***2023***



**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： | CS2011班 |
| 学 号： | U202011675 |
| 姓 名： | 徐锦慧 |
| 电 话： | 18071680638 |
| 邮 件： | 2939361916@qq.com |
| 完成日期： | 2023-12-17 |

目 录

[1 课程实验概述 1](#_Toc153735691)

[**1.1** **实验目的** 1](#_Toc153735692)

[**1.2** **实验要求** 1](#_Toc153735693)

[2 PA1：最简单的计算机 3](#_Toc153735694)

[**2.1** **方案设计** 3](#_Toc153735695)

[**2.2** **结果分析** 10](#_Toc153735696)

[**2.3** **实验必答题** 12](#_Toc153735697)

[**2.4** **心得体会** 14](#_Toc153735698)

[3 PA2：冯诺依曼计算机系统 16](#_Toc153735699)

[**3.1** **方案设计** 16](#_Toc153735700)

[**3.2** **结果分析** 27](#_Toc153735701)

[**3.3** **实验必答题** 32](#_Toc153735702)

[**3.4** **心得体会** 33](#_Toc153735703)

[4 PA3：批处理系统 35](#_Toc153735704)

[**4.1** **方案设计** 35](#_Toc153735705)

[**4.2** **结果分析** 40](#_Toc153735706)

[**4.3** **实验必答题** 44](#_Toc153735707)

[**4.4** **心得体会** 45](#_Toc153735708)

[参考文献 46](#_Toc153735709)

# **课程实验概述**

## **实验目的**

计算机系统能力是指能自觉运用系统观，理解计算机系统的整体性、关联性、层次性、动态性和开放性，并用系统化方法，掌握计算机软硬件协同工作及相互作用机制的能力。系统能力包括系统分析能力、系统设计能力和系统验证及应用能力三个方面，三个方面相辅相成，共同构成计算机专业本科毕业生的基本能力和专业素养。

《系统能力培养综合实践之指令模拟器》就是为了培养计算机专业学生的系统能力而设置的。课程以软件实践为主，内容循序渐进、由浅入深，使学生能够快速入门，且帮助学生通过实践对理论、技术和方法进行巩固和理解，但是又具有高阶性、 趣味性和挑战度，不断激发学生的兴趣和创造性；课程强化系统观，能够强化并检验学生的系统化综合能力；课程结合工程应用，能帮助学生理解计算机系统从底层硬件到高层软件的全套技术，使得学生对计算机系统各层次的技术有更加深刻的认知。

## **实验要求**

1. **安装与配置开发工具链**

基于主流版本的虚拟机平台与开源操作系统Linux，安装基本的代码编辑、编译、调试、版本管理软件与重要插件， 如vim、gcc、gdb、git，并掌握这些软件的基本使用方法，为下一步的系统级开发奠定坚实的基础。

1. **构建基础编译环境**

通过git工具下载模拟器软件，并根据文档完成基础开发环境的配置。 通过此过程充分理解Linux系统中环境变量的功能与重要性，及其在Makefile中的常见使用方式。

1. **打造基础调试工具**

开发一些必要的调试基础设施，如表达式求值功能、监视点、比较器，为解决开发过程中可能遇到的复杂软件缺陷做好准备。

1. **理解模拟器的基本原理**

掌握使用一段程序模拟一条机器指令的基本方法，并能够在系统给出的基本框架下，结合指令集相关文档，实现X86、MIPS或RISC-V等众多主流指令集中的一个。

1. **构造基本运行时环境**

理解运行时环境的基本概念及其在系统中的重要作用，并通过阅读文档，在系统给出的基本框架下，完成主要库函数的功能开发。

1. **构建基本输入输出环境。**

理解计算机系统与输入输出设备交互的基本原理与方式方法， 并依据文档要求，在系统中添加对于键盘、时钟、图形控制器等常见输入输出设备的支持。 在完成上述目标后，系统即能具备较好的可交互性与可展示性。

1. **理解系统调用的基本原理与实现流程**

了解计算机系统权限分级的原因与现状，理解自陷指令的功能与基本流程，理解系统调用的作用，并依据文档要求实现基本的系统调用。

1. **理解文件系统的基本工作原理**

了解文件系统的主要功能，依据文档要求，实现一个简化的文件系统，支撑对复杂应用所包含的独立数据文件的访问，从而使得模拟器能够支持复杂应用。

1. **理解多任务系统的基本工作原理**

了解多进程/任务并行的基本原理，了解虚存的基本原理及其对于多任务并行的重要性，了解分时功能的必要性，实现一个支持上述功能的简易多任务系统。

# **PA1：最简单的计算机**

## **方案设计**

PA的框架代码由4个子项目构成: NEMU，Nexus-AM，Nanos-lite和Navy-apps，它们共同组成了一个完整的计算机系统。而PA1只与NEMU子项目有关,它主要由4个模块构成: monitor、CPU、memory和设备。

NEMU是一个用来执行其它客户程序的程序,可以随时了解客户程序执行的所有信息。为了提高调试的效率，同时也作为熟悉框架代码的练习，我们需要在monitor中实现一个简易调试器，命令具体的格式和功能如表2.1所示：

表2.1 调试器功能表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 命令 | 格式 | 示例 | 说明 |
| 1 | 帮助 | help | help | 打印命令的帮助信息 |
| 2 | 继续运行 | c | c | 继续运行被暂停的程序 |
| 3 | 退出 | q | q | 退出NEMU |
| 4 | 单步执行 | si [N] | si 10 | 让程序单步执行N条指令后暂停 |
| 5 | 打印程序状态 | info SUBCMD | info r | 打印寄存器、监视点状态 |
| 6 | 表达式求值 | p EXPR | p $eax+1 | 求出表达式EXPR的值 |
| 7 | 扫描内存 | x N EXPR | x 10 $esp | 输出以EXPR为起始内存地址的4N个字节 |
| 8 | 设置监视点 | w EXPR | w \*0x2000 | 当表达式EXPR的值发生变化时，暂停程序 |
| 9 | 删除监视点 | d N | d 2 | 删除序号为N的监视点 |

其中，1、2、3号功能系统已实现，下面分别详细说明功能4~9的实现方案。

**2.1.1 单步执行**

单步执行命令的格式为 si [N]，即让程序执行N条指令后暂停，当N没有给出时，默认为1。

我们首先在cmd\_table[] 数组中添加cmd\_si命令，使其可以在help命令中被输出，代码如下：

cmd\_table [] = {

……

{ "si"，"Pause after single-stepping N instructions，si [N]"，cmd\_si },

}

接着在ui.c文件中实现cmd\_si()函数。通过strtok()函数识别命令中的参数，并判断其是否为空。例如在“si n”的命令中识别出si 和n ，从而得知这是一条单步执行n条指令的命令。再调用cpu\_exec()函数并传入参数n，让CPU向前执行n条指令。具体代码如下：

static int cmd\_si(char \*args) {

/\* extract the first argument \*/

char \*arg = strtok(NULL，" ");

int n = 1; // default n = 1

if (arg != NULL) {

n = atoi(arg);

}

cpu\_exec(n); // execute n instructions

return 0;

}

**2.1.2 打印程序状态**

打印寄存器的命令为 info SUBCMD，即打印寄存器或监视点的信息。

同上，我们首先在cmd\_table[] 数组中添加cmd\_info命令：

{ "info"，"Print register status，info SUBCMD"，cmd\_info },

接着在ui.c文件中实现cmd\_info()函数。通过strtok()函数识别命令中的参数，如果为“r”，则调用isa\_reg\_display()输出寄存器信息；如果为“w”则调用wp\_display()输出监视点信息；否则输出相应的报错信息。

isa\_reg\_display()在nemu/src/isa/riscv32/reg.c文件中实现，用于打印寄存器的名字和存储的数值，代码如下：

void isa\_reg\_display() {

for (int i = 0; i < 32; i++) {

printf("%s\t\t%#x\t\t%d\n"，regsl[i]，reg\_l(i)，reg\_l(i));

}

}

wp\_display()在nemu/src/imonitor/debug/watchpoint.c文件中实现，用于打印所有监视点的序号、参数和数值，代码如下：

void wp\_display() {

printf("NUM\tEXPR\tVAL\n");

WP\* wp = head;

while(wp) {

printf("%u\t%s\t%u\n"，wp->NO，wp->exp，wp->val);

wp = wp->next;

}

}

**2.1.3 表达式求值**

表达式求值命令的格式为 p EXPR，即求出表达式EXPR的值。

首先进行词法分析。利用编译原理的相关知识补充表达式匹配规则，并在枚举类型中加上TK\_DERE和TK\_NEGA，以解决含有解引用和负号的表达式。其中空格串的token类型是TK\_NOTYPE，不参加求值过程。具体匹配规则如下：

{"\\+"，'+'}, // plus

{"-"，'-'}, // minus

{"\\\*"，'\*'}, // multiply

{"/"，'/'}, // divide

{"\\("，'('}, // left parenthesis

{"\\)"，')'}, // right parenthesis

{"(0x|0X)[0-9a-fA-F]+"，TK\_HEXINT}， // hexadecimal integer

{"[1-9][0-9]\*|0"，TK\_DECINT}, // decimal integer

{" +"，TK\_NOTYPE}, // spaces

{"=="，TK\_EQ}, // equal

{"!="，TK\_UEQ}, // unqeual

{"&&"，TK\_LOGICAND}, // logic and

给出正则表达式后，就可以在make\_token()函数中识别token。用position变量来指示当前处理到的位置，并按顺序尝试不同的规则来匹配当前字符串。若识别成功就用Log()输出配对信息，同时将匹配类型记录到tokens中。如果token类型是数字或寄存器名，还需要记录其值并判断缓冲区是否溢出。若尝试了所有规则都无法成功，则打印匹配失败的位置信息并退出。关键代码如下：

static bool make\_token(char \*e) {

……

while (e[position] != '\0') {

/\* Try all rules one by one. \*/

for (i = 0; i < NR\_REGEX; i ++) {

if (regexec(&re[i]，e + position，1，&pmatch，0) == 0 && pmatch.rm\_so == 0) {

char \*substr\_start = e + position;

int substr\_len = pmatch.rm\_eo;

Log(……);

position += substr\_len;

assert(nr\_token < MAX\_TOKSIZE); // prevent array subscript overflow

tokens[nr\_token].type = rules[i].token\_type;

if (rules[i].token\_type==TK\_DECINT || ……) {

assert(substr\_len < 32); // check if buffer overflows

strncpy(tokens[nr\_token].str，substr\_start，substr\_len);

tokens[nr\_token].str[substr\_len] = '\0';

}

……

}

接着在eval()函数中对表达式进行递归求值，用BNF的思想给出算数表达式的归纳定义：

<expr> ::= <number> # 一个数是表达式

| "(" <expr> ")" # 在表达式两边加个括号也是表达式

| <expr> "+" <expr> # 两个表达式相加也是表达式

| <expr> "-" <expr>

……

为了在token表达式中指示一个子表达式，我们用整数p和q来指示这个子表达式的开始和结束位置。若p>q，则标记错误信息并返回；若p=q，说明表达式只有一个token且类型应为数字或寄存器名，如果类型符合返回token值，否则报错；若表达式被一对匹配的括号包围，就通过eval(p + 1，q - 1，success)继续递归。如果以上条件都不符合，则说明表达式需要分裂成操作符op和两个子表达式val1、val2，递归求出val1、val2后返回最终结果val1 op val2。eval()函数的流程图如图2.1所示。

