必答题+测试结果的说明+心得建议

一、必答题

PA1

1. **我选择的ISA是** *riscv32*
2. **理解基础设施**

   $500 * 90\% * 30 * 20 = 270000s = 75h$

   一学期将要花费 75 个小时的时间在调试上。

   $500 * 90\% * 20 * 20 = 180000s = 50h$

   一学期可以节省 50 个小时调试的时间。
3. **查阅 riscv32 手册**
   - **riscv32有哪几种指令格式?**

     6种
   - **LUI指令的行为是什么?**

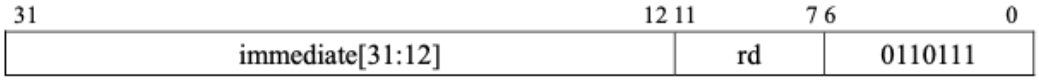
     根据文档中的说明

     lui rd, immediate　　x[rd] = sext(immediate[31:12] << 12)

     高位立即数加载 *(Load Upper Immediate)*. U-type, RV32I and RV64I.

     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。

     压缩形式：**c.lui** rd, imm

     `LUI`指令的行为是*高位立即数加载*
   - **mstatus寄存器的结构是怎么样的?**

     根据文档可知`mstatus`寄存器的结构如下所示

     - mstatus（Machine Status）它保存全局中断使能，以及许多其他的状态，如图 10.4 所示。

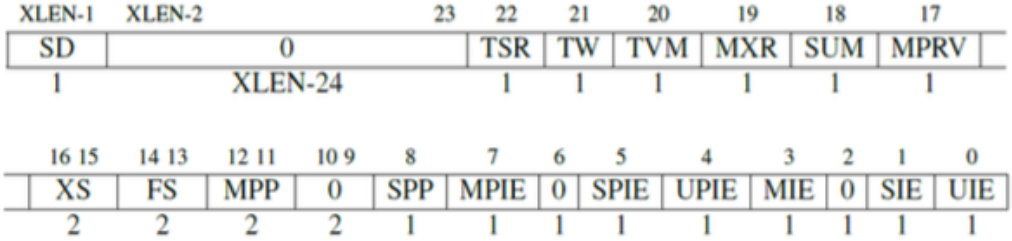
     

     图 10.4：mstatus 控制状态寄存器。在仅有机器模式且没有 F 和 V 扩展的简单处理中，有效的域只有全局中断使能、MIE 和 MPIE（它在异常发生后保存 MIE 的旧值）。RV32 的 XLEN 时 32，RV64 是 40。（来自[Waterman and Asanovic 2017]中的表 3.6；有关其他域的说明请参见该文档的第 3.1 节。）

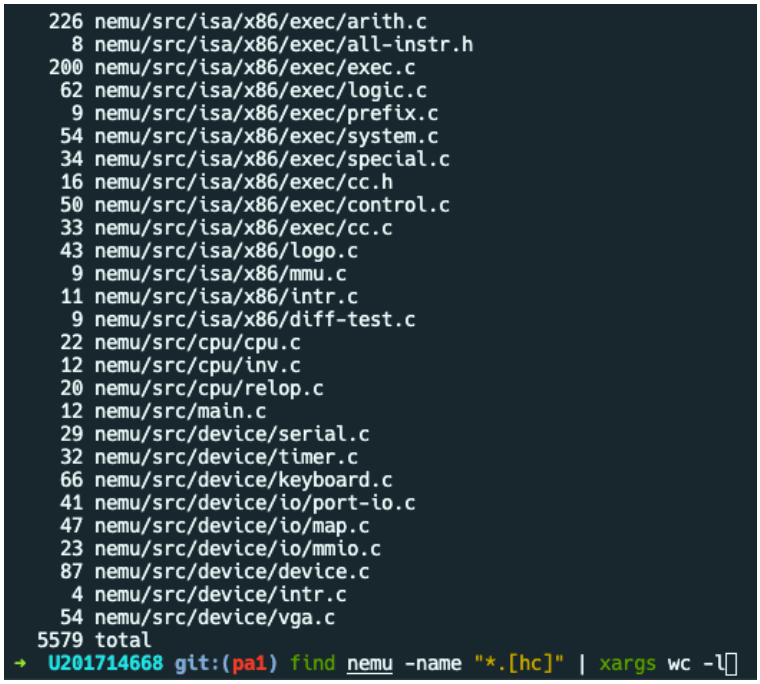
4. **完成PA1的内容之后, `nemu/`目录下的所有.c和.h和文件总共有多少行代码? 你是使用什么命令得到这个结果的? 和框架代码相比, 你在PA1中编写了多少行代码? (Hint: 目前`pa1`分支中记录的正好是做PA1之前的状态, 思考一下应该如何回到"过去"?) 你可以把这条命令写入`Makefile`中, 随着实验进度的推进, 你可以很方便地统计工程的代码行数, 例如敲入`make count`就会自动运行统计代码行数的命令. 再来个难一点的, 除去空行之外, `nemu/`目录下的所有`.c`和`.h`文件总共有多少行代码?**

