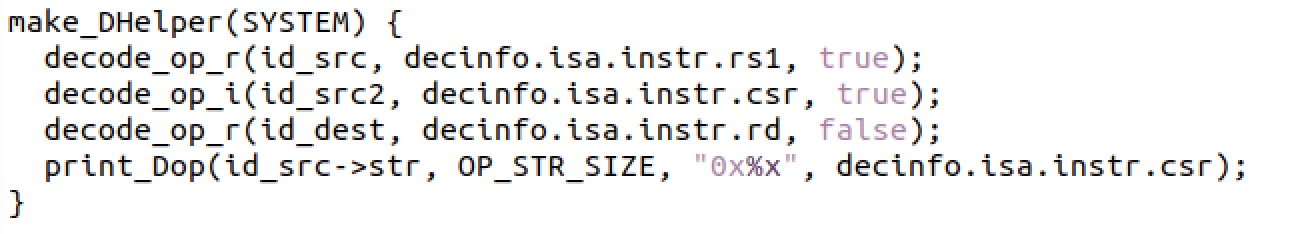
实现与系统调用有关的新指令，查看相关文档，需要实现的指令为I型指令，如下表所示：



可以看到其opcode都为0x1C，也就是28，按照对应顺序填入opcode\_table，并实现对应的译码辅助函数和执行辅助函数：

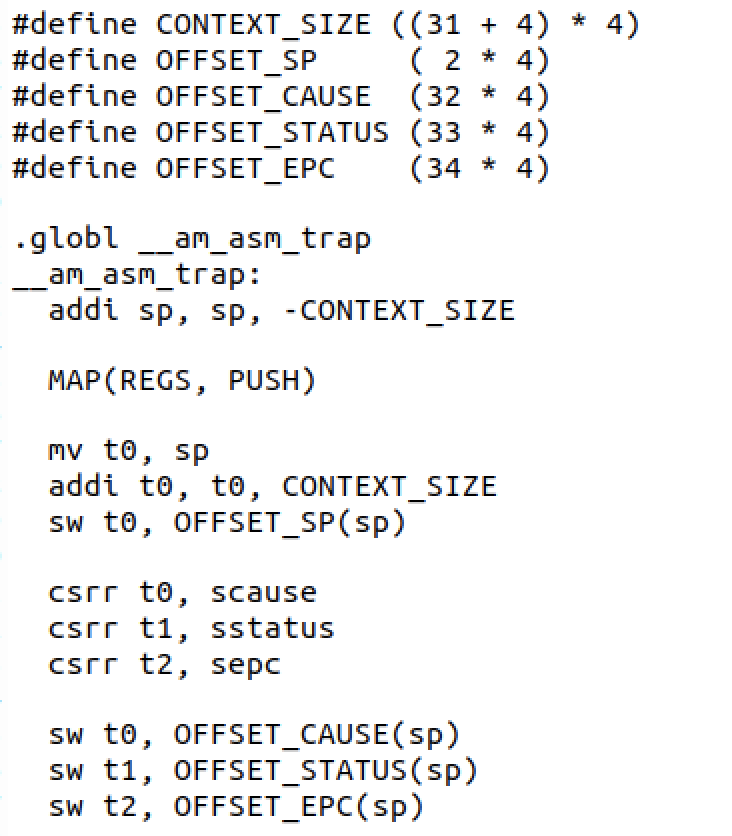


实现系统调用命令的译码辅助函数如下：

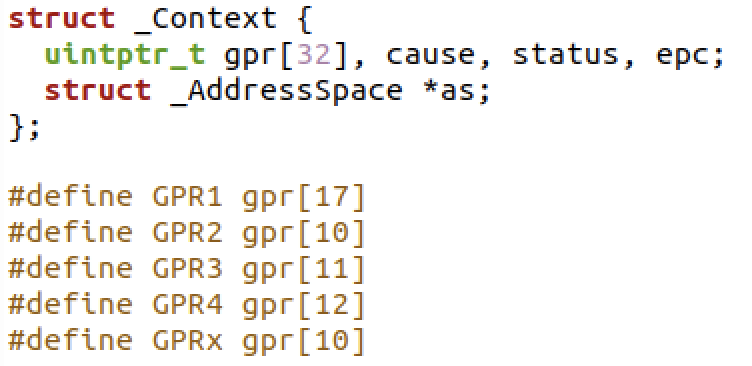


执行辅助函数中，通过funct3字段来实现不同指令的区分，其中更特别的是要根据指令立即数的值来区分ecall和sret指令。ecall 指令会调用 raise\_intr 函数来完成触发异常响应过程；sret 指令会返回到保存在sepc指令的下一条指令（通过sepc中的值加4来获得）；CSR 相关指令是为了对控制状态寄存器进行修改。为了完成CSR相关指令，我们实现一些辅助函数：其中 get\_csr 函数是根据参数返回具体的一个 csr 寄存器，write\_csr 函数是将值写入到相应的 csr 寄存器中。由于篇幅限制，在此省略执行辅助函数的实现。

接着我们需要重新组织定义的\_Context结构体的成员, 使得这些成员的定义顺序和nexus-am/am/src/$ISA/nemu/trap.S中构造的上下文保持一致。查看该文档内容如下：

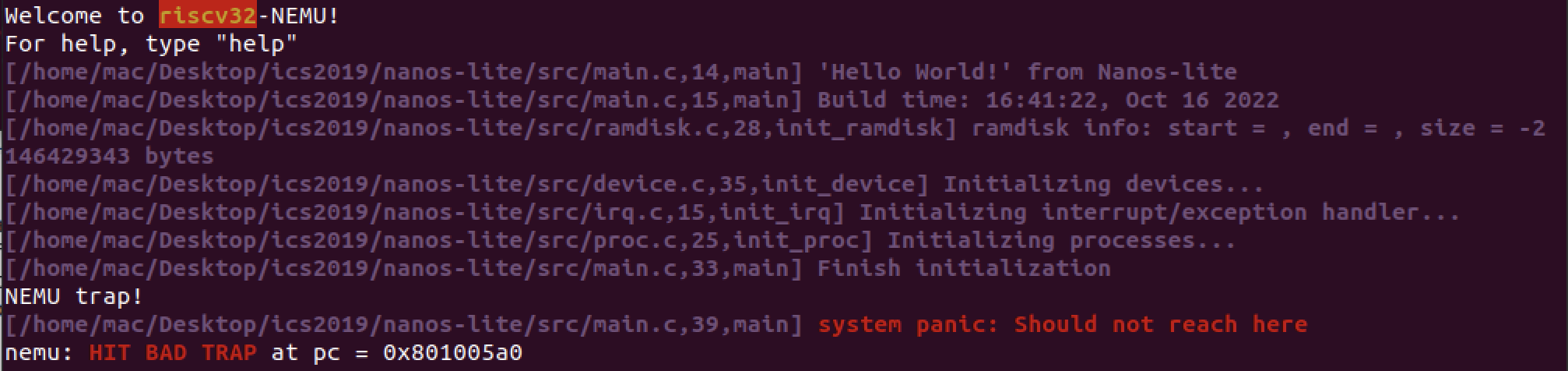


通过OFFSET\_CAUSE、OFFSET\_STATUS、OFFSET\_EPC的定义可以知道，这三者的顺序肯定是cause、status、epc。而处理这三者之前的“MAP(REGS，PUSH)”指令是对32个寄存器的压栈操作。完成\_Context的定义如下所示：