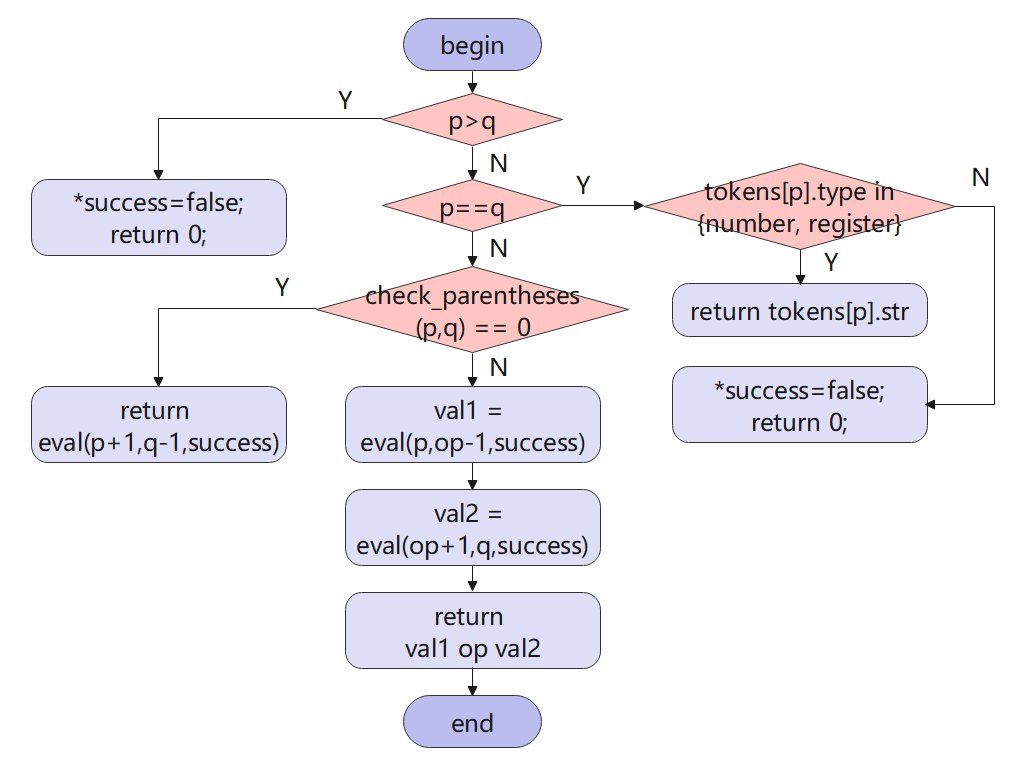


图2.1 eval()函数流程图

其中check\_parentheses()函数用于判断表达式是否被一对匹配的括号包围，同时检查表达式中的左右括号是否匹配。如果不匹配，则说明表达式不符合语法，直接输出报错信息并返回即可。check\_parentheses()返回的情况共有3种：①表达式中左右括号不匹配；②表达式中左右括号匹配，但表达式不是由“(”和“)”包围着的；③表达式既被一对括号包围，又满足左右括号匹配。

最后，在expr()中整合以上函数，并补充cmd\_help和cmd\_p的系统调用，完成对表达式的求值。由于表达式中可能含有解引用和负号，我们需要在expr()中判断并标记。如果tokens[i]的类型为“\*”或“-”，且其前一个字符为{+，-，\*，/，(}中的任意一个，则说明“\*”不是乘号而是解引用，“-”不是减号而是负号，从而正确计算出表达式。expr()的代码如下：

uint32\_t expr(char \*e，bool \*success) {

\*success = true;

if (!make\_token(e)) { \*success = false; return 0; }

// check if expression contains dereference or negative symbol

for(int i = 0; i < nr\_token; i++) {

if(tokens[i].type=='\*' && (i==0 || get\_priority(i-1)<5))

tokens[i].type = TK\_DERE;

else if(tokens[i].type=='-' && (i==0 || get\_priority(i-1)<5))

tokens[i].type = TK\_NEGA;

}

return eval(0，nr\_token-1，success);

}

**2.1.4 扫描内存**

扫描内存的命令为 x N EXPR，即求出表达式EXPR的值，并将结果作为起始内存，以十六进制的形式输出连续的4N个字节。

同上，我们首先在cmd\_table[] 数组中添加cmd\_x命令：

{ "x"，"Output 4N bytes with EXPR as the starting address，x N EXPR"，cmd\_x },

接着在ui.c文件中实现cmd\_x()函数。通过strtok()函数识别命令中的表达式e和字节数n，接着用2.1.3中实现的expr()求出表达式的值，最后输出4n个字节即可。代码如下：

static int cmd\_x(char \*args) {

/\* extract the first argument \*/

char \*N = strtok(NULL，" "); char \*e = strtok(NULL，"\n");

if(N==NULL || e==NULL){ printf("Error expression!\n");return 0; }

int n = atoi(N); bool success;

paddr\_t base\_addr = expr(e，&success);

if(!success){ printf("Error expression!\n");return 0; }

for(int i=0; i<n; i++，base\_addr+=4){

if(i%4 == 0){

if(i != 0) printf("\n");

printf("%#x\t:"，base\_addr);

}

printf("%#x\t"，paddr\_read(base\_addr，4));

}

printf("\n");

return 0;

}

**2.1.5 设置监视点**

监视点的功能主要是监视表达式的值，命令的格式为 w EXPR，当表达式EXPR的值变化时，需要暂停程序并输出相关信息。

为了发挥监视点的功能，我们需要扩展2.1.3中表达式求值的实现，即补充十六进制、寄存器、指针解引用的正则匹配规则：

<expr> ::= <decimal-number>

| <hexadecimal-number> # 以"0x"开头

| <reg\_name> # 以"$"开头

| "\*" <expr> # 指针解引用

……

由于调试器需要满足用户设置多个监视点的需求，我们需要使用链表组织监视点的信息。监视点的结构体如下：

typedef struct watchpoint {

int NO;

struct watchpoint \*next;

char exp[65535];

uint32\_t val;

} WP;

其中NO表示监视点的序号，next表示指向下一个监视点的指针，exp表示监视的表达式，val表示监视点当前的值。在nemu/src/monitor/debug/watchpoint.c文件中，我们用wp\_pool存储所有的监视点，head指针用于组织正在使用中的监视点，free\_指针用于组织空闲的监视点，并通过init\_wp\_pool()对以上结构进行初始化。

为了实现创建监视点的功能，我们需要补充new\_wp()函数。若free\_链表中有空闲的监视点，则从中取出第一个插入head链表并返回该监视点，否则调用assert(0)终止程序。

为了检查监视点的变化情况，我们需要补充check\_wp()函数。首先遍历head链表，用expr()求出当前监视点的新值，并与现有值进行比较。若表达式的值变化，则需要输出相关信息，并更新监视点的值。check\_wp()的代码如下：

bool check\_wp() {

WP\* wp = head; bool flag = false;

while(wp) {

bool success;

uint32\_t new\_val = expr(wp->exp，&success);

assert(success);

if(new\_val != wp->val){

printf("Watchpoint NO.%u has changed:\n"，wp->NO);

printf("expression = %s，old value = %u，new value = %u\n"，wp->exp，wp->val，new\_val);

wp->val = new\_val，flag = true;

}

wp = wp->next;

}

return flag;

}

完成以上函数后，就可以实现对监视点的管理。当用户给出一个待监视表达式时，系统就会通过new\_wp()申请一个空闲的监视点结构，并将表达式记录下来。每当cpu\_exec()执行完一条指令，就调用check\_wp()来比较它们的值有没有发生变化，若发生变化，程序就会因触发监视点而暂停，通过将nemu\_state.state变量设为NEMU\_STOP来实现。最后输出相关提示，并返回到ui\_mainloop()循环中等待用户的命令。

**2.1.6 删除监视点**

与2.1.5类似，删除监视点的命令为 d N，即取消监视序号为N的监视点，由函数free\_wp()实现。若链表head为空，说明没有需要释放的监视点，调用assert(0)并退出；否则遍历head，找到序号为N的监视点，将其从head归还到free\_链表中。注意由于链表的特殊性，需要额外考虑选中监视点为head的头结点的情况。代码如下：

void free\_wp(uint32\_t num) {

if(head == NULL) { assert(0); return 0; }

WP \*wp = NULL，\*tmp = head;

if(head->NO != num){

while(tmp->next->NO != num) {

if(tmp->next == NULL){

printf("Cannot get NO.%u watchpoint\n"，num); return 0;

}

tmp = tmp->next;

}

tmp = tmp->next;

}

wp = tmp; tmp = tmp->next; wp->next = free\_; free\_ = wp;

return 0;

}

## **结果分析**

**2.2.1 单步执行**

实验结果如图2.2所示。输入help命令，系统能输出si命令的相应解释；输入si，程序自动向前执行一条指令；输入si 1和si 3，系统也能执行相应的步数。

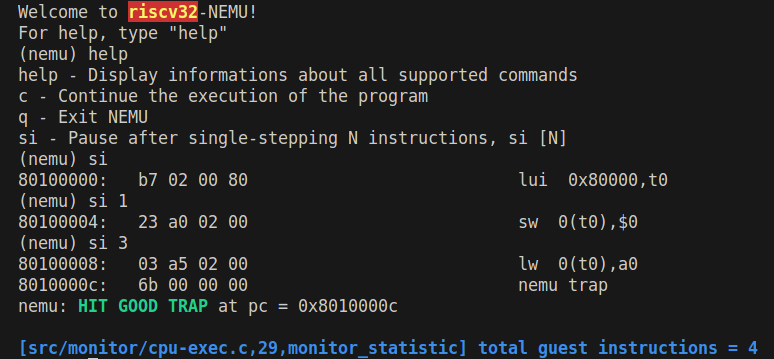


图2.2 单步执行结果

**2.2.2 打印程序状态**

实验结果如图2.3所示。输入help命令，系统能输出info命令的相应解释；输入错误参数时，系统能打印相应的错误提示；输入info r，系统也能正确打印寄存器的信息。