   通过`git checkout pa1`回到刚完成pa1的状态

   在`nemu/`目录下输入

   ```
   find nemu -name "*.[hc]" | xargs wc -l
   ```

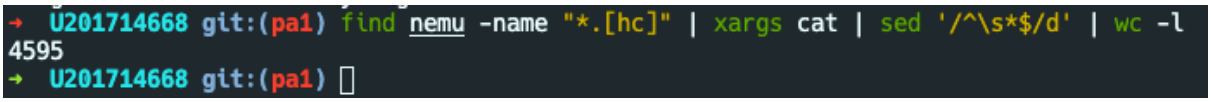
   即可输出`nemu`目录下所有`.c`文件以及`.h`文件的行数，如下图所示

   对于空行可以通过以下指令来去除

   ```
   find nemu -name "*.[hc]" | xargs cat | sed '/^\s*$/d' | wc -l
   ```

   运行结果如下

   利用同样的方法也可以去除所有的空行以及注释，在此不再赘述

5. **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` 打开gcc所有的警告

   `-Werror` 将所有的警告当成错误进行处理

   使用`-Wall`以及`-Werror`有利于发现代码中不严谨之处，如类型的隐式转换

PA2

1. **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)**

   以`lw`指令为例

   ```
   `isa_exec`函数被执行
     |
     |==> 通过`instr_fetch`函数获得当前`pc`值对应的内存位置中的指令
         调用`idex`，也即`译码-执行`函数
        |
        |==> 调用`decode_ld`函数，获得操作数，将其保存在全局变量`id_src`以及
   `id_dest`中
              调用`exec_load`函数，
             |
             |==> 根据`funct3`字段，索引`load_table`，找到对应的执行函数`exec_ld`，
   同时设定位宽width
                   调用函数`exec_ld`
                   |
                   |==>根据`lw`指令的逻辑，调用rtl函数
                       调用`rtl_lm`将id_src.addr所指向的地址取出decinfo.width宽度的
   数据，保存在临时寄存器s0中
                       调用`rlt_sr`将s0保存在id_dest.reg指向的寄存器中
                       根据宽度调用print_asm_template2打印具体的汇编指令
   ```

2. **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?**

   - 去掉static 不会发生错误
   - 去掉inline，编译时不会发生错误，链接时会发生错误。LD在链接不同的object时，会发现同一个符号在不同的object中被多次定义，并且我们没有告诉LD这种情况下应该怎么进行链接，因此报错

3. **编译与链接**

   - **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?**

     一个

   - **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.**

     两个

     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

   - **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.)**

出现重定义

```
➜  nemu git:(pa2) make ARCH=riscv32-nemu

Building x86-nemu
+ CC src/monitor/monitor.c
In file included from ./include/common.h:32,
                 from ./include/nemu.h:4,
                 from src/monitor/monitor.c:1:
./include/debug.h:9:21: error: redefinition of ‘dummy’
    9 | volatile static int dummy = 0;
      |                     ^~~~~
In file included from ./include/nemu.h:4,
                 from src/monitor/monitor.c:1:
./include/common.h:12:21: note: previous definition of ‘dummy’ was here
   12 | volatile static int dummy = 0;
      |                     ^~~~~
make: *** [Makefile:51: build/obj-x86/monitor/monitor.o] Error 1
➜  nemu git:(pa2) ✗ make ARCH=riscv32-nemu
```

给变量赋值之后，声明就变成了定义，所以会发生重定义问题

4. 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.)

   通过阅读文档中给出的C语言基础中的makefile基础一节以及man文档，可以对makefile有一个基本的了解

   对于Makefile，基本的工作方式如下

   1. 首先依次读取变量“MAKEFILES”定义的 makefile 文件列表
   2. 读取工作目录下的 makefile 文件
   3. 一次读取工作目录下的 makefile 文件指定的 include 文件
   4. 查找重建所有已读的 makefile 文件的规则
   5. 初始化变量值并展开那些需要立即展开的变量和函数并根据预设条件确定执行分支
   6. 建立依赖关系表
   7. 执行 rules

   具体到`/nemu/Makefile`这个文件本身，具体的编译链接过程如下

   1. 首先Makefile默认的ISA为x86，如果定义了具体的ISA，则需要保证其有效
   2. 根据ISA确定需要include的文件列表
   3. 确定编译目标文件夹（默认是`build/`）确定编译器以及链接器（`gcc`）
   4. 设置编译选项`CFLAGS`
   5. 读取所有需要编译的`.c`的文件并将其编译为`.o`文件
   6. 进行链接
   7. 执行`git commit`

PA3

1. **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?

`__am_irq_handle`函数在整个项目中，唯一出现的调用文件就是`trap.S`。其中通过`jal`指令直接跳转到对应的函数体，而函数的参数是保存在栈当中的，可以看到在`jal`指令之前有诸多压栈操作，按照顺序是

1. 32个寄存器，通过`MAP(REGS, PUSH)`
2. 成员`cause`，通过`sw t0 OFFSET_CAUSE(sp)`
3. 成员`status`，通过`sw t1 OFFSET_STATUS(sp)`
4. 成员`epc`，通过`sw t2 OFFSET_EPC(sp)`

于是不同的成员被赋值。可以在`__am_irq_handle`函数中被使用

2. **理解穿越时空的旅程 (见PA3.1阶段)**

> 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.