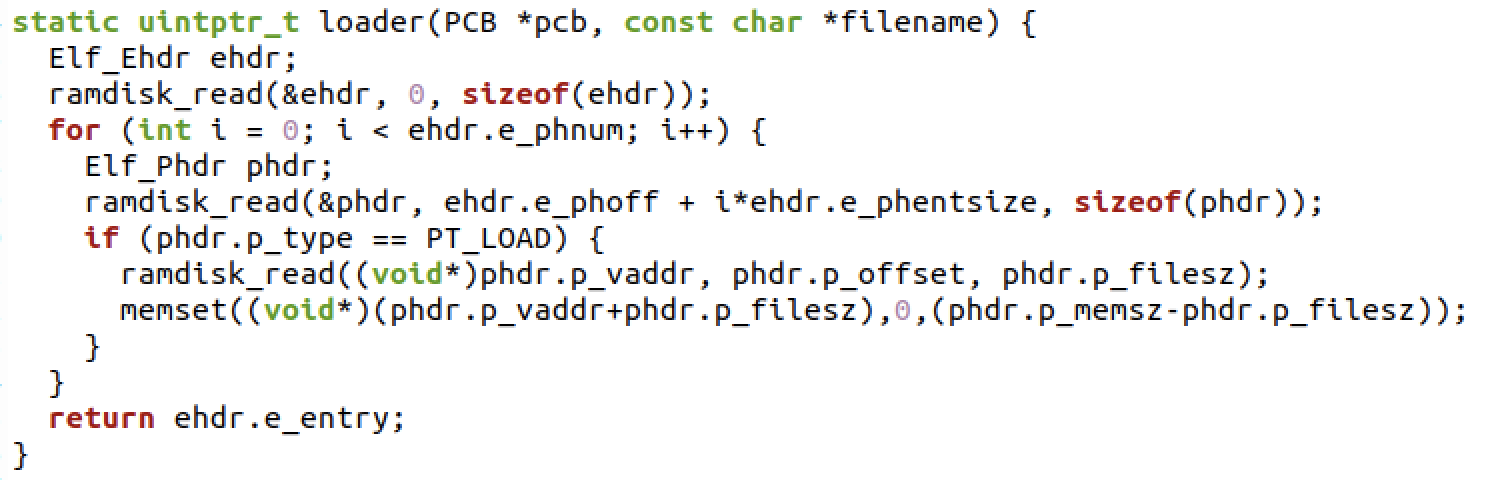
为了实现正确的事件分发，在\_\_am\_irq\_handle()中通过异常号识别出自陷异常, 当传入的\_Context结构中的cause为-1时，事件被设置成编号为\_EVENT\_YIELD的自陷事件。然后在do\_event()中识别出自陷事件\_EVENT\_YIELD ，并要求其调用printf输出一句话为“NEMU trap！”。

重新编译并运行Nanos-lite，可以看到输出了“NEMU trap！”，并且最后还是触发了main函数结尾设置的的panic()。



## PA 3.2

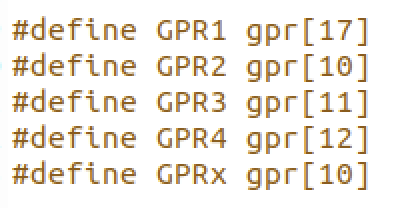
需要在Nanos-lite中实现loader的功能，修改在nanos-lite/src/loader.c中定义的loader()函数， 来把用户程序加载到正确的内存位置, 然后执行用户程序。通过 ramdisk\_read 函数把可执行文件中的代码和数据放置在正确的内存位置，然后跳转到程序入口。函数实现如下：



实现后, 在init\_proc()中调用naive\_uload(NULL, NULL), 它会调loader函数来加载第一个用户程序, 然后跳转到用户程序中执行。

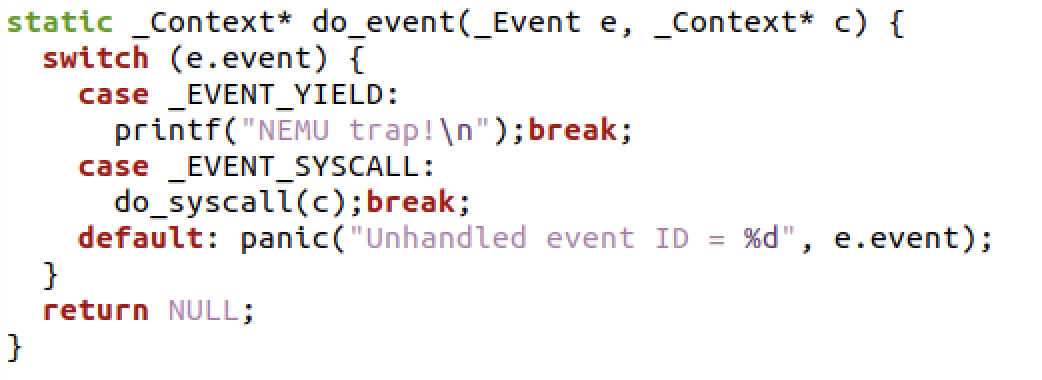


根据do\_syscall()函数的实现，我们发现定义的宏GPR1、GPR2、GPR3、GPR4、GPRx分别代表了系统调用类型、系统调用的第1、2、3个参数和系统调用的返回值。修改宏定义如下：

