

# 华中科技大学

## 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:	CS2008 班
学 号:	U202015543
姓 名:	李海涛
电 话:	18719466649
邮 件:	18719466649@163. com
完成日期:	2024-01-18



# 目 录

1	课程实验概述 .....	1
1.1	课设目的 .....	1
2	实验方案设计 .....	2
2.1	PA1-开天辟地的篇章: 最简单的计算机 .....	2
2.1.1	PA1.1: 实现单步执行, 打印寄存器状态, 扫描内存 .....	2
2.1.2	PA1.2: 实现算术表达式求值, PA1.3: 实现所有要求 .....	4
2.2	PA2 - 简单复杂的机器: 冯诺依曼计算机系统 .....	10
2.2.1	PA2.1: 在 NEMU 中运行第一个 C 程序 dummy .....	10
2.2.2	PA2.2: 实现更多的指令, 在 NEMU 中运行所有 cputest .....	13
2.2.3	PA2.3: 运行打字小游戏 .....	15
2.3	PA3 - 穿越时空的旅程: 批处理系统 .....	19
2.3.1	PA3.1: 实现自陷操作 _yield() 及其过程 .....	19
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 21	
2.3.3	PA3.3: 运行仙剑奇侠传并展示批处理系统 .....	23
3	实验结果与结果分析 .....	28
	参考文献 .....	错误!未定义书签。

# 1 课程实验概述

## 1.1 课设目的

代码框架中实现一个简化的 RISC-V 模拟器：

- (1) 可解释执行 RISC-V 执行代码
- (2) 支持输入输出设备
- (3) 支持异常流处理
- (4) 支持精简操作系统---支持文件系统
- (5) 支持虚存管理
- (6) 支持进程分时调度

最终在模拟器上运行“仙剑奇侠传”,探究“程序在计算机上运行”的机理,掌握计算机软硬件协同的机制,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,提升计算机系统能力。

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。

## 2 实验方案设计

### 2.1 PA1-开天辟地的篇章：最简单的计算机

#### 2.1.1 PA1.1: 实现单步执行, 打印寄存器状态, 扫描内存

命令 `si` 实验过程:

命令 `si n`: `n` 就是要执行的步数, 没有则默认为 1


按照提供的代码框架添加相应的新实现的命令, 类比即可。首先在 `cmd_tablep[]` 数组添加指令 `si` 对应的结构体:

`{ "si", "Single step execution", cmd_si }` (后续指令类似)

`cpu_exec()` 函数中参数 `n`: 就是执行 `n` 条指令

`char *arg = strtok(NULL, " ");` 指向命令输入后面的第一个参数。利用 `strtok` 可以解析得到 `si` 后面的 `n`, 然后利用框架的 `cpu_exec()` 函数即可实现指令 `si`

命令 `si` 测试:



```
Welcome to riscv32-NEMU!  
For help, type "help"  
(nemu) si  
80100000: b7 02 00 80          lui  0x800000,t0  
(nemu) si 2  
80100004: 23 a0 02 00          sw   0(t0),$0  
80100008: 03 a5 02 00          lw   0(t0),a0  
(nemu) █
```

命令 `info` 实验过程:

- `reg.c`
  - 实现函数 `isa_reg_display()`
  - 直接用 `printf` 打印, 参考 GDB 的输出信息, 用格式说明符 `\t`
  - 文件 `reg.h` 代码里提供了宏 `reg_l(index)` 用于访问寄存器的值, `index` 对应代码框架中提供的寄存器数组中下标
  - 不要忘记 `pc` 寄存器
- `ui.c`
  - 使用另外一个文件里的函数, 需要声明
  - 和 `cmd_si()` 一样只需分析第一个参数即可
  - 根据第一个参数是 `r` 还是 `w` 来分别实现指令 `info r` (这里要求实现) 和 `info w` (后续阶段实现)
  - 直接调用 `reg.c` 文件中实现的函数 `isa_reg_display()`

命令 `info` 测试

```

nemu: HIT GOOD TRAP at pc = 0x8010000c
[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 4
(nemu) info r
$0      0x00000000      0
ra      0x00000000      0
sp      0x00000000      0
gp      0x00000000      0
tp      0x00000000      0
t0      0x80000000      -2147483648
t1      0x00000000      0
t2      0x00000000      0
s0      0x00000000      0
s1      0x00000000      0
a0      0x00000000      0
a1      0x00000000      0
a2      0x00000000      0
a3      0x00000000      0
a4      0x00000000      0
a5      0x00000000      0
a6      0x00000000      0
a7      0x00000000      0
s2      0x00000000      0
s3      0x00000000      0
s4      0x00000000      0
s5      0x00000000      0
s6      0x00000000      0
s7      0x00000000      0
s8      0x00000000      0
s9      0x00000000      0
s10     0x00000000      0
s11     0x00000000      0
t3      0x00000000      0
t4      0x00000000      0
t5      0x00000000      0
t6      0x00000000      0
pc      0x80100010      -2146435056
(nemu)