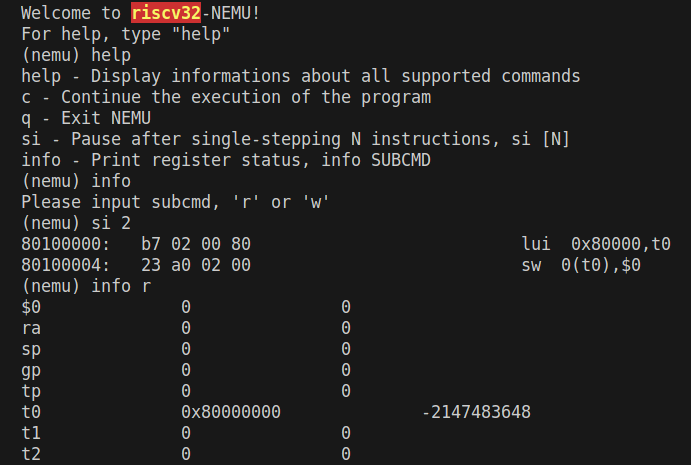


图2.3 打印寄存器结果

**2.2.3 表达式求值**

最初进行表达式求值的检验时，程序出现了如图2.4所示的编译错误。经检查发现，是“(”的正则式匹配部分有误，需要改用“\\(”的格式匹配。

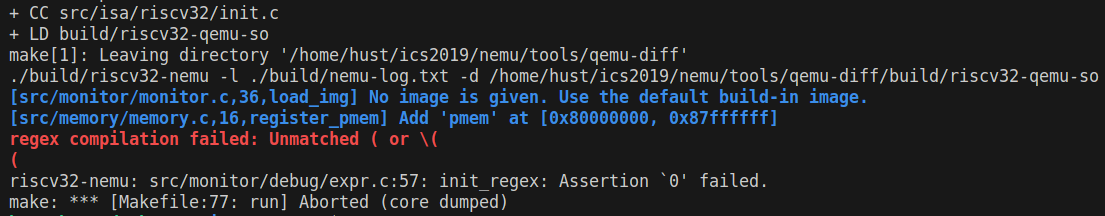


图2.4 表达式求值结果1

修改词法分析部分后，程序能通过编译，但是在执行如图2.5所示的命令时出错，经检查发现，在存储表达式的数字时出现地址越界。

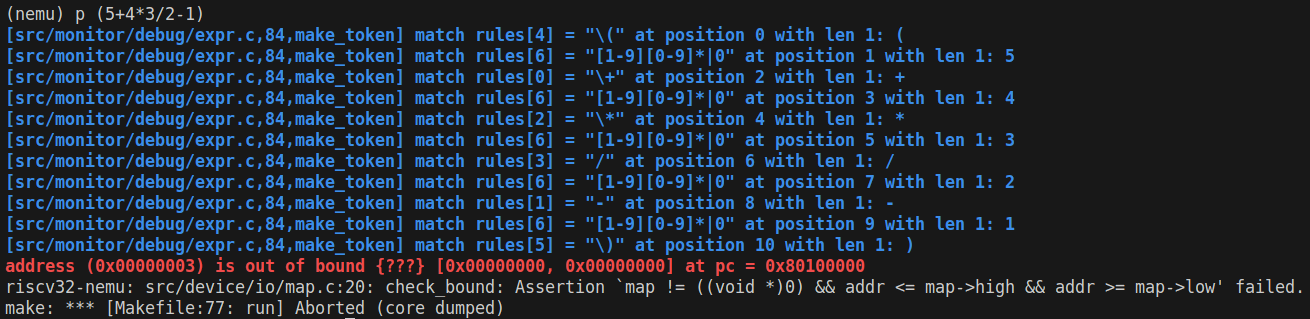


图2.5 表达式求值结果2

修改以上错误后，程序能正确执行，并且能够处理表达式中的空格和负号等情况，如图2.6所示。



图2.6 表达式求值结果3

**2.2.4 扫描内存**

实验结果如图2.7所示。输入x 4 0x80100000命令，系统能正确识别token并计算表达式0x80100000的值，输出以它为起始地址的连续16个字节。

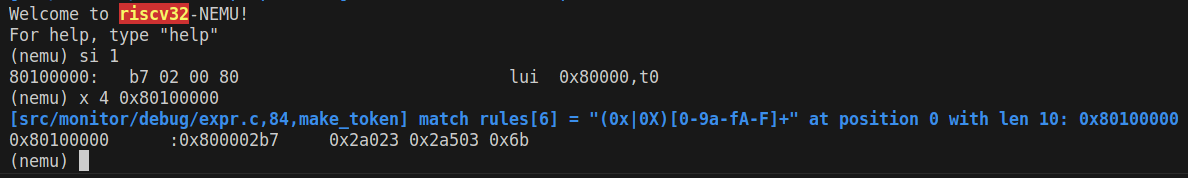


图2.7 扫描内存结果

**2.2.5 设置监视点**

输入w $t0命令，程序最初出现如图2.8的报错。经过检查发现，是由于cmd\_table[]中没有写入对cmd\_w的调用。

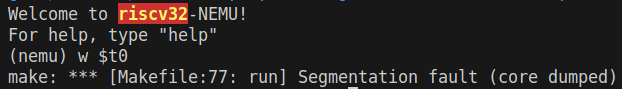


图2.8 设置监视点结果1

修改后再次运行w $t0命令，程序能够正确解析表达式$t0，并设置监视点。输入c命令继续运行，程序能够识别监视点值的变化并输出。继续输入info w命令，程序能够打印出监视点的相关信息。如图2.9所示。

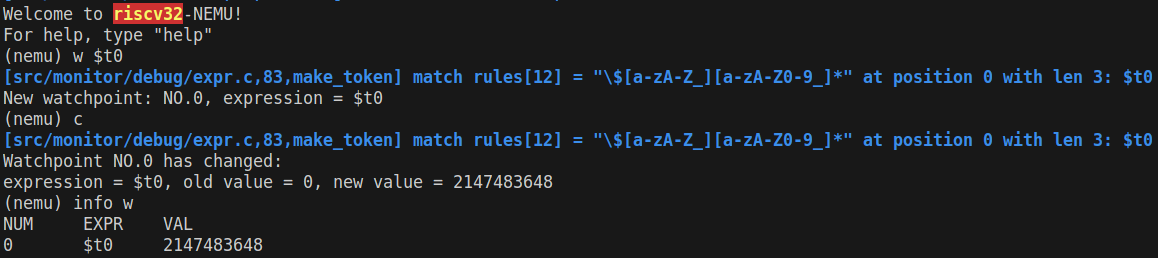


图2.9 设置监视点结果2

**2.2.6 删除监视点**

实验结果如图2.10所示。首先用w $t0命令设置一个监视点，其序号为0，再输命令d 0，能够成功删除。

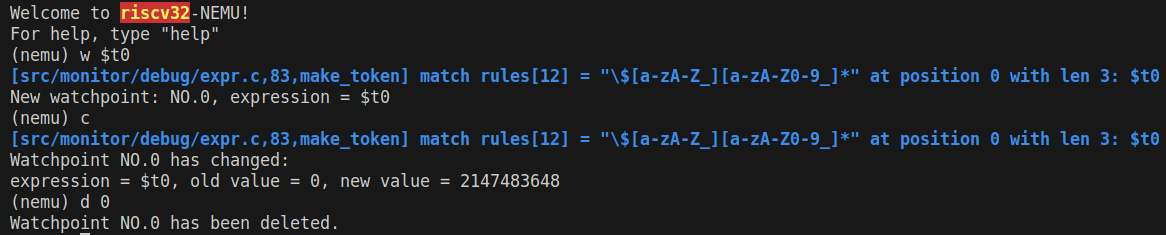


图2.10 删除监视点结果

## **实验必答题**

1. ISA

问：选择的ISA是哪一个？

答：我选择的ISA是riscv32。

1. 理解基础设施

问：假设你需要编译500次NEMU才能完成PA，有90%的次数用于调试。如果没有实现简易调试器，只能通过GDB对NEMU上的客户程序调试。每次调试中,需要花30秒从GDB中获取并分析一个信息。你需要分析20个信息才能排除一个bug。那么这个学期，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,只需要花费10秒从中获取并分析相同的信息。那么这个学期，简易调试器可以帮助你节省多少调试的时间?

答：总调试时间 = 编译次数×调试比例×(获取和分析一个信息所需时间) ×(解决一个 bug 需要的信息数)。

使用GDB时，总调试时间 = 500×0.9×30秒×20 = 27000 秒。使用简易调试器时，总调试时间 = 500×0.9×10秒×20 = 9000 秒，相当于节省18000秒调试的时间。由此可见，如果需要在调试过程中获取并分析更多的信息，简易调试器这一基础设施能带来的好处就更大。

1. 查阅手册

问：理解了科学查阅手册的方法之后，请尝试在选择的ISA手册中查阅以下问题所在的位置。①riscv32有哪几种指令格式? ②LUI指令的行为是什么? ③mstatus寄存器的结构是怎么样的?

答：① 信息在《The RISC-V Reader》文档的25页。riscv32有6种基本指令格式，分别是用于寄存器-寄存器操作的R类型指令，用于短立即数和访存load操作的I型指令，用于访存store操作的S型指令，用于条件跳转操作的B类型指令，用于长立即数的U型指令和用于无条件跳转的J型指令。① 信息在《The RISC-V Reader》的129页。②LUI指令的扩展形式为lui rd，imm，用于高位立即数加载。③信息在《The RISC-V Reader》的103页。mstatus寄存器保存全局中断使能和许多其他的状态，具体结构如图2.11所示。

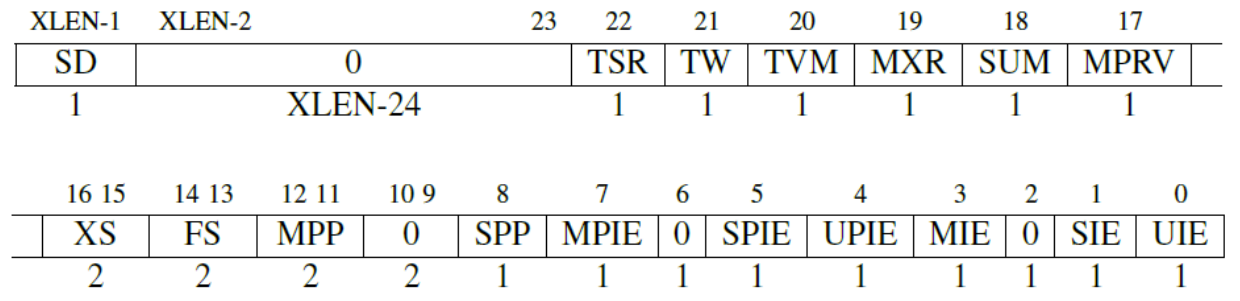


图2.11 mstatus寄存器结构

1. shell命令

问：完成PA1的内容之后，nemu/目录下的所有.c和.h和文件总共有多少代码? 你是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?

答：在PA1分支使用如下命令来获取 nemu/ 目录下的所有 .c和 .h文件的不包含注释和空行在内的代码行数：

find nemu/ -name "\*.c" -o -name "\*.h" | xargs wc -l

完成PA1之后，nemu/目录下的.c和.h文件共有5371行代码；再切换到PA0分支，共有4968行代码，即在PA1中编写了403行。

1. man命令

问：使用man打开工程目录下的Makefile文件，你会在CFLAGS变量中看到gcc的一些编译选项。请解释gcc中的-Wall和-Werror有什么作用? 为什么要使用-Wall和-Werror?