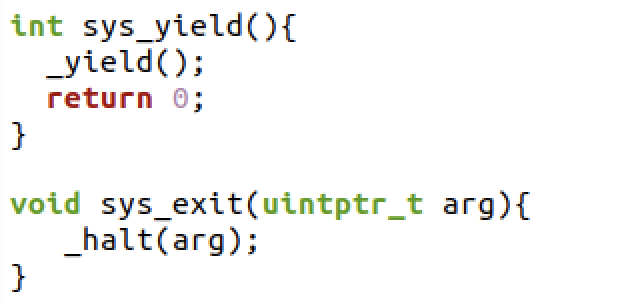


接着要实现系统调用，dummy程序已经通过直接调用\_syscall\_()触发系统调用, 需要实现函数能识别出系统调用事件\_EVENT\_SYSCALL。

首先通过do\_event()函数来识别出\_EVENT\_SYSCALL，并且调用do\_syscall()函数。



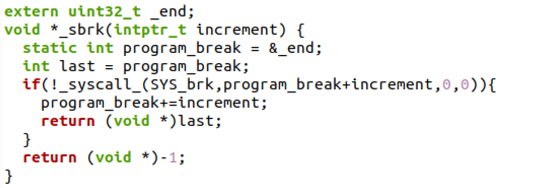
接着在do\_syscall()函数中在区分不同的系统调用事件，添加系统调用SYS\_yield 、SYS\_exit。为了能模块化，为处理不同的系统调用事件编写对应的函数。



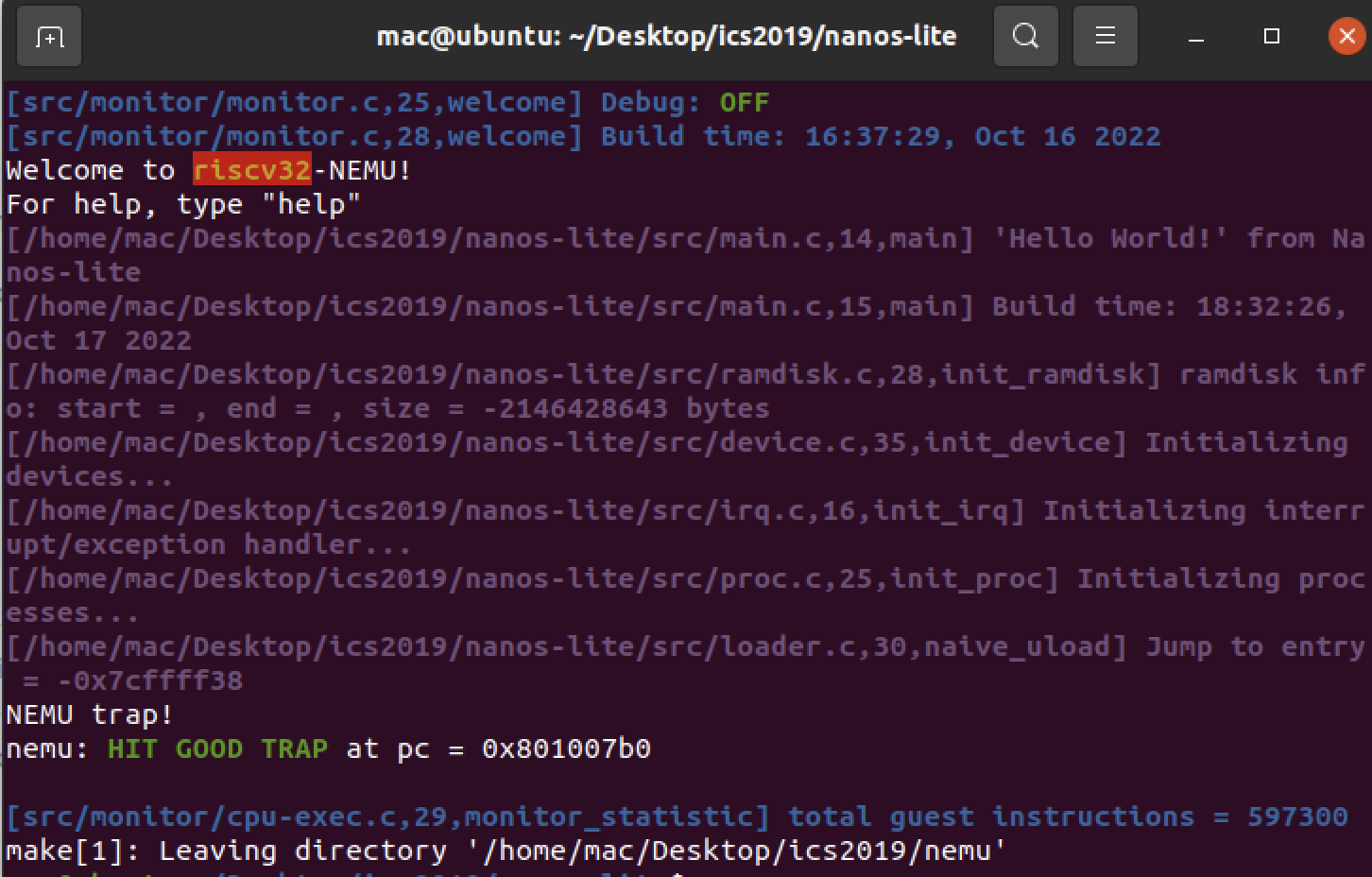
为了更好地实现printf函数，我们需要实现SYS\_write的系统调用函数。