```

命令 x 实现过程:

- ui.c
  - 利用 `strtol()` 函数将第二个参数即十六进制字符串转换成十六进制数。
  - `paddr_t` 同等于 `uint32_t` 用于存储 4 个字节长度的物理地址。利用代码框架文件 `memory.c` 提供的 `uint32_t paddr_read(paddr_t addr, int len)` 函数可以读取从地址 `addr` 开始的长度为 `len` 个字节的内容。注意，机器中一个地址对应一个字节。

命令 x 的测试

扫描 0x80100000 处内存，正好对应内置程序的二进制代码

```

(nemu) st 4
80100000: b7 02 00 80      lui 0x80000,t0
80100004: 23 a0 02 00      sw 0(t0),$0
80100008: 03 a5 02 00      lw 0(t0),a0
8010000c: 6b 00 00 00      nemu trap
nemu: HIT GOOD TRAP at pc = 0x8010000c
[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 4
(nemu) x 10 0x80100000
0x80100000: b7 02 00 80
0x80100004: 23 a0 02 00
0x80100008: 03 a5 02 00
0x8010000c: 6b 00 00 00
0x80100010: 00 00 00 00
0x80100014: 00 00 00 00
0x80100018: 00 00 00 00
0x8010001c: 00 00 00 00
0x80100020: 00 00 00 00
0x80100024: 00 00 00 00
(nemu)

```

## 2.1.2 PA1.2: 实现算术表达式求值, PA1.3: 实现所有要求

### 表达式求值

#### expr.c

```
enum{}
```

根据需要识别的 token, 添加相应的枚举变量以标识。

```
rules[]
```

在 rules 数组中, 加入相应 token 的正则表达式, 和其对应的枚举变量。

在这里特别解释为什么有的正则表达式会出现两个反斜杠\\: 首先如果 token 是正则表达式中的元字符, 需要用转义字符'\'', 然后在 C 语言字符串中需要用到两个反斜杠\\才能表示转义字符'\''

特别注意, 这里的十六进制应该定义在十进制数前面, 否则十六进制数 0x 开头会被先看作时十进制 token

指针解引用 token 为了区分乘号, 这个问题不在这里解决, 而在求值的时候根据前一个 token 的类型来解决。

为了解决后续表达式求值时求取主运算符(用来分治表达式), 在 rules 结构体数组和后续的 tokens 结构体数组中都加入成员 int 型 priority 来表示优先级

```
make_token()
```

在这里, 除了整数 token 其他只需记录 token 的类型即可, 整数还需要将表示整数的字符串记录下来。

由于记录字符串的数组长设置为 32, 所以首先需要判断需要记录的字符串是否超过长度。然后利用 make\_token() 代码框架中提供的 substr\_start, substr\_len 来进行字符串拷贝, 需要特别注意要在字符串最后填上'\0'

```
expr()
```

表达式求值总接口: 包含词法分析和递归求值过程

在这里, 先区分指针解引用和乘号: 如果\*号位于第一个, 或者\*号前一个 token 类型为空格或左括号(, 则该\*将被识别为指针解引用而不是乘号

success 用于在 ui.c 中调用 expr() 时判断求值是否成功

eval() 第一个参数对应第一个 token 的数组下标, 第二个参数对应第二个 token 的数组下标

debug: 全文的 type 要统一, 这里不要将 TK\_MULTIPLE 写成'\*'

```
eval()
```

按照讲义上提供的分治迭代思路和代码框架加以完善

在进行分析之前，先进行空格处理：将表达式左右两边的空格字符“去掉”  
 表达式只有一个 token 的时候：十进制数，十六进制数，寄存器。这里寄存器利用函数 `isa_reg_str2val()` 求值，后面再分析  
 在分治表达式之前，先特判指针解引用情况，因为找主运算符函数不会识别指针解引用。因此这种情况下，需要手动分治，将\*号后面的式子看成表达式再递归一次 `eval()`  
 除 0 表达式处理：只需在最后的迭代合并时，判断第二个值是否为 0 即可

`isa_reg_str2val()`

这里比较简单，代码框架中字符串数组 `regsl[]` 记录了寄存器的名字，`reg_l()` 宏可以读取寄存器的值。这里用一个循环遍历所有情况，利用 `strcmp()` 函数来识别寄存器名字。注意这里要特殊判断 `pc` 寄存器和 `$0` 寄存器  
 使用 `strcmp()` 往往在 `if` 语句中，不要忘记 `==0`  
`uint32_t isa_reg_str2val(const char *s, bool *success) {`