- 首先`_yield`函数，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。
- 汇编`ecall`指令将会由`ecall`对应的`EHelper`来执行相关的函数，函数中会调用`raise_intr`函数，参数NO即为a7寄存器的值，表示中断号。
- 在`raise_intr`函数中会保存`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。也就是`__am_asm_trap`函数的入口地址，也就是汇编代码`trap.S`中的起始位置。开始执行。
- 汇编代码会执行到上述的`__am_irq_handle`函数
- `__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(支持的系统调用的个数)就说明是系统调用。此处是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`
- 同样是根据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`，此处是打印信息到终端
- 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
- `sret`指令将会调用nemu中针对`sret`指令的执行函数，从`sepc`寄存器中读出之前保存的pc，将其加4，表示中断发生时的吓一跳指令的地址，并进行跳转

至此`yield`函数执行完毕

3. **hello程序是什么, 它从而何来, 要到哪里去 (见PA3.2阶段)**

> 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?

- hello程序对应的elf文件会在整个项目编译的时候，将其在ramdisk的偏移位置保存在`files.h`的记录表中(`disk_offset`成员)。
- 接着通过load函数解析对应的elf文件，得到具体的入口地址，保存在`e_entry`中
- 通过`((void(*)())entry) ()`跳转到对应位置进行执行。
- 在main函数中通过printf进行输出，printf函数首先会尝试进行`_brk`，如果失败则一个字符

一个字符的通过`write`输出到终端，如果成功则将字符串作为整体调用`write`进行输出。
    - `write`函数会调用`_write`系统调用，在对应的处理函数中，发现输出的对象是stdout，则直接通过`serial_write`进行输出。
4. **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?**
    - 在`navy-apps/apps/palc/hal/hal.c`中的`redraw`函数中通过`NDL_DrawRect`和`NDL_Render`更新屏幕。
    - `NDL`通过之前的初始化操作维护了一块画布`canvas`， 并将其绘制操作限定在该画布上。
    - `NDL_DrawRect`会将传入的`pixels`保存到会把传入的像素逐个存入画布的对应位置
    - 首先调用了`libndl`库，在该库中会打开设备文件`/dev/fb`和`/dev/fbsync`，在接收到该函数调用后会向`/dev/fb`设备文件中写入，在该函数 中，判断出要写的文件是`/dev/fb`设备文件之后，会调用`fb_write`帮助函数， 之后会调用`draw_rect`函数，该函数位于`nexus-ambsibc/io.c`中，在函数内会调用`_io_write`函数
    - 而`_io_write`会转发给`__am_video_write`函数， 该函数中会执行 out 汇编指令，将数据传送给`vga`设备中。
    - vga 设备在接收到数 据后会保存在定义的显存中，当之后 NDL 库向`/dev/fbsyn`设备文件中写入时， vga 设备最终会调用 SDL 库来更新画面。
    - `NDL_Render`函数会对画布的每一行先调用`fseek`把偏移量定位到该行起点在屏幕中对应的位置，然后调用`fwrite`输出画布的一行，并调用`fflush()`刷新缓冲，最后调用`putc`向`fbsyncdev`输出 0 进行同步，并调用`fflush`刷新缓冲。

二、测试结果以及说明

PA1

不同指令的测试

1. 帮助指令`help`

```
(nemu) help
help - Display informations about all supported commands
c - Continue the execution of the program
q - Exit NEMU
si - Single step
info - Show info of regs/watchpoint
p - Evaluate given expression
x - Scan memory
w - Set watchpoint
d - Remove watchpoint
(nemu)
```

2. 单步执行`si`

   包含参数缺省值以及指定步数的情况

```
(nemu) si
80100000:   b7 02 00 80                                     lui  0x80000,t0
(nemu) si 2
80100004:   23 a0 02 00                                     sw  0(t0),$0
80100008:   03 a5 02 00                                     lw  0(t0),a0
(nemu)
```

3. 打印寄存器信息`info r`

```
(nemu) info r
[src/monitor/debug/ui.c,120,cmd_info] 进入info
[src/monitor/debug/ui.c,130,cmd_info] 打印参数点
$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
(nemu)
```

4. 表达式求值

```
(nemu) p ((20 * 30) - 100 + ($tp + 10) *  ($t0 * 10000))
结果为500
(nemu) p (1+2)*(4/3)
结果为3
(nemu) p (3/3+)(123*4
[src/monitor/debug/expr.c,306,eval] 求值失败
[src/monitor/debug/expr.c,334,eval] main_op: 求第一个子表达式出错
[src/monitor/debug/ui.c,162,cmd_exp] cmd_exp: 求值失败
```