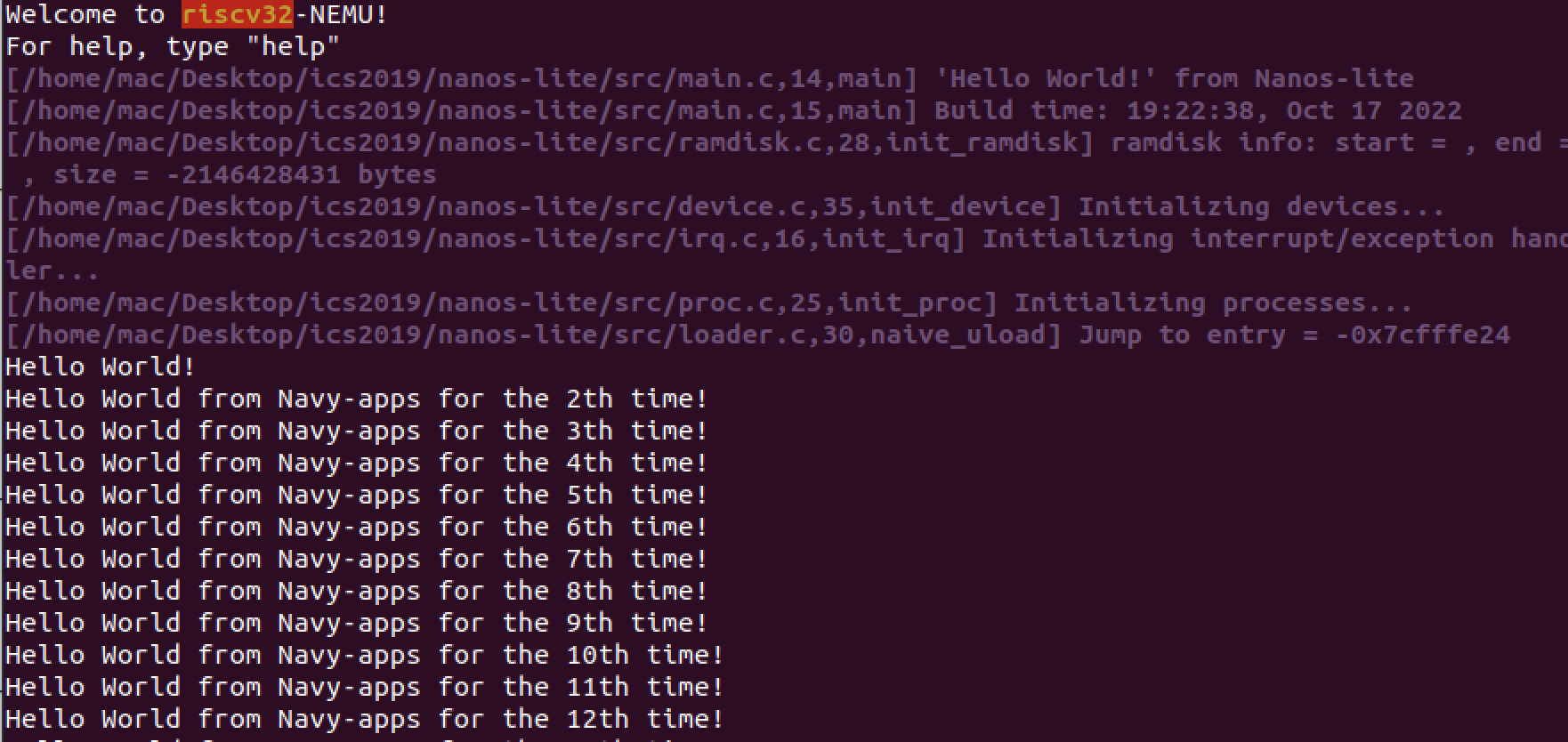
实现堆区大小调整我们按照文档提示实现SYS\_brk的功能。其中\_sbrk函数如下：



实现完之后运行dummy程序，运行结果如下所示：

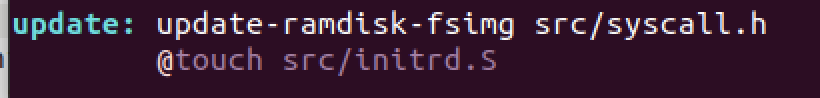


运行hello程序结果如下：



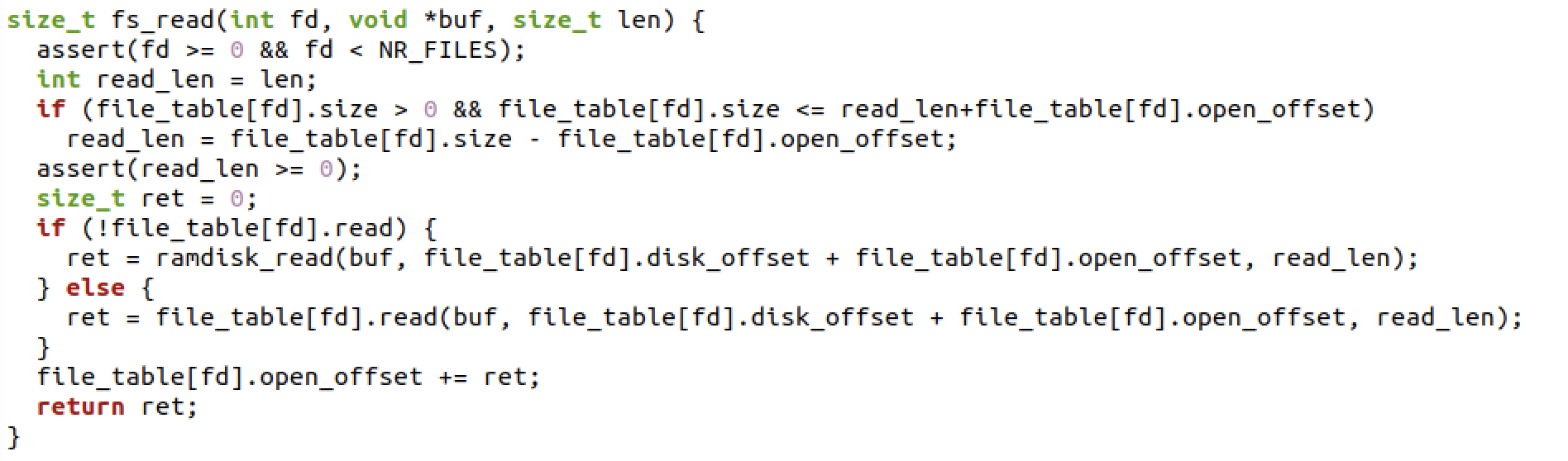
## PA 3.3

修改nanos-lite的makefile如下：



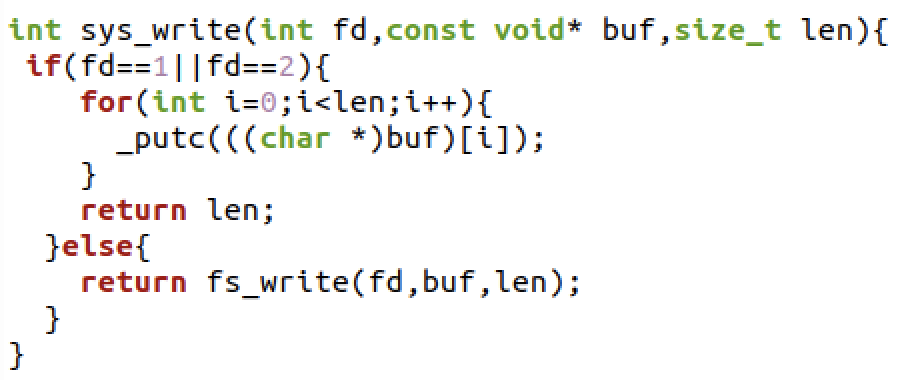
我们需要实现简易文件系统，并且将设备抽象成文件并支持相关操作。首先需要在nanos-lite/src/fs.c文件中实现fs\_open、fs\_read、fs\_write、fs\_lseek、fs\_close函数。

fs\_read和fs\_write函数还是通过ramdisk\_read和ramdisk\_write函数来实现，只不过此时与文件结构Finfo中的disk\_offset和open\_offset有关，操作的对象是通过传入的fd参数在file\_table中取得的文件。