`check_parentheses()`

匹配左右括号：可以用消消乐的思想，一个左括号需要一个右括号来消掉，用一个值来表示这个关系，如果最后值为 0 则证明是符合 BNF 的表达式

`get_op()`

找到分治表达式的主运算符：首先过滤掉括号，然后只需从左到右找到 `priority` 值最小的运算符即可（从左到右扫描，解决了情况：4+1+2，第二个加号才是首次分治的主运算符。注意，越前面分治的是真正计算越后面）

## ui.c

由于需要在 `ui.c` 使用 `reg.c` 的 `isa_reg_display()` 函数，所以需要声明一下，或者 `#include "reg.h"`

实现 `p` 命令，和前面实现 `si`, `info`, `x` 命令基本流程一样  
 在 `cmd_table[]` 添加相应的成员

`//pa2: 表达式求值`  
`{ "p", "Expression evaluation", cmd_p },`

`cmd_p()`

直接调用 `expr()` 即可，然后根据 `success` 输出相应的提示信息

## 如何测试你的代码

### gen-expr.c

gen\_rand\_expr()

特别注意，choose(n)生成的随机数范围为  $0 \sim n-1$

在这里我人为地控制随机生成的表达式长度为 12 个字符（包含一个 '\0' 字符），然后也是采用分治递归的思想

当当前处理的表达式长度只有 1 个或 2 个字符时，只能是整数。这里生成 1 位整数时取巧不生成 0，为了防止表达式出现除 0 的情况（但还是有一种间接除 0 的情况：7\*24/(6/88)，无法处理）

main()

将验证表达式求值的程序源代码作为字符串存储，然后利用 sprintf() 和格式说明符，将生成的随机表达式放入源代码字符串合适的位置。

利用相关文件处理函数将代码字符串输出成一个 .c 文件，再用 system() 编译 .c 文件生成可执行文件，用相关管道处理函数执行可执行文件得到输出结果

### main.c

手动编译 gen-expr.c 文件：gcc gen-expr.c -o gen-expr

运行 gen-expr 100 次，并将输出输入到 input 文件：./gen-expr 100 > input  
（参考 The Missing Semester 相关 bash 知识）

在 init\_monitor() 后读取 input 文件，调用 expr() 函数，对比两个结果看是否一直

这里 fscanf() 返回值不读取会发出警告，并且存放返回值的变量 useless 如果未使用也会发出警告，在定义变量 useless 时使用 \_\_attribute\_\_((unused)) 可以告诉编译器该变量无用，不再发出警告

## 监测点（本来是 pa2 内容，但由于连通 pa1.2，所以在这里写了）

注意，监视点不等于断点

## w 指令

### ui.c

类似的完善基础框架



```
cmd_table[]
```

```
//pal: 实现监视点  
{ "w", "Set watchpoint", cmd_w },
```

```
cmd_w()
```

调用 `new_wp()` 函数，并根据其返回值输出提示信息

## watchpoint.h

WP 结构体

expr: 表示监视点的表达式

changed, newValue, oldValue: 用于后续的检查监视点的值是否发生变化

## watchpoint.c

```
new_wp()
```

首先判断两种生成监视点失败的情况:

1. 监视点池没有空闲位置 (设置最多有 32 个监视点)
2. 监视点的表达式不合法, 无法求值, 需要重新回到 (nemu)

然后就是简单的链表操作, 注意在空闲链表和占用链表的增删结点的操作

```
watchpoint_monitor()
```

特别注意, 这里需要遍历所有已设的监视点来查看值是否改变, 而不是只检查 head 结点

## cpu-exec.c

在对应位置, 按照讲义提示增加

```
//pal: 实现监视点  
bool changed = watchpoint_monitor();  
if(changed)  
    nemu_state.state = NEMU_STOP;
```

## info w 指令

### ui.c

```
cmd_info()
```

完善上面实现 info r 的函数。当匹配'w'时直接调用 watchpoint\_display() 函数

```
else if(strcmp(arg, "w") == 0)
    //pal: 打印监视点信息
    watchpoint_display();
```

### watchpoint.c

```
watchpoint_display()
```

简单的链表遍历，注意首先判断链表为空的情况也就是头指针为 NULL

## d 指令

### ui.c

类似的完善基础框架

```
cmd_table[]
```

```
//pal: 删除监视点
{ "d", "Delete watchpoint", cmd_d },
```

```
cmd_d()
```

摘取第一个参数 token，将其转换成整型作为参数传入 free\_wp()

### watchpoint.c

```
get_wp()
```

获取需要删除的监视点，根据 NO，遍历链表即可

get\_prev\_wp()

获取需要删除的监视点的前一个监视点，原理同上

free\_wp()