5. 监视点相关

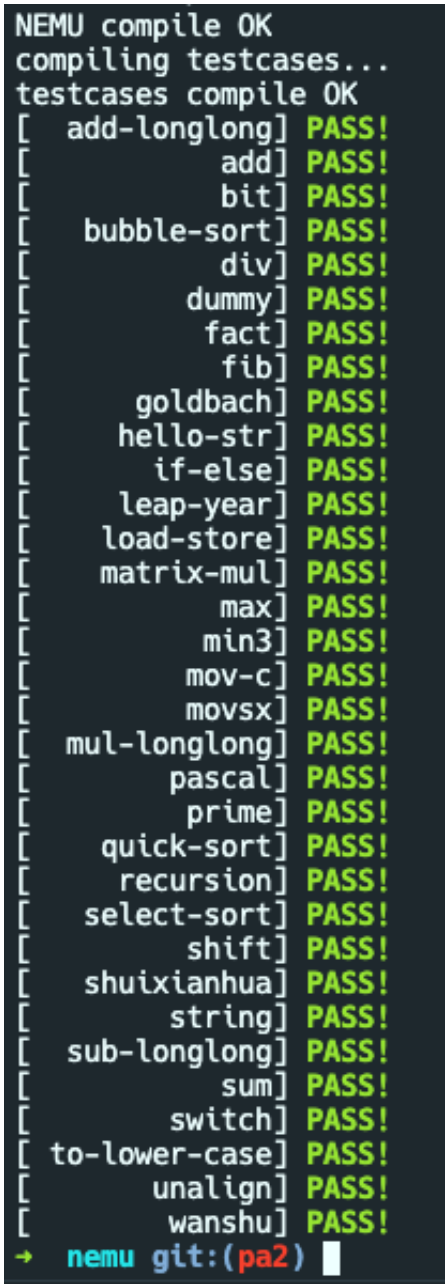
```
(nemu) w $t0 == 0
[src/monitor/debug/watchpoint.c,26,new_wp] 新建的监视点的value为$t0 == 0
(nemu) w 100 * $s11 * 10
[src/monitor/debug/watchpoint.c,26,new_wp] 新建的监视点的value为100 * $s11 * 10
(nemu) info w
[src/monitor/debug/ui.c,120,cmd_info] 进入info
WP NO.1 "100 * $s11 * 10" value=0
WP NO.0 "$t0 == 0" value=1
(nemu) d 0
(nemu) info w
[src/monitor/debug/ui.c,120,cmd_info] 进入info
WP NO.1 "100 * $s11 * 10" value=0
(nemu)
```

所有功能均实现完毕

PA2

> 以下测试一部分是在macbook中的virtualbox虚拟机中运行的，一部分是在一台双系统的ubuntu中运行的

1. runall.sh回归测试脚本



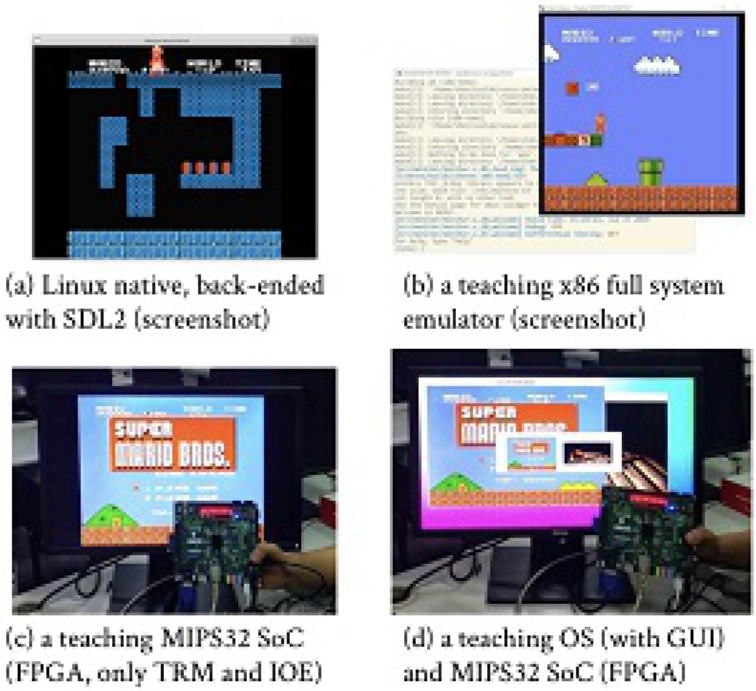
2. Microbench

```
Empty mainargs. Use "ref" by default
======= Running MicroBench [input *ref*] =======
[qsort] Quick sort: * Passed.
  min time: 5516 ms [92]
[queen] Queen placement: * Passed.
  min time: 5324 ms [88]
[bf] Brainf**k interpreter: * Passed.
  min time: 39743 ms [59]
[fib] Fibonacci number: * Passed.
  min time: 428913 ms [6]
[sieve] Eratosthenes sieve: * Passed.
  min time: 130769 ms [30]
[15pz] A* 15-puzzle search: * Passed.
  min time: 20424 ms [21]
[dinic] Dinic's maxflow algorithm: * Passed.
  min time: 10910 ms [99]
[lzip] Lzip compression: * Passed.
  min time: 21878 ms [34]
[ssort] Suffix sort: * Passed.
  min time: 5681 ms [79]
[md5] MD5 digest: * Passed.
  min time: 65710 ms [26]
================================================
MicroBench PASS        53 Marks
                 vs. 100000 Marks (i7-7700K @ 4.20GHz)
Total time: 775686 ms
nemu: HIT GOOD TRAP at pc = 0x801072a0

[src/monitor/cpu-exec.c,30,monitor_statistic] total guest instructions = 9054015931
make[1]: Leaving directory '/home/samuel/U201714668/nemu'
➜  microbench git:(pa3)
```