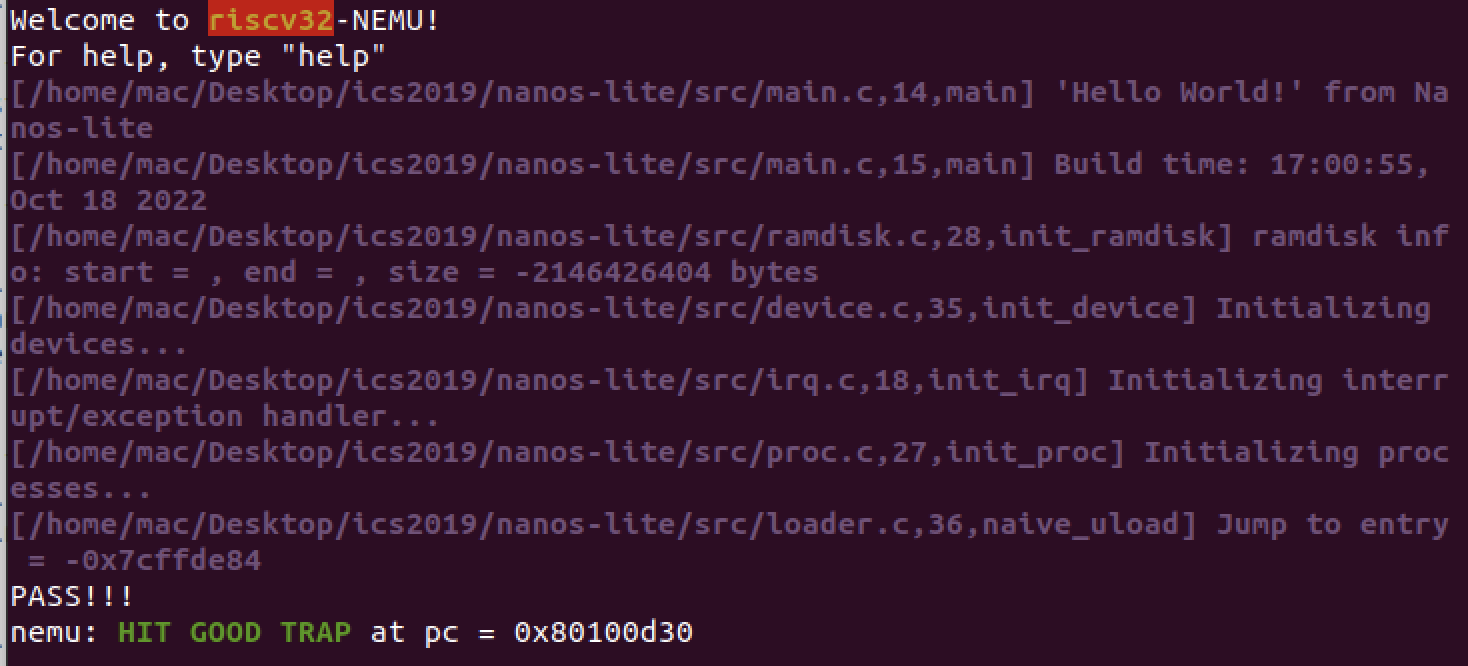


完成文件系统操作相关的函数之后，需要重写loader函数。此时的loader函数不能在直接调用ramdisk相关函数，而是需要通过文件系统相关的函数来进行加载函数。通过fs\_open打开传入的filename，接着通过fs\_read来读取以及fs\_lseek函数来调整文件被打开后的读写指针open\_offset。

修改传入naive\_uload()函数的文件名来运行测试程序text，却发现程序正常执行了，但是没有输出“PASS！！！”的信息。再查看file.h发现里面的内容也正常写入了，那么初步判断为该程序正常执行，但是printf标准输出流的功能没有实现。回到SYS\_write系统调用的功能，此时其具体实现已经修改为对文件的操作fs\_write，自然没有输出的功能，于是我们修改该函数如下所示：

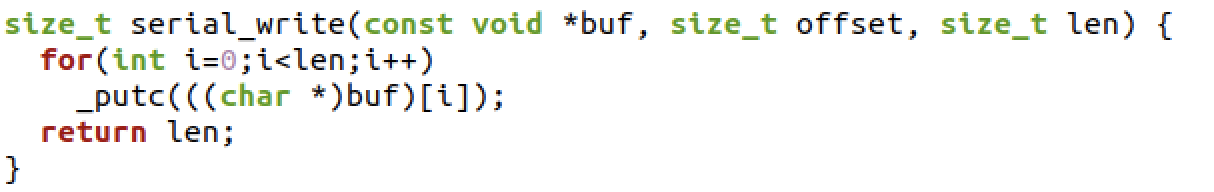


此时运行text测试程序结果如下，可以正常输出“PASS！！！”的信息：

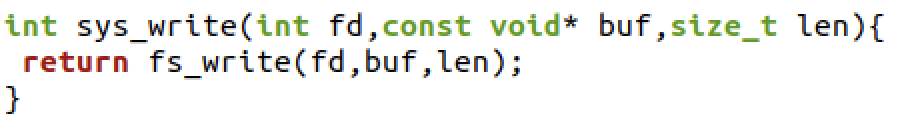


接着实现虚拟文件系统 VFS，把 IOE 抽象成文件。

为了能够实现串口输出，也就是上面我发现的问题：当修改sys\_write具体实现为fs\_write之后不输出的问题，我们需要实现serial\_write，并且在文件为标准输出流写时调用该函数。serial\_write的实现也很简单，就是之前fd等于1或2时调用\_putc函数的操作，具体实现如下：