利用函数 get\_wp() 和 get\_prev\_wp()，再加上链表增删结点的操作即可

## 测试

首先设置 3 个监视点，然后执行第一条指令后由于寄存器 t0 发生改变而导致停摆，用 info w 输出信息可知实现监视点功能实现测试成功，然后再删除第一个监视点，再用 info w 输出可知删除监视点功能测试成功

```
(nemu) w $t0
watchpoint no: 0      expr: $t0      value: 0
(nemu) w $t1
watchpoint no: 1      expr: $t1      value: 0
(nemu) w $t2
watchpoint no: 2      expr: $t2      value: 0
(nemu) si
80100000:  b7 02 00 80          lui  0x80000,t0
watchpoint changed! Please use 'info w' to check!
(nemu) info w
Num      Expr      OldValue      NewValue
2        $t2        0              0
1        $t1        0              0
0        $t0        0              -2147483648
(nemu) si
80100004:  23 a0 02 00          sw   0(t0),$0
(nemu) d $t0
Delete watchpoint success!
(nemu) info w
Num      Expr      OldValue      NewValue
2        $t2        0              0
1        $t1        0              0
(nemu)
```

利用 \$pc == codeAddr 实现断点功能，测试成功

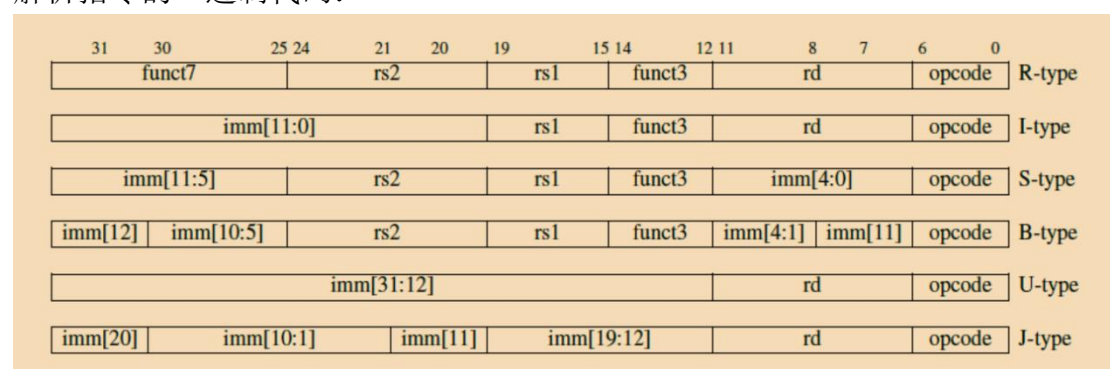
```
(uenv) 
0      2bc == 0x80100008      0
inw    exbl      0x9a9f06      0x6a9f06
(uenv) fuTo M
m9fcpbofuf cpaudeqi bfe926 n26 ,fuTo M, fo cpecki
80100004:  53 90 05 00      zm  0(f0)'20
(uenv) zf
80100000:  p1 05 00 80      jnf  0x80000'f0
(uenv) zf
m9fcpbofuf uo: 0      exbl: 2bc == 0x80100008      9f06: 0
(uenv) M 2bc == 0x80100008
fol pefb' flbe „pefb„
m9fcowe fo 0x80100008-NEWNI
```

## 2.2 PA2 - 简单复杂的机器：冯诺依曼计算机系统

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

#### 一条指令的执行流程

以代码框架中已经实现的 `lw` 指令为例，假设具体指令为：`lw t0, 0x1000(t1)` `nemu` 首先进入 `cpu_exec()` 模拟 `cpu` 工作，然后 `exec_once()` 一条条执行指令。`exec_once()` 通过 `isa_exec()` 执行指令，`update_pc()` 更新 `pc` 寄存器的值（`pc` 寄存器的内容就是内存中存放指令二进制代码的地址）`isa_exec()` 进入 `instr_fetch()` 利用 `pc` 寄存器访问内存得到指令的二进制代码。解析指令的二进制代码：

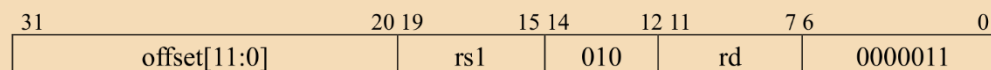


`riscv32` 指令二进制代码不同字段表示的含义如上图所示，代码框架中通过联合体 `Instr`。 `Instr` 利用联合体 `union` 共享相同的内存空间的性质，在体内定义多个结构体用来解析上图中的 6 种指令类型，结构体中的成员直接对应二进制代码中的字段。所有指令的识别区分根据 `opcode` 字段的 [2-6] 位，所以代码框架中定义了一个包含所有要用指令的数组 `opcode_table[]`，`opcode` 字段的 [2-6] 位对应该数组的下标，数组元素的类型为 `OpcodeEntry` 结构体。