由于电脑的性能实在是太低，所以分数也很差。

3. ppt演示



(a) Linux native, back-ended with SDL2 (screenshot)

(b) a teaching x86 full system emulator (screenshot)

(c) a teaching MIPS32 SoC (FPGA, only TRM and IOE)

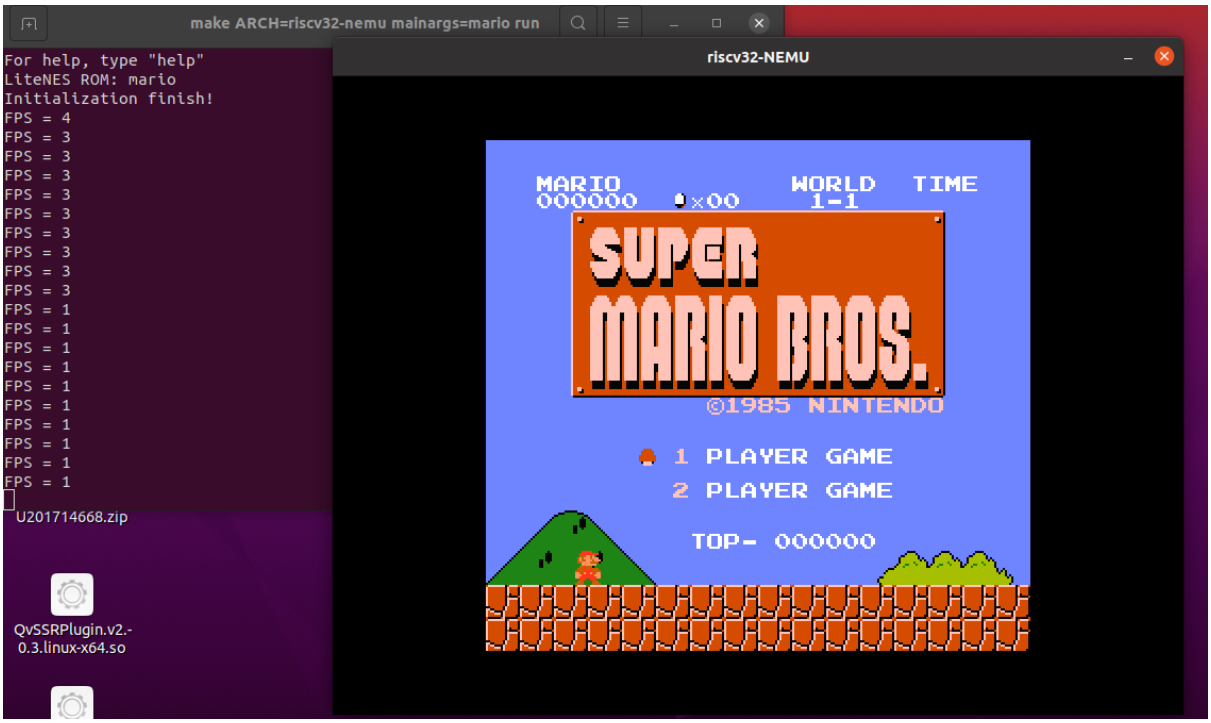
(d) a teaching OS (with GUI) and MIPS32 SoC (FPGA)

Figure 4. The same LiteNES emulator running on different platforms.