实现该函数并且修改stdout的输出函数之后，我们再修改sys\_write为下图所示的实现，发现也可以正常实现text程序的“PASS！！！”信息的正常输出：

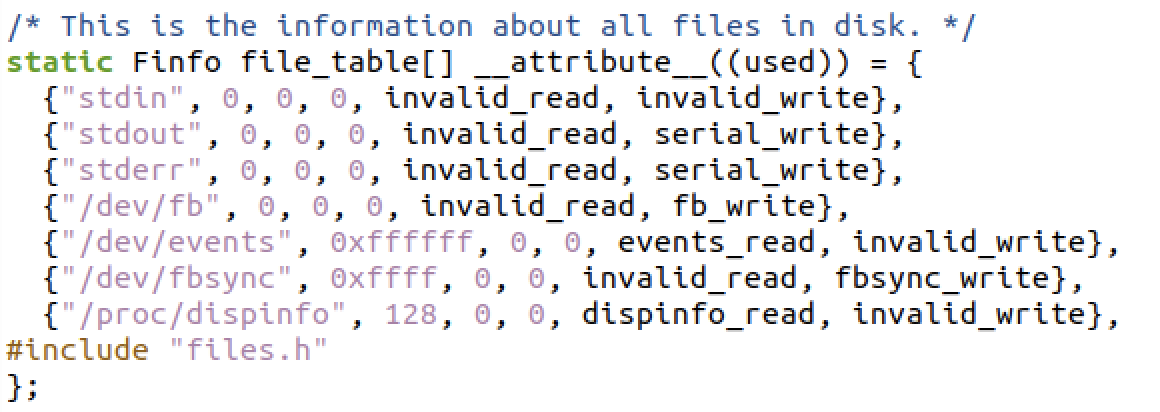


时钟和键盘的输入对系统而言可以视作事件，我们再将事件抽象成文件/dev/events，该文件需要支持读操作, 用户程序可以从中一次读出一个输入事件。

为了更新屏幕, 只需要将像素信息写入VGA的显存。首先我们把显存抽象成文件/dev/fb，它需要支持写操作和lseek。接着我们还需要为用户程序提供刷新屏幕的接口，刷新操作通过写入设备文件/dev/fbsync来触发, 它只需要支持写操作, 每次写入时都会将/dev/fb文件中的内容同步到屏幕。

为了能够更好地在屏幕显示信息，用户程序还需要获得屏幕大小的信息。屏幕大小的信息通过/proc/dispinfo文件来获得, 它需要支持读操作。

于是我们修改file\_table如下：



接下来需要完成events\_read、fb\_write、fbsync\_write、init\_device、dispinfo\_read 函数的实现。

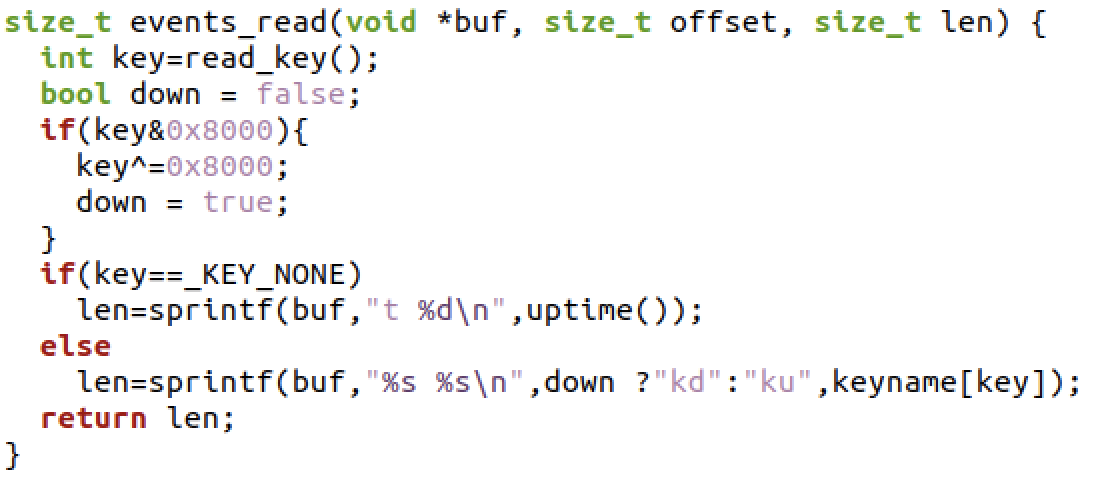
首先我们在fs.c中实现init\_fs函数，对/dev/fb文件进行初始化：



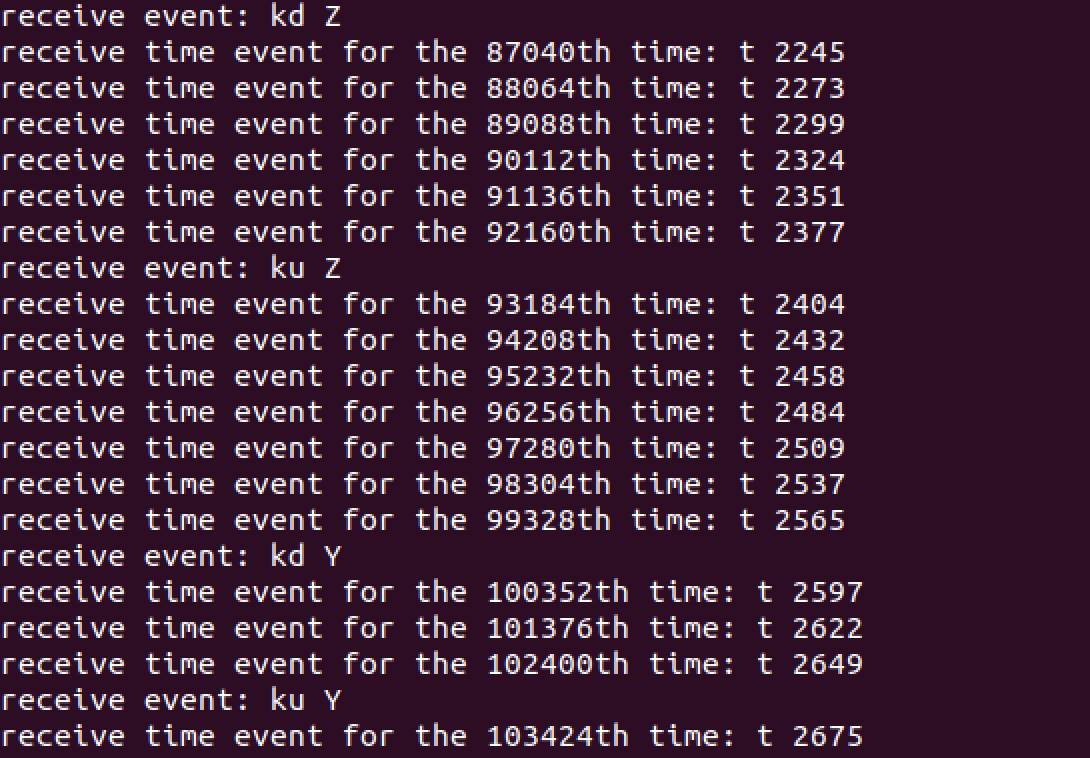
其中FD\_FB宏如下定义（代码框架中定义的），其顺序和file\_table中前四个是一致的：