答：①-Wall选项会使编译器对代码中的潜在问题提出警告，包括但不限于未使用的变量、类型不匹配、未初始化的变量等。通过启用 -Wall，开发者可以更容易地发现代码中的问题，提高代码质量和可维护性。②-Werror选项将警告信息视为错误，即如果编译器发出任何警告，它将导致编译过程失败。将警告视为错误的主要目的是强制开发者解决代码中的警告，以确保代码的质量和稳定性。③启用-Wall 和 -Werror可以提高代码质量和可维护性，有助于避免潜在的运行时问题和错误。

## **心得体会**

通过本次实验，我完成了单步执行、打印程序状态、表达式求值等功能，再次复习了编译原理、操作系统等方面的知识，从而进一步巩固所学的计算机知识。通过实现NEMU，我也更深的理解了CPU的原理和调试器的工作内容。

在实验过程中，我们需要结合提示文字理解NEMU的框架代码，这也锻炼了我接手一个陌生项目的能力，能够分阶段分重点的阅读代码，我也体会到了理解代码框架是一个螺旋上升的过程，不必因为看不懂某些细节而感到沮丧。通过“如何阅读手册”章节，我学会了通过目录等信息快速定位目标问题，逐步细化搜索范围，这种筛选信息的能力在今后的学习工作中也十分重要。

同样，在实验中我也遇到了一些问题，例如正则式匹配错误、地址越界等，在分析报错信息、和同学沟通交流后成功解决了。这也锻炼了我的调试、分析和解决问题的能力。

# **PA2：冯诺依曼计算机系统**

## **方案设计**

PA2涉及NEMU，Nexus-AM子项目，主要考察CPU和输入输出设备的知识。具体由3个子任务构成：

1. 根据CPU取指-译码-执行的指令周期，在NEMU中运行第一个C程序dummy。
2. 实现更多的指令，在NEMU中运行所有的cputest。
3. 补充输入输出设备的相关代码，运行打字小游戏。

以下分别介绍这3部分的实现。

**3.1.1 实现dummy程序**

首先在nexus-am/tests/cputest/目录下输入以下命令编译运行dummy.c程序：

make ARCH=riscv32-nemu ALL=dummy run

得到对应的反汇编文件dummy-riscv32-nemu.txt，如图3.1所示：

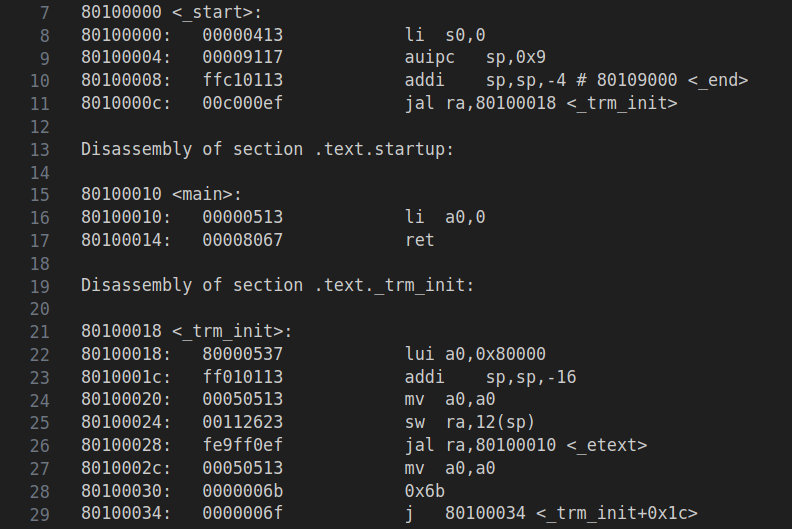


图3.1 反汇编结果

为了成功运行dummy.c程序，需要在NEMU中补充该文件中的所有汇编指令：addi，auipc，j，jal，jalr，li，lui，mv，ret，sw。而由文档可知，系统引入了对译码、执行和操作数宽度的解耦实现和RTL，因此添加一条新指令只需以下两步：① 在opcode\_tabl中填写正确的译码辅助函数，执行辅助函数以及操作数宽度。② 用RTL实现正确的执行辅助函数，使用时需要遵守小型调用约定。阅读代码可以发现，j和lui在系统中均已实现，只需要完成addi，auipc，jal，jalr，li，mv，sw，ret指令。以下分别介绍他们的实现过程。

1. addi/ li/ mv

查阅RISC-V手册可知，addi/ li/ mv指令均为I型指令，且2-5位都是00100，在opcode\_table中处于同一位置，因此合并在一起实现。根据取指-译码-执行的指令周期，首先在decode.c中实现他们的译码函数，代码如下：

make\_DHelper(I) {

decode\_op\_r(id\_src，decinfo.isa.instr.rs1，true);

decode\_op\_i(id\_src2，decinfo.isa.instr.simm11\_0，true);

decode\_op\_r(id\_dest，decinfo.isa.instr.rd，false);

}

该函数可用于解析所有的I类型指令，其中id\_src存储源操作数，id\_src2存储立即数，id\_dest存储目的操作数。在解析过程中，通过decode\_op\_r和 decode\_op\_i宏，分别解析寄存器和立即数，并将结果存储到对应的操作数结构体中。接着，在compute.c中实现他们的执行函数，代码如下：

make\_EHelper(I) {

switch (decinfo.isa.instr.funct3) {

case 0: // li，mv，addi

rtl\_addi(&id\_dest->val，&id\_src->val，decinfo.isa.instr.simm11\_0);

if(decinfo.isa.instr.rs1 == 0) { // li

print\_asm\_template2(li);

} else if (decinfo.isa.instr.simm11\_0 == 0) { // mv

print\_asm\_template2(mv);

} else { // addi

print\_asm\_template3(addi);

}

}

rtl\_sr(id\_dest->reg，&id\_dest->val，4);

}

该函数可用于执行所有的I型指令，并根据decinfo.isa.instr.funct3字段判断是什么指令。如果为0则对应addi/ li/ mv，直接使用rtl\_addi()函数计算目的操作数的值。再通过decinfo.isa.instr.rs1字段区分这3个指令，分别调用print\_asm\_template()打印汇编指令模版。最后通过rtl\_sr()将目的操作数的值写回目的寄存器。

由于这3条指令的2-5位都是00100，结合以上译码函数和执行函数，需要将opcode\_table的第4为改为IDEX(I，I)。

1. auipc

auipc为U型指令，其译码函数make\_DHelper(U)已实现，只需要补充执行函数。auipc主要用于生成一个相对于当前PC的大立即数，在执行中先通过rtl\_add()将立即数与当前程序计数器cpu.pc相加，然后将结果写入目的寄存器id\_dest->reg。最后通过 print\_asm\_template2 打印汇编指令模板。代码如下：

make\_EHelper(auipc) {

rtl\_add(&id\_dest->val，&cpu.pc，&id\_src->val);

rtl\_sr(id\_dest->reg，&id\_dest->val，4);

print\_asm\_template2(auipc);

}

auipc的2-5位为00101，结合译码和执行函数，需要将opcode\_table的第5为改为IDEX(U，auipc)。

1. jal

jal为J型指令，用于无条件跳转。其译码函数如下：

make\_DHelper(J) {

s0 = (decinfo.isa.instr.simm20<<20) + (decinfo.isa.instr.imm19\_12<<12)

+ (decinfo.isa.instr.imm11\_<<11) + (decinfo.isa.instr.imm10\_1<<1);

decode\_op\_i(id\_src，s0，true);

decode\_op\_r(id\_dest，decinfo.isa.instr.rd，false);

}

该函数可以用于解析所有J型指令。在解析过程中，先计算跳转目标的地址，再用过decode\_op\_i()和decode\_op\_r()分别设置源、目的操作数。jal具体的执行逻辑为把下一条指令的地址(pc+4),然后把pc设置为当前值加上符号位扩展的offset，执行函数代码在control.c文件中实现：

make\_EHelper(jal) {

s0 = cpu.pc + 4;

rtl\_sr(id\_dest->reg，&s0，4); // x[rd]=pc+4

rtl\_add(&decinfo.jmp\_pc，&cpu.pc，&id\_src->val); // pc+=sext(offset)

rtl\_j(decinfo.jmp\_pc);

print\_asm\_template2(jal);

}

jal的2-5位为11011，需要将opcode\_table的第27为改为IDEX(J，jal)。

1. jalr/ ret

根据RISC-V手册，ret指令和jalr指令对应的opcode相同，因此只需实现jalr。jalr为I型指令，其译码函数make\_DHelper(I)已实现，只需在control.c中补充执行函数。jalr执行过程和jal类似，但需要把pc设置为x[rs1]+sign-extend(offset)，因此只需改写rtl\_add()函数的调用，代码如下：

rtl\_add(&decinfo.jmp\_pc，&id\_src->val，&id\_src2->val); // pc=(x[rs1]+sext(offset))&~1

jalr的2-5位为11001，需要将opcode\_table的第25为改为IDEX(I，jalr)。

1. sw

sw为S型指令，其译码和执行函数系统均已实现，只需根据对应的宽度修改对应的store\_table：

static OpcodeEntry store\_table [8] = {

EXW(st，1)，EXW(st，2)，EXW(st，4)，EMPTY，EMPTY，EMPTY，EMPTY，EMPTY

};

最后，在decode.h和all-instr.h中申明实现的所有辅助函数，即可通过dummy.c的测试。

**3.1.2 实现所有cputest**

此阶段需要实现更多的汇编指令，来通过nexus-am/tests/cputest/目录下的所有测试用例。其中，string.c和hello-str.c使用了未实现的系统库函数，需要自己编写，并且TODO处的部分函数也需要补充实现。因此本阶段的实现分为这3部分介绍：实现库函数、实现TODO处辅助函数、实现所有指令。

1. **实现所有库函数**
2. 字符串库函数

字符串相关的库函数在nexus-am/libs/klib/src/string.c文件中，包含strlen()，strcpy()，strncpy()，strcat()，strcmp()，strncmp()，memset()，memcpy()，memcmp()。由于这些函数简单常见，此处的具体实现略。

1. vsprintf()

vsprintf()是格式化函数，用于根据提供的格式字符串fmt和可变参数ap进行字符串格式化。它遍历格式字符串，通过调用相应的辅助函数处理每个格式说明符，如“%d”，“%x”，“%u”，“%s”，“%c”。格式化的字符被写入由out指向的内存位置，最终以空字符结尾并返回写入的字符数。函数的流程图如图3.2所示。