---

4. 打字游戏

5. litenes



PA3

1. dummy测试

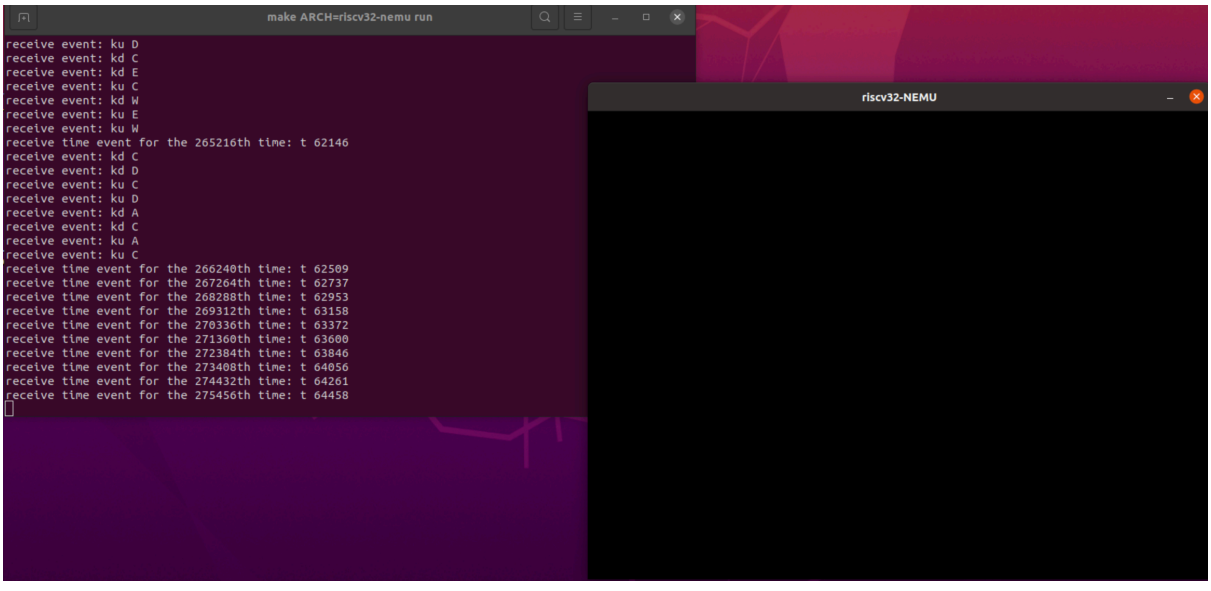
```
Welcome to riscv32-NEMU!
For help, type "help"
Hello! This is Samuel[from nanos-lite/src/logo.txt file][/media/sf_VirtualBox_Shared_Folder/U201714668/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/media/sf_VirtualBox_Shared_Folder/U201714668/nanos-lite/src/main.c,15,main] Build time: 00:12:21, Jan 14 2021
[/media/sf_VirtualBox_Shared_Folder/U201714668/nanos-lite/src/ramdisk.c,29,init_ramdisk] ramdisk info: start = 0x80104a18, end = 0x82297455, size = 35203645 bytes
[/media/sf_VirtualBox_Shared_Folder/U201714668/nanos-lite/src/device.c,61,init_device] Initializing devices...
[/media/sf_VirtualBox_Shared_Folder/U201714668/nanos-lite/src/irq.c,27,init_irq] Initializing interrupt/exception handler...
[/media/sf_VirtualBox_Shared_Folder/U201714668/nanos-lite/src/proc.c,27,init_proc] Initializing processes...
[/media/sf_VirtualBox_Shared_Folder/U201714668/nanos-lite/src/loader.c,53,naive_uload] Jump to entry = 83000110
self int
nemu: HIT GOOD TRAP at pc = 0x801017d0

[src/monitor/cpu-exec.c,30,monitor_statistic] total guest instructions = 1759233
make[1]: Leaving directory '/media/sf_VirtualBox_Shared_Folder/U201714668/nemu'
➜  nanos-lite git:(pa3)
```