然后进入 `idex()`，在 `idex()` 利用当前指令的 `OpcodeEntry` 类型按序进行指令译码和指令执行步骤。一条指令对应一个 `OpcodeEntry` 结构体，`lw` 指令对应 `IDEX(lw, load)`（`IDEX` 是一个宏，对应一个 `OpcodeEntry` 类型，用于快速定义）。结构体内有该指令对应的译码辅助函数 `make_DHelper(lw)` 和执行辅助函数 `make_EHelper(lw)`（代码框架中，会利用变量的别名或函数的别名来简化函数的声明和定义，解释函数的功能，甚至会出现别名的别名-多次递进），以及指令处理的宽度 `width`。

`make_DHelper(lw)` 根据 `lw` 指令的功能：

**lw** `rd, offset(rs1)` `x[rd] = sext(M[x[rs1] + sext(offset)])[31:0]`  
字加载 (*Load Word*). I-type, RV32I and RV64I.  
从地址 `x[rs1] + sign-extend(offset)` 读取四个字节，写入 `x[rd]`。对于 RV64I，结果要进行符号位扩展。  
压缩形式: **c.lwsp** `rd, offset; c.lw` `rd, offset(rs1)`



解析二进制代码中的 `rs1`, `imm(offset)` 立即数和 `rd`, 将它们的内容存放在指令信息整合体 `decinfo`。示例: `lw t0, 0x1000(t1)` 中 `rs1`=寄存器 `t1` 的对应编号, `imm=0x1000`, `rd`=寄存器 `t0` 对应的编号, 但按照功能我们要得到是寄存器 `t1` 存的内容而不是 `t1` 对应的编号, 所以还需要将操作数分为寄存器和立即数分别看待。代码框架中利用 `make_DopHelper(r)` 和 `make_DopHelper(i)` 来实现。得到完指令执行需要的数据信息后, 意味着指令译码阶段结束, 进入指令执行阶段。

由于所有写入指令 (不止有 `lw`, 还有 `lh`, `ld`, `lwu`.....) 的 `opcode` 字段的 [2-6] 位都相同, 都会进入 `make_EHelper(load)`, 所以需要再细分, 代码框架中又定义了一个 `OpcodeEntry` 结构体数组 `load_table[]`。不同的写入指令虽然 `opcode` 字段的 [2-6] 位都相同, 但是 `funct3` 字段不同, 所以结构体数组 `load_table[]` 的下标对应的就是指令的 `funct3` 字段。值得注意的是, 所有写入指令的译码过程都一样, 都可以用 `make_DHelper(ld)`, 所以结构体数组 `load_table[]` 的元素 `OpcodeEntry` 类型只需记录执行辅助函数 `make_EHelper()` 和宽度 `width`。

`lw` 指令执行从 `make_EHelper(load)` 进入到 `make_EHelper(ld)`。在 `make_EHelper(ld)` 利用前面译码得到的指令数据信息 `decinfo` 和 `rtl` 指令实现 `lw` 指令的具体功能: `x[rd]=sext(M[x[rs1] + sext(offset)][31:0])` 将内存地址为寄存器编号为 `rs1` 的寄存器内容和立即数相加的结果的内容存取到寄存器编号为 `rd` 的寄存器中。

上述的 `make_DHelper(ld)` 和 `make_EHelper(ld)` 都用到 `rtl` 指令。`rtl` 指令原本指的是硬件描述语言, 这里代码框架中由于是虚拟机所以用编程的方式实现 `rtl` 指令。可以将 `rtl` 指令理解为一些基础指令, 用于实现一些基本功能, 用它们组合即可实现不同的指令。代码框架中文件 `rtl-wrapper.h` 定义了要使用到的 `rtl` 指令, 这里用 `concat()` 拼接: 例如 `rtl_li()=interpret_rtl_li()`, `interpret_rtl_li()` 定义在文件 `rtl.h` 中。

## 类比

类似, 将要实现的指令按照上面的流程, 添加相应的函数即可, 不同的是不同指令译码过程不同, 要实现的功能不同, 因此使用的 `rtl` 指令也不同。在 `pa2.1` 中按照讲义的要求是查看 `dummy` 程序的反汇编文件了解实现程序需要用到哪些指令, 然后实现, 但由于后续 `pa2.2` 需要实现所有的测试程序, 所以在这里一次性都实现了 (反汇编所有的测试程序, 总结所有的需要实现的指令)。