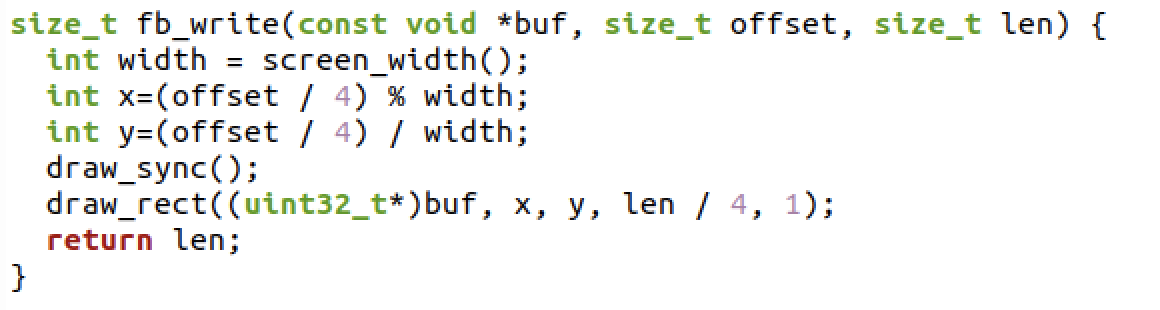
events\_read函数优先处理按键事件，通过调用read\_key函数来获取当前的key码，进行解析。如果此时key为KEY\_NONE，则读取时间写入buf中；否则再根据此时按键是否为按下输出按键码到buf中。实现如下：



对时钟和按键进行测试，运行events程序并测试，其结果如下：

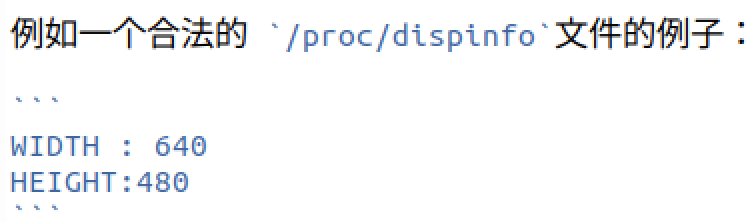


fb\_write()函数用于把buf中的len字节写到屏幕上offset处。VGA的内容是在屏幕上按行优先显示，需要先从offset结合屏幕宽度计算出此时需要在屏幕上进行打印的坐标, 然后调用IOE的draw\_rect()进行绘制，调用draw\_sync()函数是为了实现同步显示：

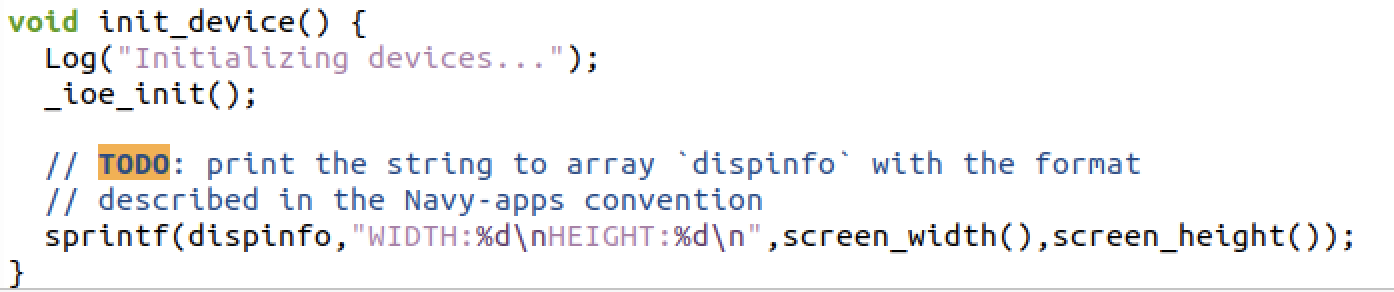


fbsync\_write()函数直接调用draw\_sync()函数即可。

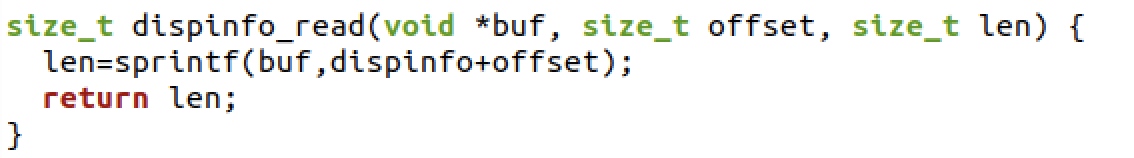
init\_device()将/proc/dispinfo的内容提前写入到字符串dispinfo中，查看在Navy\_apps里对/proc/dispinfo的内容定义如下：