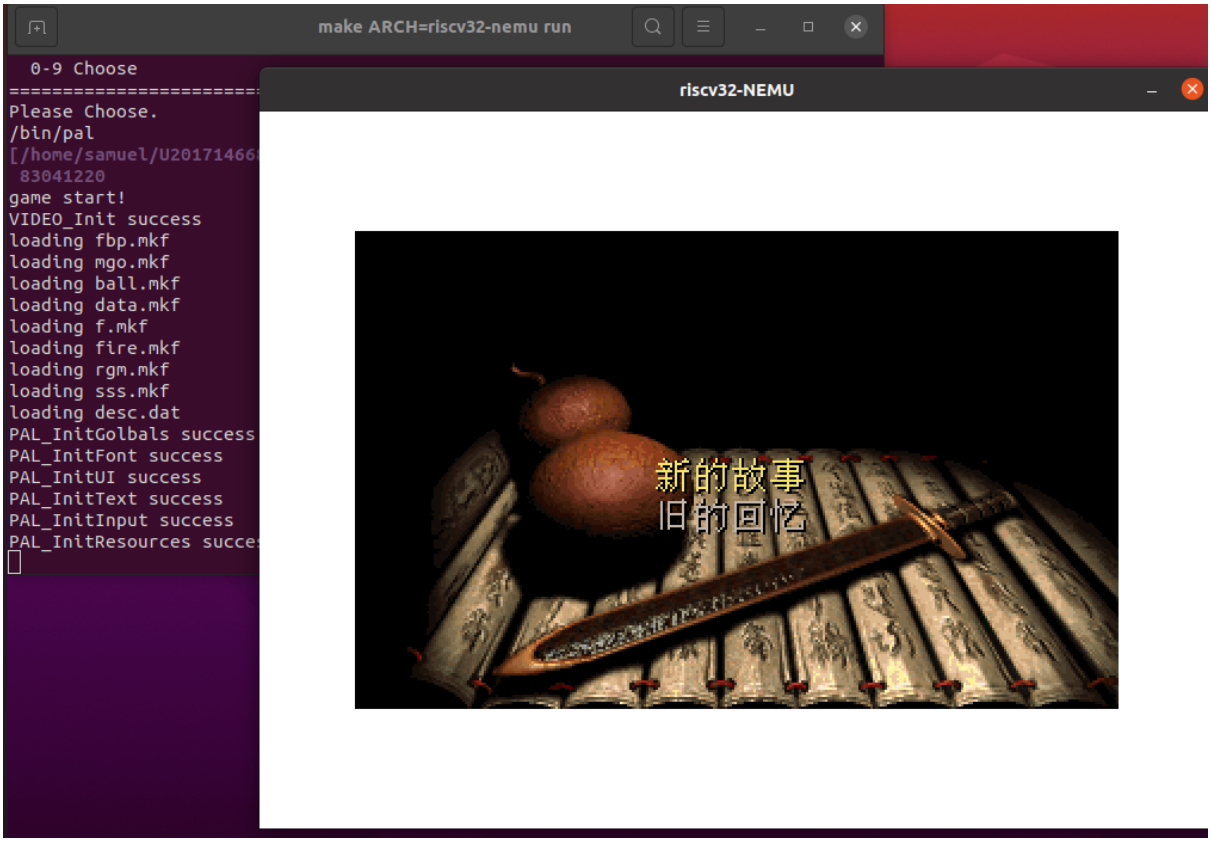
2. hello测试

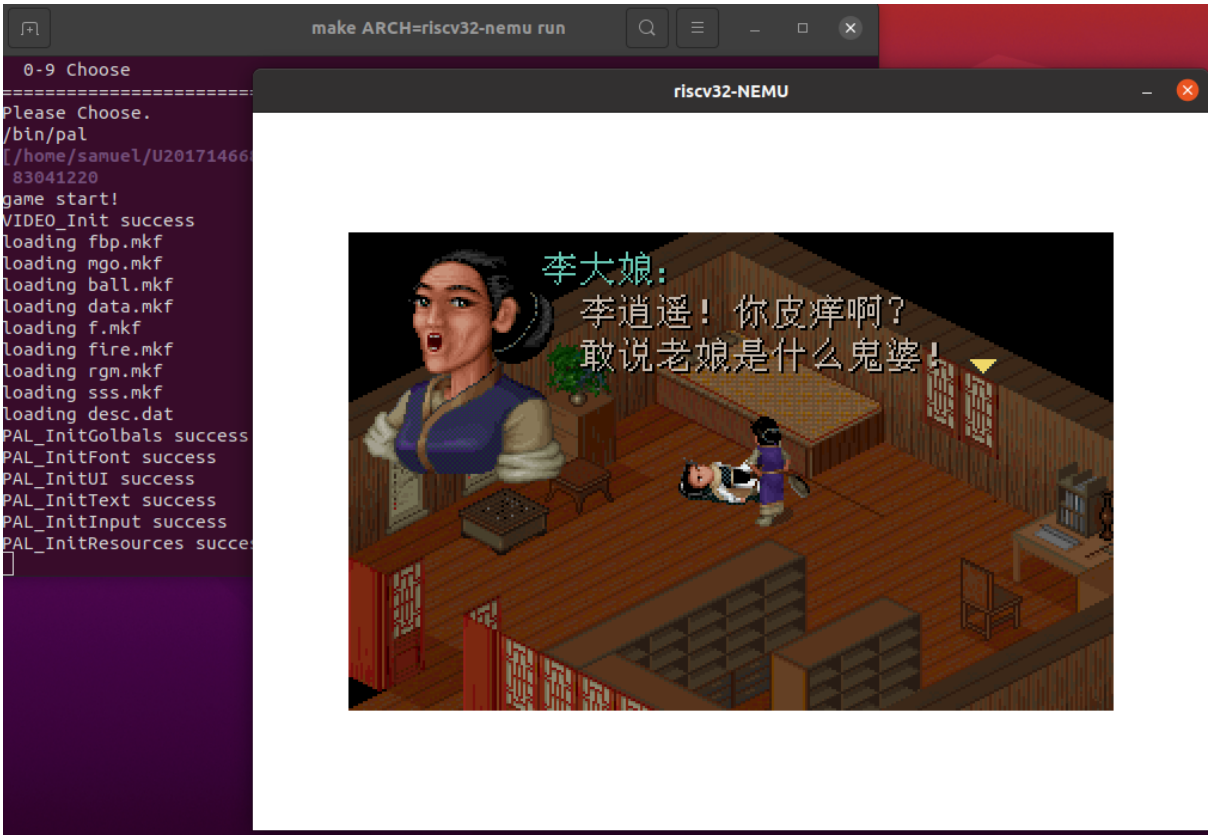
```
Hello World from Navy-apps for the 106th time!
Hello World from Navy-apps for the 107th time!
Hello World from Navy-apps for the 108th time!
Hello World from Navy-apps for the 109th time!
Hello World from Navy-apps for the 110th time!
Hello World from Navy-apps for the 111th time!
Hello World from Navy-apps for the 112th time!
Hello World from Navy-apps for the 113th time!
Hello World from Navy-apps for the 114th time!
Hello World from Navy-apps for the 115th time!
Hello World from Navy-apps for the 116th time!
Hello World from Navy-apps for the 117th time!
Hello World from Navy-apps for the 118th time!
Hello World from Navy-apps for the 119th time!
Hello World from Navy-apps for the 120th time!
Hello World from Navy-apps for the 121th time!
Hello World from Navy-apps for the 122th time!
Hello World from Navy-apps for the 123th time!
Hello World from Navy-apps for the 124th time!
Hello World from Navy-apps for the 125th time!
Hello World from Navy-apps for the 126th time!
Hello World from Navy-apps for the 127th time!
Hello World from Navy-apps for the 128th time!
Hello World from Navy-apps for the 129th time!
Hello World from Navy-apps for the 130th time!
Hello World from Navy-apps for the 131th time!
Hello World from Navy-apps for the 132th time!
Hello World from Navy-apps for the 133th time!
Hello World from Navy-apps for the 134th time!
```