其中，my\_putchar()是一个辅助函数，用于“%c”格式，即将单个字符c写入由\*\*str指向的内存位置，然后递增\*str指针。代码如下：

void my\_putchar(char \*\*str，char c) {

if (str) { \*\*str = c; ++(\*str); }

}

my\_puts()用于“%s”格式，即将以null结尾的字符串s写入由\*\*str指向的内存位置。它通过迭代字符串中的每个字符，使用my\_putchar()逐个写入。代码如下：

void custom\_puts(char \*\*str，const char \*s) {

while (\*s) { my\_putchar(str，\*s++); }

}

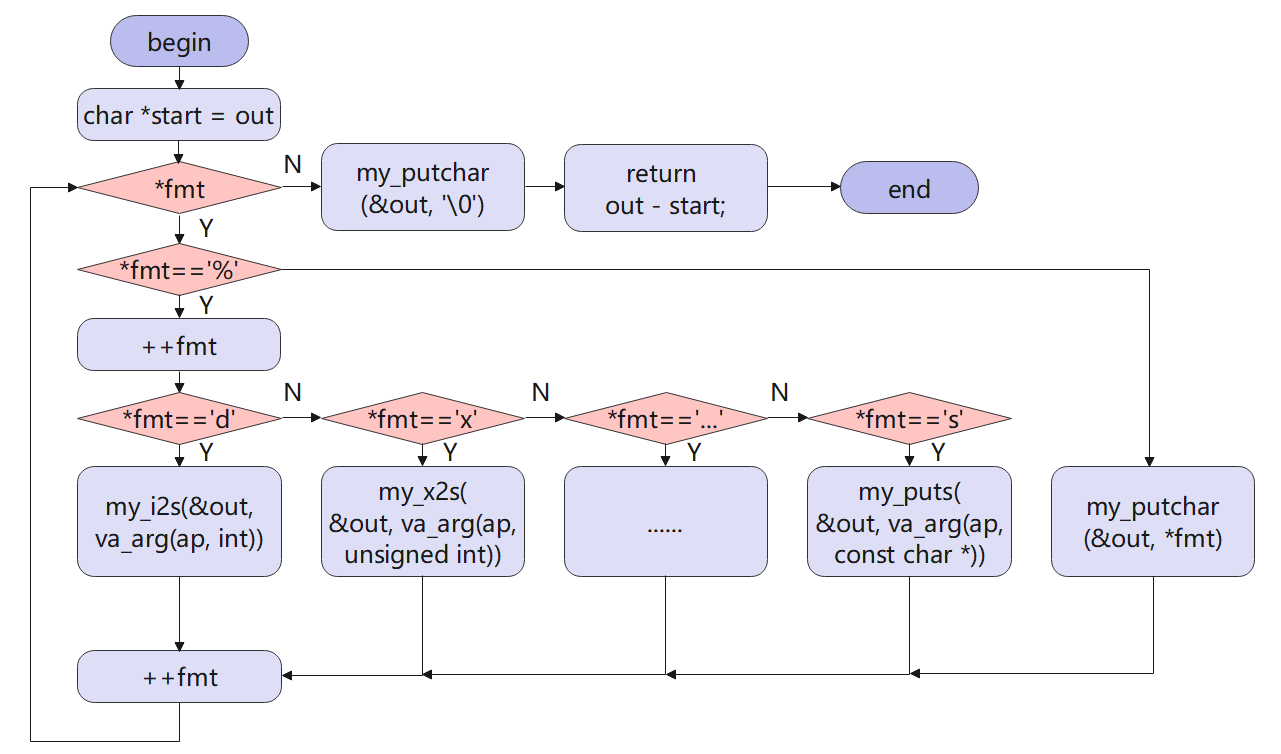


图3.2 vsprintf()流程图

my\_i2s()用于“%d”和“%u”格式，即将整数num转换为其ASCII表示，并写入由\*\*str指向的内存位置。它首先处理符号，递归地将数字除以10直到小于10，再将各位数字写入内存位置。代码如下：

void my\_i2s(char \*\*str，int num) {

if (num < 0) { my\_putchar(str，'-'); num = -num; }

if (num >= 10) { my\_i2s(str，num / 10); }

my\_putchar(str，'0' + (num % 10));

}

my\_x2s()用于“%x”格式，即将无符号整数num转换为其十六进制表示，并写入由\*\*str指向的内存位置。它使用一个固定数组hex\_digits存储十六进制字符，并递归地将数字除以16。代码如下：

void my\_x2s(char \*\*str，unsigned int num) {

const char hex\_digits[] = "0123456789ABCDEF";

if (num >= 16) { my\_x2s(str，num / 16); }

my\_putchar(str，hex\_digits[num % 16]);

}

1. printf()函数

结合C库stdarg.h中提供的宏，可以用上面实现的vsprintf()完成printf()。具体代码如下：

int printf(const char \*fmt，...) {

va\_list args;

va\_start(args，fmt);

char buffer[1000];

int out\_len = vsprintf(buffer，fmt，args);

for(int i =0; buffer[i] != '\0'; i++) {

\_putc(buffer[i]);

}

va\_end(args);

return out\_len;

}

以上代码中，首先创建一个变量参数列表args，调用va\_start()初始化参数列表，指向第一个可变参数。接着通过vsprintf()将格式化后的字符串存储到缓冲区buffer中，并获得格式化后的字符串长度out\_len。再调用\_putc()逐个字符输出到标准输出，最后通过va\_end()清理参数列表。

1. snprintf()函数

和printf()类似，借助stdarg.h中的宏和vsprintf()可以轻松实现snprintf()。具体过程略。

1. **实现TODO处辅助函数**

为了通过所有cputest，还需要实现nemu/include/rtl/rtl.h中RTL伪指令相关的辅助函数。

1. rtl\_not()

函数接受两个参数dest和src1，它们分别是指向RTL目的和源操作数寄存器的指针。函数的作用是将src1中的值按位取反，并将结果存储到dest中。实现过程较简单，此处略。

1. rtl\_sext()

此函数用于对有符号整数进行符号扩展，将源整数的高位符号位复制扩展到目标整数的高位。实现时首先将src1转换为int32\_t类型的临时变量temp，再根据要扩展的字节数，分别进行不同的符号扩展。代码如下：

static inline void rtl\_sext(rtlreg\_t\* dest，const rtlreg\_t\* src1，int width) {

// dest <- signext(src1[(width \* 8 - 1) .. 0])

int32\_t temp = \*src1;

switch(width) {

case 4: \*dest = \*src1; break;

case 3: temp = temp << 8; \*dest = temp >> 8; break;

case 2: temp = temp << 16; \*dest = temp >> 16; break;

case 1: temp = temp << 24; \*dest = temp >> 24; break;

default: assert(0);

}

}

1. rtl\_msb()

这个函数主要是获取有符号整数的最高有效位的值。对于一个width字节的整数，其最高有效位即为整数的符号位。具体实现过程略。

1. rtl\_mux()

这个函数实现了一个三目运算符的功能，根据条件cond选择将src1或src2的值存入目的操作数。具体实现过程略。

1. **实现所有指令**

补充好以上的库函数和其他辅助函数后，在nemu/目录运行以下命令，即可得到cputest中所有测试文件的反汇编结果：

bash runall.sh ISA=riscv32

进而得到需要实现的所有汇编指令。根据指令在opcode\_table中的位置，我们将系统还未实现的合并为3组，如表3.1所示。（load和store相关的指令将在后续讨论）

表3.1 opcode\_table中的指令分组

|  |  |  |  |
| --- | --- | --- | --- |
| x | opcode\_tabble[x] | 译码/执行函数 | 包含指令 |
| 00100 | IDEX(I，I) | make\_DHelper(I)  make\_EHelper(I) | I-type: slli，seqz，sltiu，not，xori，srai，srli，andi |
| 01100 | IDEX(R，R) | make\_DHelper(I)  make\_EHelper(R) | R-type: add，sll，slt，sltu，xor，srl，or，and，mul，mulh，mulhu，div，divu，rem，sub，neg，sra |
| 11000 | IDEX(B，B) | make\_DHelper(B)  make\_EHelper(B) | B-type: beq，beqz，bne，bnez，blt，bgtz，bltz，bge，ble，blez，bltu，bgeu，bleu |

以下分别介绍这些指令的具体实现。

1. I-type指令

I-type指令包括slli，seqz，sltiu，not，xori，srai，srli，andi。其译码函数为make\_DHelper(I)，在3.1.1中已实现；执行函数为make\_EHelper(I)，在3.1.1中完成了addi，li，mv的部分。由于这些指令的decinfo.isa.instr.funct3大都不同，实现时现根据这一字段区分每个指令，并通过RTL伪指令分别处理。若有funct3相同的，则再根据funct7等字段区分。部分代码如下：

make\_EHelper(I) {

switch (decinfo.isa.instr.funct3) {

……

case 1: // slli

rtl\_shli(&id\_dest->val，&id\_src->val，decinfo.isa.instr.simm11\_0);

print\_asm\_template3(slli);

break;

……

case 5:

if(decinfo.isa.instr.funct7) { // srai

rtl\_sari(&id\_dest->val，&id\_src->val，decinfo.isa.instr.rs2);

print\_asm\_template3(srai);

} else { // srli

rtl\_shri(&id\_dest->val，&id\_src->val，decinfo.isa.instr.rs2);

print\_asm\_template3(srli);

}

……

rtl\_sr(id\_dest->reg，&id\_dest->val，4);

}

1. R-type指令

R-type指令的译码函数为make\_DHelper(R)，与I-type的类似，依次识别2个源操作数和1个目的操作数即可，具体代码略。

R-type指令的执行函数为make\_EHelper(R)。为了更准确地区分不同的指令，函数先将funct7左移3位，与funct3相加，得到funct73。再通过该字段的值执行相应的运算，如加法、减法、与、或、异或等等。每个case处理一种指令，并调用相应的RTL操作实现指令功能，调用print\_asm\_template3()打印汇编指令的模板信息。最后通过rtl\_sr()将计算结果写回目标寄存器。部分代码如下：

make\_EHelper(R) {

int funct73 = (((unsigned)decinfo.isa.instr.funct7)<<3) + ((unsigned)decinfo.isa.instr.funct3);

switch(funct73) {

case 0b0000000000: // add

rtl\_add(&id\_dest->val，&id\_src->val，&id\_src2->val);

print\_asm\_template3(add);

break;

case 0b0000000001: // sll

rtl\_shl(&id\_dest->val，&id\_src->val，&id\_src2->val);

print\_asm\_template3(sll);

break;

……

default:

assert(0);

}

rtl\_sr(id\_dest->reg，&id\_dest->val，4);

}

1. B-type指令

B-type的译码函数为make\_DHelper(B)。首先通过按位左移和按位或操作将各个字段组合成目标地址，并存储到临时寄存器s0中。接着用decode\_op\_r()解码指令的两个源操作数rs1和rs2。最后用rtl\_add()将s0和当前指令的地址cpu.pc相加，得到最终的分支目标地址，并将结果存储在decinfo.jmp\_pc中。这个地址在后续的执行阶段用于实现条件分支跳转。代码如下：

make\_DHelper(B) {

s0 = (decinfo.isa.instr.simm12<<12) + (decinfo.isa.instr.imm11<<11)

+ (decinfo.isa.instr.imm10\_5<<5) + (decinfo.isa.instr.imm4\_1<<1);

decode\_op\_r(id\_src，decinfo.isa.instr.rs1，true);

decode\_op\_r(id\_src2，decinfo.isa.instr.rs2，true);

rtl\_add(&decinfo.jmp\_pc，&s0，&cpu.pc);

}

B-type的执行函数为make\_DHelper(B)。根据nemu/include/rtl/relop.h中的关系运算枚举类型，这里使用branch\_table数组来确定分支的条件。执行过程中，先调用rtl\_jrelop()，根据传入的funct3、源和目的操作数来判断是否满足条件，从而实现条件跳转。接着根据branch\_table[decinfo.isa.instr.funct3]的值区分并不同指令，打印相应的汇编指令模板，用于调试和输出。具体代码如下：

static uint32\_t branch\_table[8] = { RELOP\_EQ，RELOP\_NE，RELOP\_FALSE，RELOP\_FALSE，RELOP\_LT，RELOP\_GE，RELOP\_LTU，RELOP\_GEU };

make\_EHelper(B) {

rtl\_jrelop(branch\_table[decinfo.isa.instr.funct3]，&id\_src->val，&id\_src2->val，decinfo.jmp\_pc);

switch(branch\_table[decinfo.isa.instr.funct3]) {

case RELOP\_EQ: print\_asm\_template3(beq);break; // beq，beqz

case RELOP\_NE: print\_asm\_template3(bne);break; // bne，bnez

……

default: assert("Illegal branch opcode!");break;