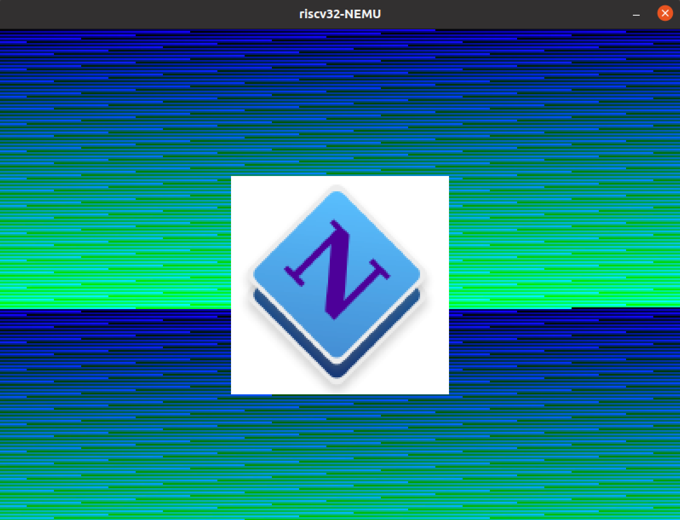
按照上述定义实现init\_device()函数如下：



dispinfo\_read()函数用于把字符串dispinfo中offset开始的len字节写到buf中，只需要调用sprintf函数即可。



最后运行bmptest程序，其运行结果如下：

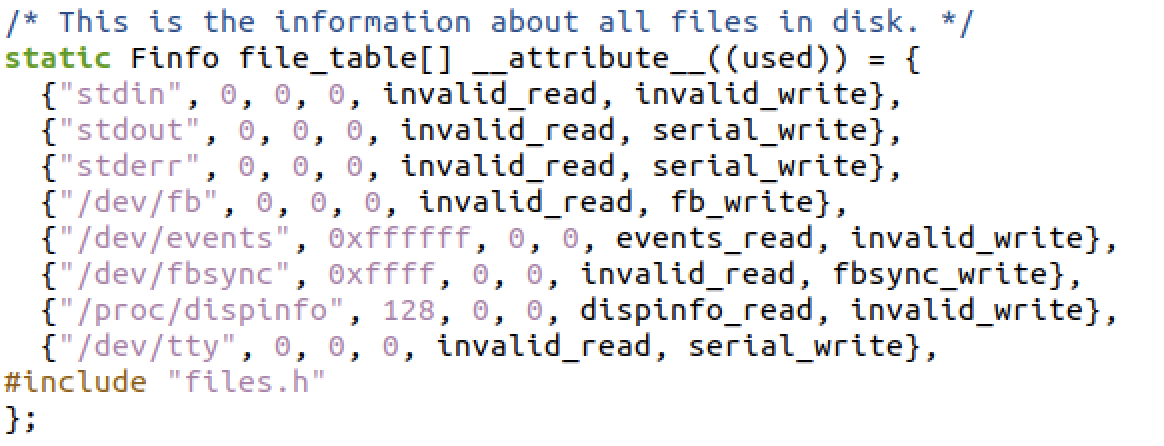


运行仙剑奇侠传：

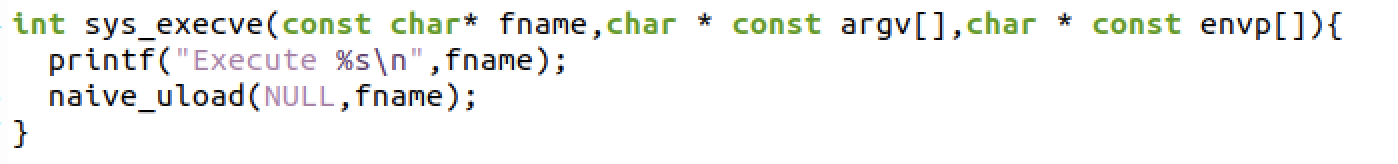




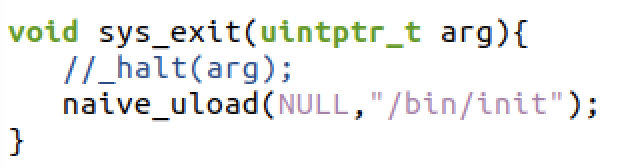
实现批处理系统，在文件系统中添加一个特殊文件/dev/tty：



接着实现SYS\_execve系统调用，其功能也很简单，就是加载对应的程序文件：



修改SYS\_exit系统调用，在退出时不是直接结束程序，而是回到init程序进行再次选择：



运行结果如下：



## 必答题

### 理解上下文结构体的前世今生

在\_\_am\_irq\_handle()中看到有一个上下文结构指针c, c指向的上下文结构是在trap.s中跳转到\_\_am\_irq\_handle()函数之前通过压栈来保存的。这个上下文结构是通过保存了32个寄存器和scause、sstatus、sepc三个控制状态寄存器获得的；其赋值就是通过trap.s中的压栈指令来完成的。

### 理解穿越时空的旅程

1. 调用\_yield()函数，首先通过内联汇编代码将a7设置为-1，表示当前的ecall类型是 \_yield ，接着执行了 ecall 指令。

2. 汇编 ecall 指令将会由 ecall 对应的执行辅助函数来执行相关的函数，与 ecall对应的指令函数分支中会调用 raise\_intr 函数，参数NO即为a7寄存器的值，也就是-1，表示中断号。

3. 在 raise\_intr 函数中会保存当前的 epc 到 sepc 寄存器，将中断号保存到 scause 寄存器，并从 stvec 获得中断入口地址并进行跳转。跳转到目标地址是 \_\_am\_asm\_trap 函数的入口地址，其实就是汇编代码 trap.S 中的起始位置。

4. 汇编代码执行上一小节中所述的数据压栈操作，并且会执行jal指令跳转到\_\_am\_irq\_handle 函数进行执行。

5. \_\_am\_irq\_handle 函数根据 c->cause 来分别进行处理，如果是 -1 就表示yield事件，如果 是0到19就说明是系统调用。此处是-1表示yield，于是填充ev.event成员 为 \_EVENT\_YIELD 并调用用户定义的回调函数 do\_event。

6. 在do\_event函数中根据event的类型来分别处理，如果是 \_EVENT\_YIELD 就打印出信息到终端，如果是 \_EVENT\_SYSCALL 的话就调用 do\_syscall。在执行yield函数中是打印信息到终端。

7. 函数结束之后将会回到 trap.S 汇编代码，恢复上下文并调用 sret 指令。

8. sret 指令将会调用nemu中针对 sret 指令的执行辅助函数，从 sepc 寄存器中读出之前保存的 pc，将其加4，表示中断发生执行指令的下一条指令地址，并进行跳转，恢复正常执行。

### hello程序是什么

用户程序的入口位于navy-apps/libs/libc/src/platform/crt0.c中的\_start()函数, 它会调用用户程序的main()函数, 从main()函数返回后会调用exit()结束运行。

hello程序以程序文件的形式保存在硬盘中，被编译成ELF文件之后，运行加载通过 naive\_uload 函数读至模拟磁盘ramdisk上，并放在约定好的位置。用户程序需要被链接到内存位置0x83000000(riscv32)附近。可执行文件位于ramdisk偏移为0处, 访问它就可以得到用户程序的第一个字节。

naive\_uload 函数同时记录其位置为entry，接着通过((void(\*)())entry) ()指令跳转到对应位置进行执行。

hello程序中的printf通过 SYS\_write 系统调用来输出字符，程序执行完毕后回收其内存空间，也就是清空ramdisk上的内容。

### 仙剑奇侠传究竟如何运行

仙鹤的像素信息存放在数据文件mgo.mkf中，Nanos-lite通过库函数读取该文件获取画面的像素信息。画面通过 VGA 像素显示，在本节中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。

# 总结与感想

在本次课程设计中，我基于设计好的模拟器代码框架，实现了一个简易调试器，完成了所需的所有 RISC-V 指令，实现 I/O测试了打字小游戏，实现了系统调用和文件系统，最终在实现的模拟器上运行了游戏——仙剑奇侠传。

在PA1阶段，我们需要实现简易调试器，其内容大体和gdb相似。这个过程比较简单也比较好理解，但是可以让我清晰地体会到调试器的实现原理。其中比较复杂的是实现表达式求值，结合了编译原理中学到的词法分析、C语言程序递归求值等知识。

在PA2阶段，需要完成指令系统。这让我翻开了在计算机组成原理中留下的指令集文件，对比理解了riscv32中的I型、R型、B型的指令。这一阶段一开始上手比较难，需要自己去理解各种宏、各种基础函数的代码，结合才能理解程序是如何实现指令的译码和执行的。实现设备输入输出也是需要对基础的函数和定义有比较完整的了解。

在PA3阶段，在前一个阶段的基础上，实现系统调用的指令，完成文件系统，以及将设备I/O抽象成文件。这一阶段整体上文档比较详细，而且之前有了一些基础，运行起仙剑奇侠传不那么难。

总体而言，本实验结合了操作系统、计算机组成原理、计算机系统原理、编译原理等专业课程的知识，完成了一次知识的整合和模拟真实计算机。整个实验设计得合理且有实践意义，对于系统能力培养这门课程确实是可以达到目的。