3. events测试



4. 运行仙剑奇侠传





三、心得建议

这次的项目可以说是整个大学期间事件跨度最长的了。同时也涉及到了非常多的知识，从最基础的c语言，到数据结构算法，具体的指令集，以及vga等更加具体的知识。很幸运的我选择了riscv32作为实现的指令集（其实pa1的时候是准备做x86的，后来由于时间原因还是改成了riscv32）。在整个实现的过程，尤其是pa2，深刻感受到了riscv的简明（相对于x86来说）。整个项目分为几个阶段，我最终是完成了pa3，所以以下就分阶段分别进行介绍。

Pa1

pa1的重点其实就是表达式求值的实现以及监视点的实现。在表达式求值的过程中，由于文档其实给出了很清晰的思路，所以还是比较顺畅。主要还是在一些细节的处理上，比如

- 对解引用和乘法的区分。在具体实现的时候采用了先整体遍历一遍`tokens`，将原本识别为乘法的解引用标记为解引用之后再进行求值的方法，而不是在求值的过程中进行判断。
- 括号不匹配时会有两种情况，一种是单纯的不匹配，另一种是括号错误，前者是可以进行计算的，后者无法进行计算。需要在遍历的过程中对括号进行计数来进行区分。

同时在监视点的实现过程中我也遇到了一些问题，一开始想着节约空间就没有为监视点内的表达式申请空间，而是直接保存了指针，但是后来发现这样会出现问题，于是才使用malloc单独分配了空间。

Pa2

pa2主要就是实现RV32I以及RV32M指令集以及klib的实现。一开始看代码的时候着实是让人头大，因为框架代码中使用了多重重叠的宏定义，而我使用的编辑器(vscode)又无法自动的将其展开到最后一层，就导致非常难以理解。在理清了整个指令执行的过程之后，发现其实基本上就是体力活，只需要根据指令的具体说明，在函数体中调用rtl相关的函数即可。而在实现klib时遇到了各种各样的问题，主要就是print相关的函数，因为一开始的时候并没有考虑到后续的许多需求，只是想着快点写完，然后之后在运行pa3中的某些代码的时候就会出现莫名其妙的错误，最后检查出来是因为自定义的格式化参数没有实现相关的功能，如`%x`，在运行pa2的回归测试的时候，string测试一直不通过。最终定位到memset的代码。

```
// 通过
void* memset(void* v,int c,size_t n) {
  unsigned char * ptr = v;
  unsigned char val = c;
  while (n-- > 0) { // 唯一的区别
    ptr[n] = val;
  }
  return v;
}
// 没有通过
void* memset(void* v,int c,size_t n) {
  unsigned char * ptr = v;
  unsigned char val = c;
  while (--n >= 0) { // 唯一的区别
    ptr[n] = val;
  }
  return v;
}
```

按照理论来说两者应该是一样的，但是实际上一个测试可以通过另一个测试无法通过，就很奇怪。

在完善了string.c之后，`hello-str`测试也出了问题。通过对hello-str不断的修改，发现出现常规字符的时候就会报错，进而将错误定位到vsprintf函数中对于非格式化字符的处理中，我直接使用了_putc函数，但是应该要根据输出位置的不同（标准输出还是内存）使用putc函数，修改之后解决问题。后续的输入输出主要是通过参考native的实现来完成的。同时在运行microbench的时候遇到了非常奇怪的问题，因为我在macbook上的虚拟机跑分非常的低（只有几十分），所以我就特地在另一台电脑上安装了ubuntu的双系统，在双系统上运行游戏，如打字游戏可以感觉到运行速度明显提高，但是microbench的跑分还是非常的低，就很奇怪。

Pa3

这个部分理解起来还是比较困难的，主要是因为涉及到了汇编以及c的联合编译，以及elf等具体的格式。但是在整个调用的流程完全弄清楚之后，又非常的开心，尤其是汇编和c之间的完美合作，感觉非常的神奇。当然实际编写代码的还是遇到了很多困难。在运行hello测试的时候，只有使用`write`的输出可以正常进行，而使用`printf`的输出则无法正常进行，当时卡了非常久想了各种办法还是没有找到问题所在，后来干脆就先放着不管，先做后面的文件系统了，因为想着那些loader相关的函数之后都需要使用fs相关的函数重新编写一遍。之后在实现了fs之后，hello测试确实可以通过了，现在回想起来大概率是loader函数写的有问题。

这次的pa项目虽然还有很多遗憾的地方，比如后续的pa4和pa5都没有实现，以及前面的一些代码还有非常多优化的空间，但是从已经完成的部分我已经学到了很多，比如环境搭建，我在最初编写代码的过程中摸索出了一套非常棒的工具链，就是使用virtualbox的无头模式启动虚拟机，并将项目文件放在共享文件夹下，这样就可以直接在本机上