}

}

1. load相关指令

load相关的指令有lw，lh，lhu，lb，lbu。其中lw，lhu，lbu的译码和执行函数均已实现，只需补充lh和lb的执行函数，在witch语句中根据指令宽度和decinfo.isa.instr.funct3判断并操作即可。代码如下：

make\_EHelper(ld) {

rtl\_lm(&s0，&id\_src->addr，decinfo.width);

switch (decinfo.width) {

case 4: print\_asm\_template2(lw); break; // lw

case 2: // lh，lhu

if(decinfo.isa.instr.funct3 == 1) {rtl\_sext(&s0，&s0，2); print\_asm\_template2(lh);}

else { print\_asm\_template2(lhu); }

break;

case 1: // lb，lbu

if (decinfo.isa.instr.funct3 == 0) {rtl\_sext(&s0，&s0，1); print\_asm\_template2(lb);}

else { print\_asm\_template2(lbu); }

break;

default: assert(0);

}

rtl\_sr(id\_dest->reg，&s0，4);

}

1. store相关指令

store相关指令有sw，sh，sb，系统均已实现。

**3.1.3 输入输出**

此阶段主要包含串口、时钟、键盘、VGA设备的输入输出实现，涉及大量的API调用。以下分别介绍各部分的实现。

1. 串口

系统在nexus-am/am/src/nemu-common/trm.c中已经提供了串口的功能，只需要在nemu/include/common.h中定义宏HAS\_IOE，即可编译运行hello程序。

1. 时钟

首先完善\_\_am\_timer\_init()，通过通过读取RTC地址来获取系统的启动时间，完成初始化。代码如下：

static uint32\_t boot\_time;

void \_\_am\_timer\_init() {

boot\_time = inl(RTC\_ADDR);

}

接着在\_\_am\_timer\_read()中实现宏\_DEVREG\_TIMER\_UPTIME。函数先从RTC地址读取当前时间，减去系统启动时间boot\_time，得到系统运行的时间，并更新\_DEV\_TIMER\_UPTIME\_t结构的高位和低位。代码如下：

case \_DEVREG\_TIMER\_UPTIME:

\_DEV\_TIMER\_UPTIME\_t \*uptime = (\_DEV\_TIMER\_UPTIME\_t \*)buf;

uint32\_t past\_time = inl(RTC\_ADDR);

uptime->hi = 0;

uptime->lo = past\_time - boot\_time;

return sizeof(\_DEV\_TIMER\_UPTIME\_t);

1. 键盘

为了实现readkey test测试，需要在nexus-am/am/src/nemu-common/nemu-input.c中实现\_DEVREG\_INPUT\_KBD的功能：通过读取键盘寄存器，解析键盘状态和键码。函数首先将从键盘地址KBD\_ADDR读取的键盘状态和键码合并到一个 32 位的值中。通过与KEYDOWN\_MASK进行按位与运算，得到键盘是否按下，并记录到kbd->keydown中；通过与KEYDOWN\_MASK进行取反运算，得到键盘的键码，并记录到kbd->keycode中。其中，宏KEYDOWN\_MASK是一个用于检测键盘按键是否按下的掩码，值为0x8000。具体代码如下：

case \_DEVREG\_INPUT\_KBD:

\_DEV\_INPUT\_KBD\_t \*kbd = (\_DEV\_INPUT\_KBD\_t \*)buf;

uint32\_t keyboard\_code = inl(KBD\_ADDR);

kbd->keydown = keyboard\_code & KEYDOWN\_MASK ? 1 : 0;

kbd->keycode = keyboard\_code & ~KEYDOWN\_MASK;

return sizeof(\_DEV\_INPUT\_KBD\_t);

1. VGA

VGA可以用于显示颜色像素，是最常用的输出设备。为了实现绘制图像的功能，需要在nemu-video.c中补充宏\_DEVREG\_VIDEO\_FBCTL的代码。

函数先将传入的buf强制类型转换为\_DEV\_VIDEO\_FBCTL\_t结构，方便后续的操作。接着提取结构中的各个字段，如像素数据pixels、起始坐标x、y、绘制宽度w、高度h、屏幕宽度W和高度H。接着计算需要拷贝的字节数copy\_bytes，避免越界访问。获取显存的起始地址，并将pixels中的图像数据拷贝到显存中对应的位置。最后判断在图像绘制完成后是否需要同步，返回size。代码如下：

case \_DEVREG\_VIDEO\_FBCTL:

\_DEV\_VIDEO\_FBCTL\_t \*ctl = (\_DEV\_VIDEO\_FBCTL\_t \*)buf;

uint32\_t \*pixels = ctl->pixels;

int x = ctl->x，y = ctl->y，w = ctl->w，h = ctl->h;

int W = screen\_width();

int H = screen\_height();

int copy\_bytes = sizeof(uint32\_t) \* (w < W - x ? w : W - x);

uint32\_t \*vmem = (uint32\_t \*)(uintptr\_t)FB\_ADDR;

for (int i = 0; i < h && y + i < H; i++) {

memcpy(&vmem[(y + i) \* W + x]，pixels，copy\_bytes);

pixels += w;

}

if (ctl->sync) { outl(SYNC\_ADDR，0); }

return size;

## **结果分析**

**3.2.1 实现dummy程序**

实验结果如图3.3所示。输入make命令，系统能部分正确编译，但在运行中一直卡住，不能正确运行至出现“HIT GOOD TRAP”。如图3.3所示。

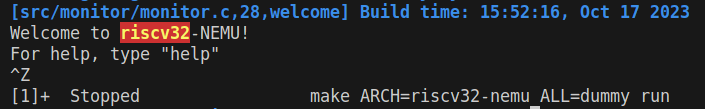


图3.3 dummy错误结果

经过检查调试发现，jal的译码程序make\_DHelper(J)中的移位运算缺少括号，导致程序不能正确运行结束。修改后可以成功执行，如图3.4所示。

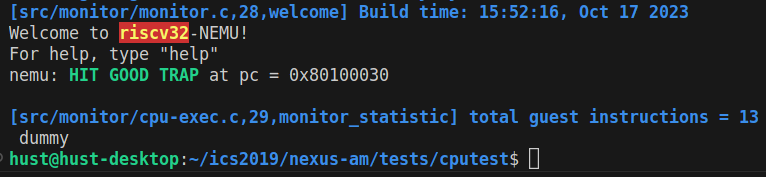


图3.4 dummy正确结果

**3.2.2 实现所有cputest**

由于所有测试文件的指令不尽相同，一起测试时难以发现具体是哪些指令出现问题。因此该阶段先列出所有需要测试的文件，再各自编译运行每个文件，可快速判断出哪些指令是正确的，哪些指令有问题。

在测试if-else.c文件时，程序在pc = 0x80100128处报错，如图3.5所示。

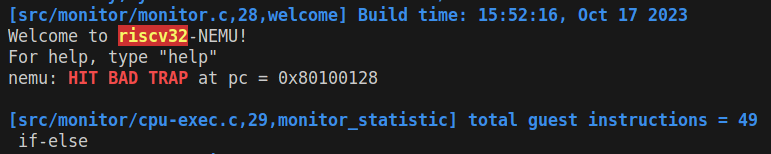


图3.5 if-else错误结果

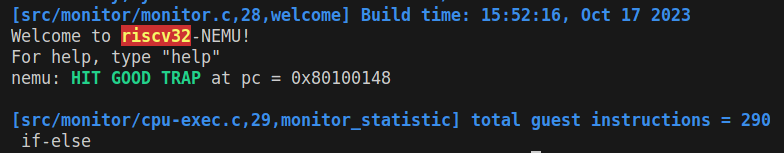


图3.6 if-else正确结果

经过调试发现，实现slt指令的执行函数时缺少signed导致运算出错，正确修改后可以得到如图3.6所示的正确结果。修正后的代码如下：

case 0b0000000010: // slt

id\_dest->val = (signed) id\_src->val < (signed)id\_src2->val;

print\_asm\_template3(slt);

成功后继续测试剩下的cputest。在编译运行load-store.c文件时，程序出现如图3.7所示的报错。

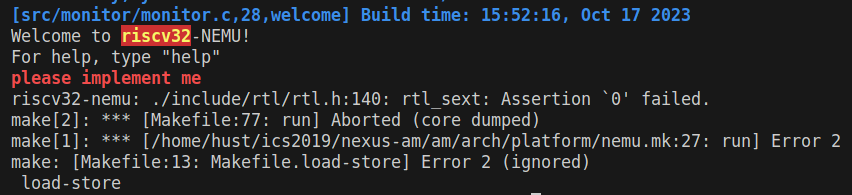


图3.7 load-store错误结果1

由“please implement me”可知，程序是由于有其他函数未实现而报错。由“rtl\_sext: Assertion ‘0’ failed”可知，还需完成rtl\_sext()等函数。完成后发现指令依然报错，如图3.8所示。

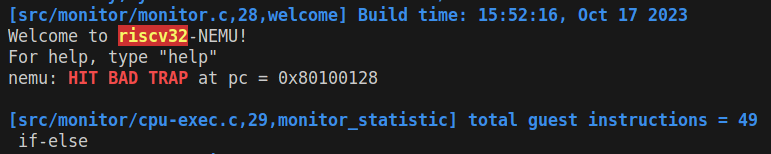


图3.8 load-store错误结果2

经过调试发现，在实现lbu指令的执行函数时，调用rtl\_sr()的位置放错。将其改到函数最后执行，即可得到正确运行结果。如图3.9所示。

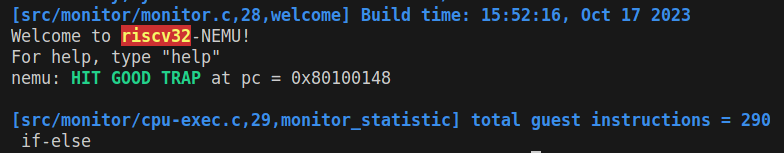


图3.9 load-store正确结果

经过以上修改后，在/nemu目录下输入命令bash runall.sh ISA = riscv32，可以成功运行cputest的所有测试文件。如图3.10所示。

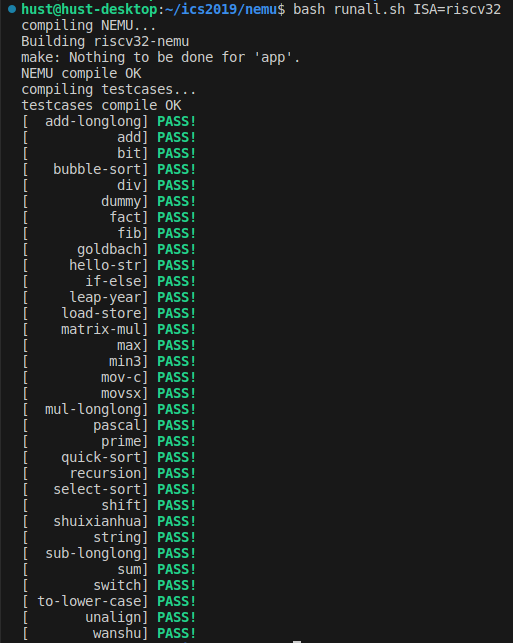


图3.10 所有cputest运行结果

**3.2.3 输入输出**

1. 串口

在nexus-am/tests/amtest/目录下键入make mainargs=h run命令，程序能正确向终端输出10行信息。如图3.11所示。

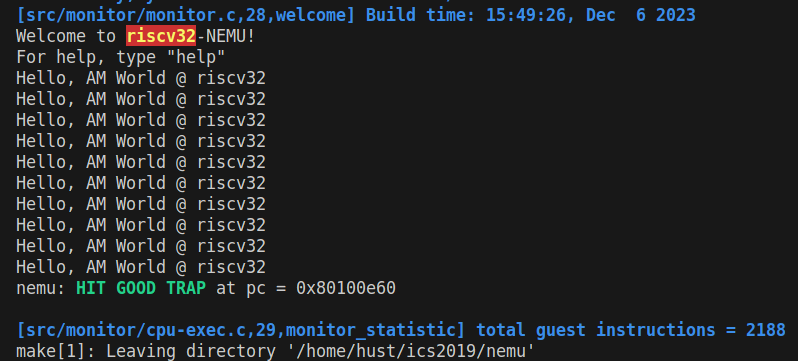


图3.11 串口运行结果

1. 时钟

实现宏\_DEVREG\_TIMER\_UPTIME后，在riscv32-nemu中运行real-time clock test测试，程序能正确的每隔1秒向终端输出信息，如图3.12所示。