## 测试

```
hust@hust-desktop: ~/ics2019/nexus-am/tests/cputest
/home/hust/ics2019/nexus-am/am/arch/platform/nemu.mk:27: warning: ignoring old recipe for target 'run'
# Building dummy [riscv32-nemu] with AM_HOME {/home/hust/ics2019/nexus-am}
# Building lib-am [riscv32-nemu]
# Building lib-klib [riscv32-nemu]
# Creating binary image [riscv32-nemu]
+ LD -> build/dummy-riscv32-nemu.elf
+ OBJCOPY -> build/dummy-riscv32-nemu.bin
Building riscv32-nemu
[src/monitor/monitor.c,48,load_img] The image is /home/hust/ics2019/nexus-am/tests/cputest/build/dummy-riscv32-nemu.bin
[src/memory/memory.c,16,register_pmem] Add 'pmem' at [0x80000000, 0x87ffffff]
[src/monitor/monitor.c,20,welcome] Debug: ON
[src/monitor/monitor.c,21,welcome] If debug mode is on, A log file will be generated to record every instruction NEMU executes. This may lead to a large log file. If it is not necessary, you can turn it off in include/common.h.
[src/monitor/monitor.c,28,welcome] Build time: 18:07:09, Jan 3 2024
Welcome to riscv32-NEMU!
For help, type "help"
nemu: HIT GOOD TRAP at pc = 0x80100030

[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 13
dummy
hust@hust-desktop:~/ics2019/nexus-am/tests/cputest$
```

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

### 实验过程

- (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
- (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 riscv32 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。

## 测试

```
← → ics2019 [SSH: hust]

问题 输出 调试控制台 终端 端口

+ CC src/isa/riscv32/exec/system.c
+ CC src/isa/riscv32/exec/special.c
+ CC src/isa/riscv32/exec/exec.c
+ CC src/isa/riscv32/exec/compute.c
+ LD build/riscv32-nemu
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!
hust@hust-desktop:~/ics2019/nemu$
```



### 2.2.3 PA2.3: 运行打字小游戏

#### 实验过程

(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。

## 测试

hello.c 程序测试:

```
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
nemu: HIT GOOD TRAP at pc = 0x80100e60

[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 2188
make[1]: Leaving directory '/home/hust/ics2019/nemu'
hust@hust-desktop:~/ics2019/nexus-am/tests/amtest$
```

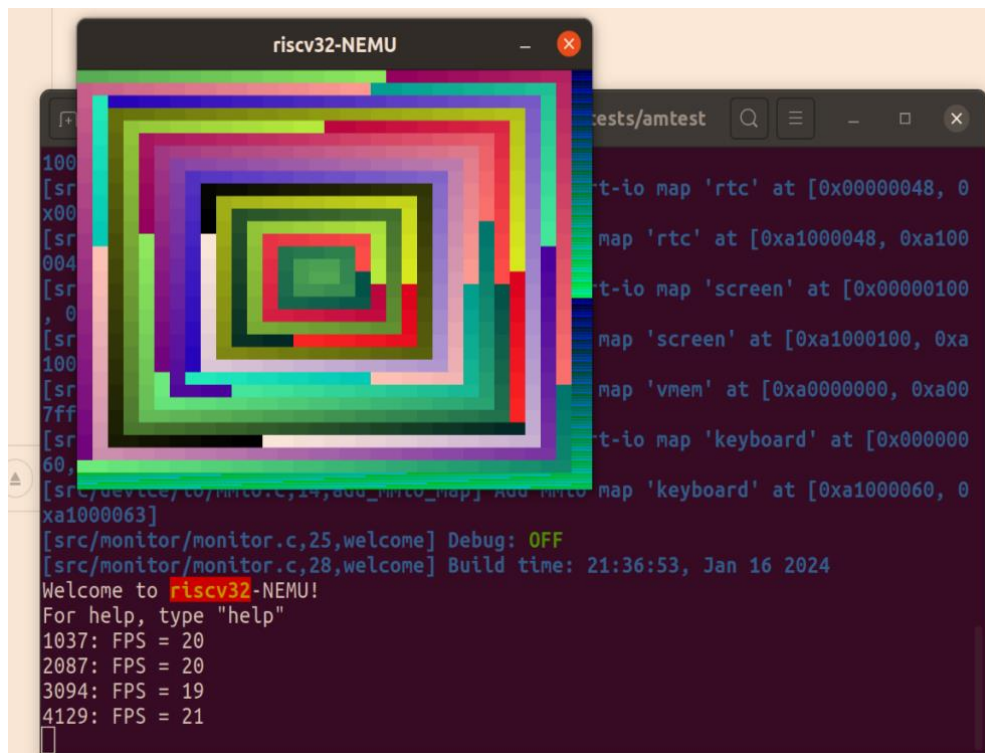
rtc.c 程序测试

```
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).
2000-0-0 2d:2d:2d GMT (10 seconds).
2000-0-0 2d:2d:2d GMT (11 seconds).
2000-0-0 2d:2d:2d GMT (12 seconds).
2000-0-0 2d:2d:2d GMT (13 seconds).
2000-0-0 2d:2d:2d GMT (14 seconds).
2000-0-0 2d:2d:2d GMT (15 seconds).
2000-0-0 2d:2d:2d GMT (16 seconds).
```