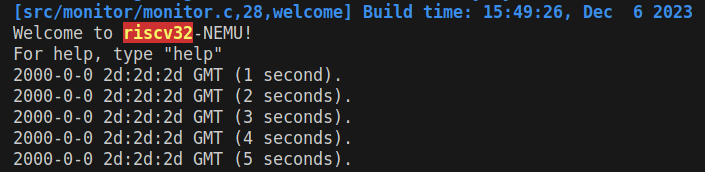


图3.12 时钟运行结果

1. 键盘

实现宏\_DEVREG\_INPUT\_KBD后，在riscv32-nemu中运行readkey test测试。在程序弹出的新窗口中按下各种按键，能够输出正确的按键信息，包括按键名、键盘码、按键状态，如图3.13所示。



图3.13 键盘运行结果

1. VGA

实现宏\_DEVREG\_VIDEO\_FBCTL后，在riscv32-nemu中运行display test测试。程序出现了如图3.14所示的报错。

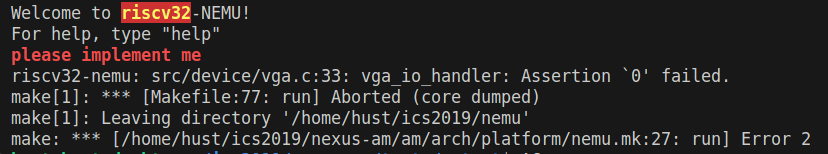


图3.14 VGA错误结果

由“please implement me”可知，程序所需的vga\_io\_handler()还未实现。在系统中查找并实现该函数后，程序弹出的新窗口能正确输出相应的动画效果，如图3.15所示。

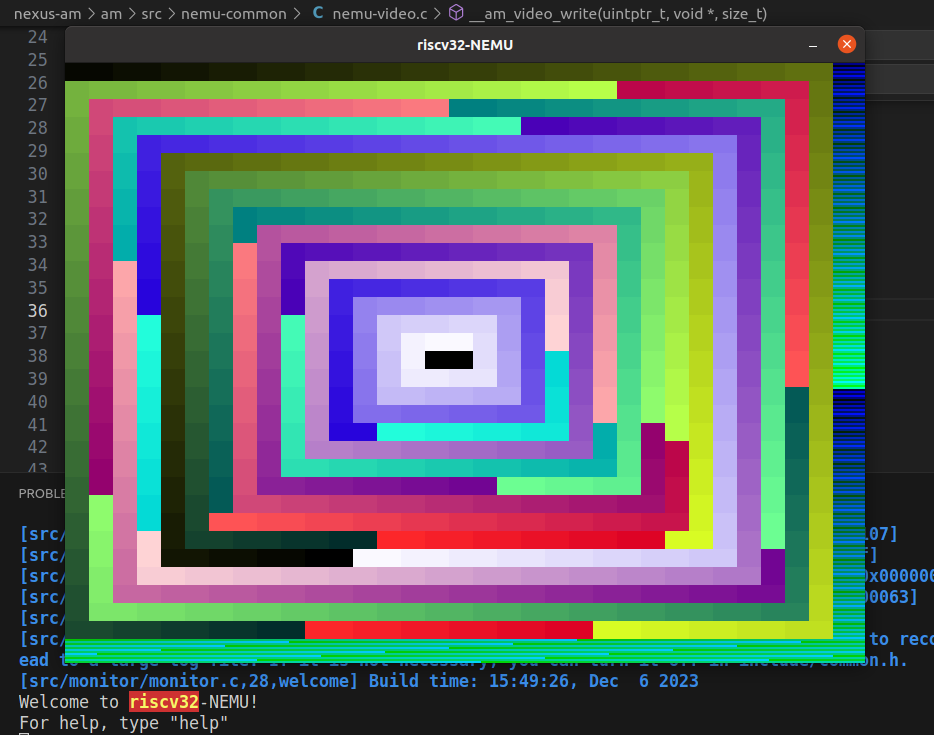


图3.15 VGA正确结果

1. 可展示的计算机系统

完整地实现以上IOE后，就可以运行系统重的幻灯片播放和打字小游戏了。打字小游戏在nexus-am/apps/typing/目录下，来源于2013年NJUCS oslab0的框架代码。为了配合移植，代码的结构做了少量调整，同时对屏幕更新进行了优化，并去掉了浮点数。实现效果如图3.16所示。

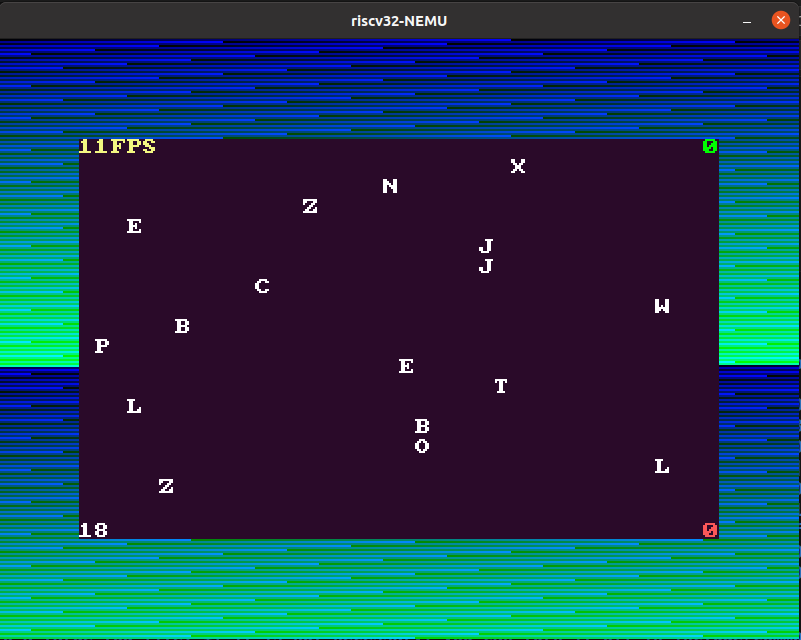


图3.16 打字小游戏

幻灯片播放在nexus-am/apps/slider/目录下，正确运行后，程序将每隔5秒切换images/目录下的图片，效果如图3.17所示。

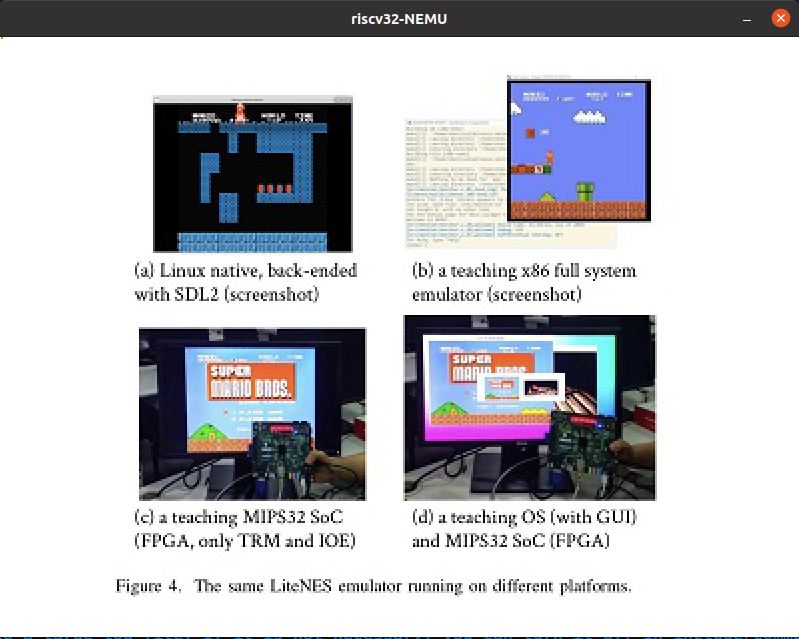


图3.17 幻灯片播放

## **实验必答题**

1. RTFSC

问：请整理一条指令在NEMU中的执行过程。

答：①取址：从指令存储器中读取下一条指令的地址，将其存储在PC中。②译码：根据从指令寄存器中读取的操作码，解析指令并从寄存器文件中读取操作数。③执行：根据译码阶段得到的操作数和指令类型进行相应的运算，得到执行结果。④更新PC：根据指令类型和执行结果，更新程序计数器（PC），指向下一条待执行的指令。

1. 编译与链接

问1：在common.h中添加volatile static int dummy，重新编译后的NEMU含有多少个dummy变量的实体?

答1：一个。因为这里只在该文件中使用static修饰符定义了一次。

问2：再在nemu/include/debug.h中添加volatile static int dummy，重新编译后的NEMU含有多少个dummy变量的实体？

答2：一个。虽然在两个文件中定义了，但这两个都使用了static修饰符，而且debug.h包含了common.h，这样就相当于在debug.h中定义了两个包含 static 修饰符的dummy，而在同一个文件中使用 static 修饰符同一个变量可以写多次定义，但只定义了一次。

问3：为两处dummy变量进行初始化dummy = 0，然后重新编译NEMU。你发现了什么问题?