keyboard.c 程序测试:

```
hust@hust-desktop: ~/ics2019/nexus-am/tests/amtest
[src/device/io/port-io.c,16,add_pio_map] Add port-io map 'screen' at [0x00000100, 0x00000107]
[src/device/io/mmio.c,14,add_mmio_map] Add mmio map 'screen' at [0xa1000100, 0xa1000107]
[src/device/io/mmio.c,14,add_mmio_map] Add mmio map 'vmem' at [0xa0000000, 0xa007ffff]
[src/device/io/port-io.c,16,add_pio_map] Add port-io map 'keyboard' at [0x00000060, 0x00000063]
[src/device/io/mmio.c,14,add_mmio_map] Add mmio map 'keyboard' at [0xa1000060, 0xa1000063]
[src/monitor/monitor.c,25,welcome] Debug: OFF
[src/monitor/monitor.c,28,welcome] Build time: 21:36:53, Jan 16 2024
Welcome to riscv32-NEMU!
For help, type "help"
Try to press any key...
Get key: 43 A down
Get key: 44 S down
Get key: 45 D down
Get key: 46 F down
Get key: 43 A up
Get key: 44 S up
Get key: 45 D up
Get key: 46 F up
```

video.c 程序测试



slider 测试:



typing 测试



litenes 测试



## 2.3 PA3 - 穿越时空的旅程: 批处理系统

### 2.3.1 PA3.1: 实现自陷操作\_yield()及其过程

#### 实验过程

(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` 字段的值, 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。22

(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。

## 测试

```
[/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: 23:20:55, Jan 16 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,35,init_device] Initializing devices...
[/home/hust/ics2019/nanos-lite/src/irq.c,25,init_irq] Initializing interrupt/exception handler...
[/home/hust/ics2019/nanos-lite/src/proc.c,25,init_proc] Initializing processes...
[/home/hust/ics2019/nanos-lite/src/main.c,33,main] Finish initialization
[/home/hust/ics2019/nanos-lite/src/irq.c,10,do_event] 事件分发
[/home/hust/ics2019/nanos-lite/src/main.c,39,main] system panic: Should not reach here
```

### 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

#### 实验过程

(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\_upload(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 23GPR? 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。

## 测试

dummy 程序测试：

```
[/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 hand
[/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 = 0x830000c8
[/home/hust/hust/ics2019/nanos-lite/src/irq.c,14,do_event] self trapping event
nemu: HIT GOOD TRAP at pc = 0x80100cc0
```

hello 程序测试：

```
[/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 hand
[/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 = 0x830000c8
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!
```



### 2.3.3 PA3.3: 运行仙剑奇侠传并展示批处理系统

#### 实验过程

(1) 修改 Makefile 文件: update: update-ramdisk-single src/syscall.h 修改为 update: update-ramdisk-fsing 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 25 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。

## 测试

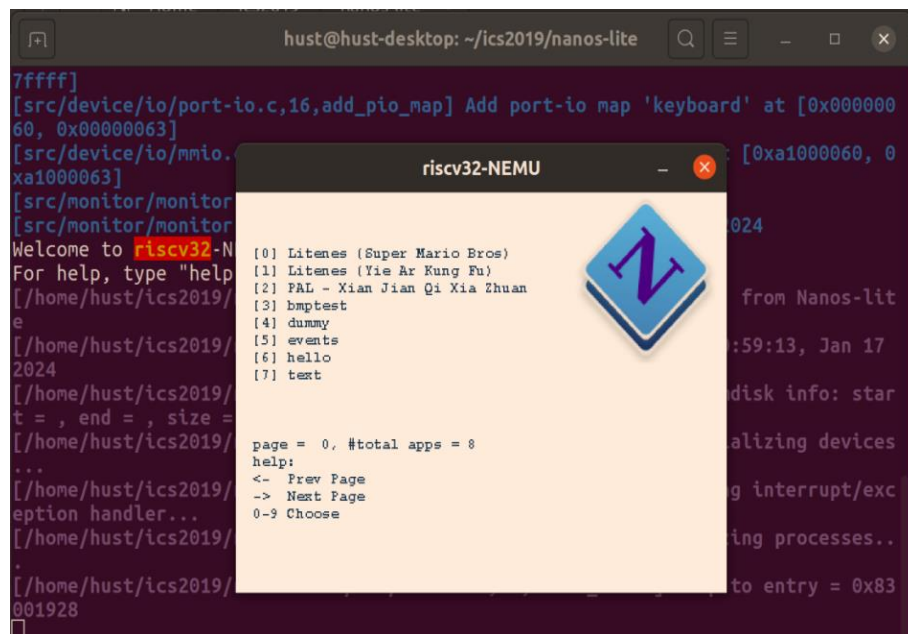
text 程序测试：

```
/home/hust/hust/ics2019/nanos-lite/src/loader.c:52,naive_upload] Jump to entry = 0x80100db4
PASS!!!
nemu: HIT GOOD TRAP at pc = 0x80100db4
```

events 程序测试：

```
/home/hust/hust/ics2019/nanos-lite/src/loader.c:52,naive_upload] Jump to entry = 0x80100db4
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
```

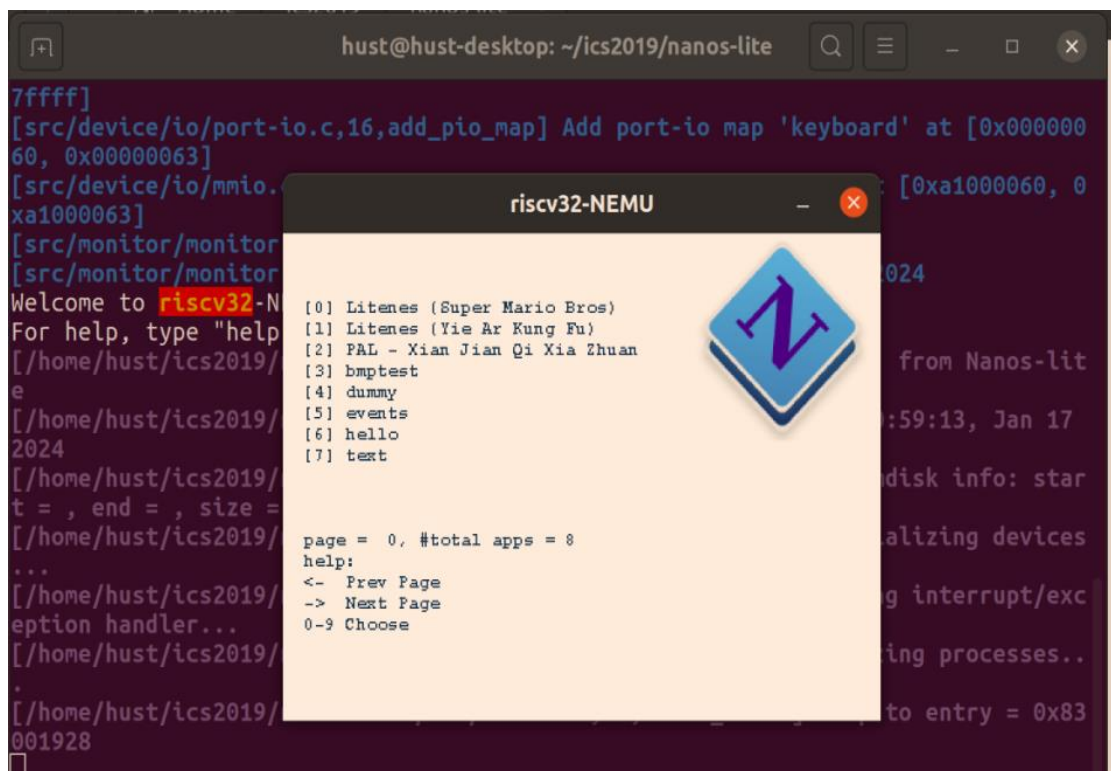
bmp test 程序测试:



仙剑奇侠传游戏初始化画面:



开始菜单:



```
hust@hust-desktop: ~/ics2019/nanos-lite
7fffff]
[src/device/io/port-io.c,16,add_pio_map] Add port-io map 'keyboard' at [0x000000
60, 0x00000063]
[src/device/io/mmio. [0xa1000060, 0
xa1000063]
[src/monitor/monitor
[src/monitor/monitor
Welcome to riscv32-N
For help, type "help
[/home/hust/ics2019/
e
[/home/hust/ics2019/
2024
[/home/hust/ics2019/
t = , end = , size =
[/home/hust/ics2019/
...
[/home/hust/ics2019/
option handler...
[/home/hust/ics2019/
.
[/home/hust/ics2019/
001928

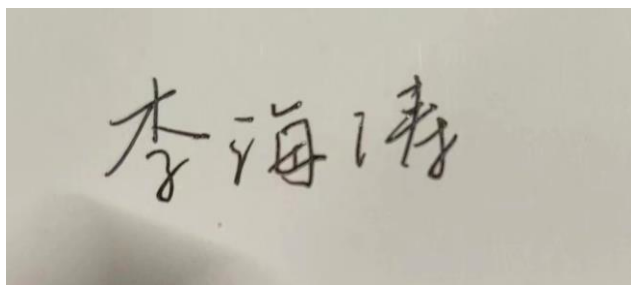
riscv32-NEMU
[0] Litenes (Super Mario Bros)
[1] Litenes (Yie Ar Kung Fu)
[2] PAL - Xian Jian Qi Xia Zhuan
[3] bmp test
[4] dummy
[5] events
[6] hello
[7] text

page = 0, #total apps = 8
help:
<- Prev Page
-> Next Page
0-9 Choose
```

### 3 实验结果与结果分析

见 2 实验方案设计每一小点

电子签名：

A photograph of a handwritten signature in black ink on a light-colored surface. The signature is written in a cursive style and reads "李海彦".