答3：重新编译时可能会出现链接错误，提示符号重复定义。在之前的情况下，由于dummy变量都被定义为static，它们处于各自文件的作用域内，不会被暴露给其他文件。因此，在链接时不会发生冲突。而现在，通过初始化将其放置在数据段中，这使得dummy的实体变得可见，导致了链接时的冲突。

1. 了解Makefile

问：请描述在nemu/目录下，make程序如何组织.c和.h文件，最终生成可执行文件。

答： Makefile中定义了一些变量和规则，如$ISA表示当前使用的指令集体系结构。这些变量用于定制编译和链接的行为，规则规定了文件之间的依赖关系和如何生成目标文件。当执行make命令时，make工具会根据规则和依赖关系判断哪些文件需要重新构建，最终生成可执行文件。

## **心得体会**

在本阶段实验中，我实现了cputest测试文件所需的所有指令，顺利通过了测试，并完成了串口、键盘、时钟、VGA等设备的输入输出功能，成功运行了幻灯片放映、打字小游戏。

在实现cputest的过程中，我深入研究了RISC-V指令集手册，理解了指令的编码和执行规则。通过逐一运行cputest，我发现并修复了一些潜在的bug，并进一步加深了对CPU执行流程的理解。这个任务对于提高对CPU指令集的熟练度非常有帮助，也让我对计算机体系结构有了更全面的认识。

通过阅读项目的gitbook，我也养成了一些良好的编程习惯，其中一个就是遵守约定。在PA2中，系统定义了RTL寄存器和相应的RTL指令，基于这些定义，原则上可以编写出任意的RTL指令序列并执行。但光靠这些定义，无法避免RTL寄存器相互覆盖的错误，此时小型调用约定就可以很好的解决这类问题。

在第3阶段中，通过实现串口、时钟等输入输出设备，我更好地理解了计算机系统中I/O设备的工作原理，尤其是键盘输入和屏幕输出。

此外，阅读和理解NEMU和Nexus-AM的代码结构，也锻炼了我在大型项目中定位和理解代码的能力。刚接触这个项目时，我还不知道如何下手，但通过RTFSC，能够慢慢明白项目的基本组织结构。在调试的过程中，我对这些模块的理解也逐渐清晰，最终能够游刃有余的在项目中添加新功能。

综上所述，本次PA2实验提供了一个全面的实践机会，通过深入的学习和实践，我不仅加深了对计算机系统的理解，还提升了编程和调试的能力。这对未来的计算机科学学习和职业发展都有着积极的影响。

# **PA3：批处理系统**

## **方案设计**

PA3涉及Nanos-lite子项目，是一个为PA量身订造的操作系统。通过编写Nanos-lite的代码，可以更深刻地认识到操作系统是如何使用硬件提供的机制（即ISA和AM）来支撑程序的运行的，这也符合PA的终极目标。具体由3个子任务构成：

1. 实现自陷操作\_yield()及其过程。
2. 实现用户程序的加载和系统调用，支撑TRM程序的运行。
3. 实现文件系统和批处理系统，运行仙剑奇侠传并展示，提交完整的实验报告。

以下分别介绍这3部分的实现。

**4.1.1 实现自陷操作**

此阶段需要在Nanos-lite中触发一次自陷操作。为了实现最简单的操作系统，硬件需要提供一种可以限制入口的执行流切换方式，即自陷指令。riscv32提供ecall作为自陷指令，具体步骤为：①设置异常入口地址、②触发自陷操作、③保存上下文、④事件分发、⑤恢复上下文。

1. 设置异常入口地址

设置异常入口地址，只需在nanos-lite/include/common.h中定义宏HAS\_CTE。这样系统初始化时便会调用init\_irq()函数，最终通过\_cte\_init()来设置异常入口地址。

1. 触发自陷操作

为了测试异常入口地址是否设置正确，需要调用\_yield()来触发自陷操作，即需要在NEMU中实现raise\_intr()函数来模拟异常响应机制。代码如下：

void raise\_intr(uint32\_t NO，vaddr\_t epc) {

decinfo.isa.sepc = epc;

decinfo.isa.scause = NO;

decinfo.jmp\_pc = decinfo.isa.stvec;

rtl\_j(decinfo.jmp\_pc);

}

函数首先设置异常发生时的返回地址sepc和异常原因scause，将中断处理的入口地址产给目标地址参数jmp\_pc，最终通过rtl\_j()执行跳转，进入中断处理程序。

1. 保存上下文

根据反汇编文件等内容，这阶段需要新增加一些指令：环境调用指令ecall、管理员模式例外返回指令sret、csrrc等控制寄存器相关指令，此处将他们合并在一起实现。新指令的译码函数为make\_DHelper(SYSTEM)，与I-type指令的类似。执行函数为make\_EHelper(SYSTEM)，与PA2的实现方法相同：在switch语句中先根据decinfo.isa.instr.funct3字段判断出具体指令，再分别处理每个case。部分代码如下：

make\_EHelper(SYSTEM) {

switch(decinfo.isa.instr.funct3) {

case 0b000: // ecall，sret

if ((decinfo.isa.instr.val & ~(0x7f)) == 0) { // ecall

raise\_intr(reg\_l(17)，cpu.pc);

} else if (decinfo.isa.instr.val == 0x10200073) { // sret

decinfo.jmp\_pc = decinfo.isa.sepc + 4;

rtl\_j(decinfo.jmp\_pc);

}

break;

case 0b001: // csrrw

s0 = read\_csr(decinfo.isa.instr.csr);

write\_csr(decinfo.isa.instr.csr，id\_src->val);

rtl\_sr(id\_dest->reg，&s0，4);

print\_asm\_template3(csrrw);

break;

……

接着重新组织\_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：

struct \_Context {

uintptr\_t gpr[32]，cause，status，epc;

struct \_AddressSpace \*as;

};

然后就可以在\_\_am\_irq\_handle()中输出上下文c的内容，通过简易调试器观察触发自陷时的寄存器状态，从而检查\_Context实现是否正确。

1. 事件分发

\_\_am\_irq\_handle()会把执行流切换的原因打包成事件，然后调用已经注册的事件处理回调函数，将事件交给Nanos-lite来处理。此部分需要在函数中补充编号为\_EVENT\_YIELD的自陷事件：

switch (c->cause) {

case -1: ev.event = \_EVENT\_YIELD; break;

接着在do\_event()中识别出自陷事件\_EVENT\_YIELD，输出相应信息即可。

**4.1.2 实现用户程序和系统调用**

1. 用户程序

为了把用户程序加载到正确的内存位置和执行用户程序，需要在Nanos-lite中实现loader的功能。loader()函数在nanos-lite/src/loader.c中定义，代码如下：

static uintptr\_t loader(PCB \*pcb，const char \*filename) {

Elf\_Ehdr Ehdr;

ramdisk\_read(&Ehdr，0，sizeof(Ehdr));

for (uint16\_t i = 0; i < Ehdr.e\_phnum; i++) {

Elf\_Phdr Phdr;

ramdisk\_read(&Phdr，Ehdr.e\_phoff + i\*Ehdr.e\_phentsize，sizeof(Phdr));

if (Phdr.p\_type == PT\_LOAD) {

ramdisk\_read((void\*)Phdr.p\_vaddr，Phdr.p\_offset，Phdr.p\_filesz);

memset((void\*)(Phdr.p\_vaddr+Phdr.p\_filesz),0,(Phdr.p\_memsz-Phdr.p\_filesz));

}

}

return Ehdr.e\_entry;

}

函数首先通过ramdisk\_read()从虚拟磁盘中读取ELF头信息，接着遍历ELF文件的所有程序头表，读取程序头信息并判断是否为可加载类型PT\_LOAD。如果可加载，则将ELF文件中的数据加载到内存，最终返回ELF文件的入口地址。

然后在init\_proc()中调用naive\_uload(NULL，NULL)，系统便通过实现的loader来加载第一个用户程序dummy，并在Nanos-lite中触发一个未处理的1号事件。

1. 系统调用

操作系统中的系统调用由自陷指令实现，这一操作被打包成事件\_EVENT\_SYSCALL。和4.1.1中的\_EVENT\_YIELD自陷事件类似，先在\_\_am\_irq\_handle()中把执行流切换的原因打包成事件，交给Nanos-lite来处理，再调用do\_event()函数根据事件类型再次进行分发。

Nanos-lite收到该事件后，会调出系统调用处理函数do\_syscall()进行处理。do\_syscall()首先通过宏GPR1从上下文c中获取用户进程之前设置好的系统调用参数，通过第一个参数系统调用号进行分发。接着添加系统调用，只需要在分发时添加相应的系统调用号，并编写相应的系统调用处理函数sys\_xxx()，最后通过宏GPRx来设置系统调用的返回值。do\_syscall()代码如下：

\_Context\* do\_syscall(\_Context \*c) {

uintptr\_t a[4];

a[0] = c->GPR1，a[1] = c->GPR2，a[2] = c->GPR3，a[3] = c->GPR4;

switch (a[0]) {

case SYS\_yield:

c->GPRx = sys\_yield(); break;

case SYS\_exit:

sys\_exit(a[1]); break;

case SYS\_write:

c->GPRx = sys\_write(a[1]，(void\*)(a[2])，a[3]); break;

case SYS\_brk:

c->GPRx = sys\_brk(a[1]); break;

……

}

return NULL;

}

接着在文件中补充函数sys\_yield()、sys\_exit()、sys\_write()、sys\_brk()，就可以实现系统调用SYS\_yield、SYS\_exit、SYS\_write、SYS\_brk。

**4.1.3 实现文件系统和批处理系统**

1. 实现完整的文件系统

为了维护文件到ramdisk上的映射，完成各种文件操作，需要在nanos-lite/src/fs.c中实现fs\_open()、fs\_read()、fs\_close()、fs\_write()和fs\_lseek()等常用的文件相关函数，并在Nanos-lite和Navy-apps的libos中添加相应的系统调用。此部分较简单常见，具体过程略。

在4.1.2，系统在loader()中直接调用ramdisk\_read()来加载用户程序。而当ramdisk中的文件数量增加之后，此种方式效率会降低。因此，在实现以上文件相关函数后，可以调用他们重新修改load()：

static uintptr\_t loader(PCB \*pcb，const char \*filename) {

Elf\_Ehdr Ehdr;

int fd = fs\_open(filename，0，0);

fs\_lseek(fd，0，SEEK\_SET);

fs\_read(fd，&Ehdr，sizeof(Ehdr));

for(int i = 0; i < Ehdr.e\_phnum; i++){

Elf\_Phdr Phdr;

fs\_lseek(fd，Ehdr.e\_phoff + i\*Ehdr.e\_phentsize，SEEK\_SET);

fs\_read(fd，&Phdr，sizeof(Phdr));

if(Phdr.p\_type == PT\_LOAD){

fs\_lseek(fd，Phdr.p\_offset，SEEK\_SET);

fs\_read(fd，(void\*)Phdr.p\_vaddr，Phdr.p\_filesz);

memset((void\*)(Phdr.p\_vaddr+Phdr.p\_filesz),0,(Phdr.p\_memsz-Phdr.p\_filesz));

}

}

fs\_close(fd);

return Ehdr.e\_entry;

}

1. 把IOE抽象成文件

为了实现一切皆文件的思想，需要扩展之前的文件系统，把串口、设备、VGA等抽象成文件。

把串口抽象成文件，只需在device.c中实现serial\_write()，然后在文件记录表中设置相应的写函数。代码如下：

size\_t serial\_write(const void \*buf，size\_t offset，size\_t len) {

for (int i = 0; i < len; i++) { \_putc(((char\*)buf)[i]); }

return len;

}

把设备输入抽象成文件，需要在device.c中实现events\_read()。函数首先调用read\_key()获取当前的键盘输入，根据不同的键盘输入生成不同的输出信息，然后将这些信息写入buf中，从而模拟读取键盘输入的操作。代码如下：

size\_t events\_read(void \*buf，size\_t offset，size\_t len) {

int keycode = read\_key();

if ((keycode & 0xfff) == \_KEY\_NONE) {

len = sprintf(buf，"t %d\n"，uptime());

} else if (keycode & 0x8000) {

len = sprintf(buf，"kd %s\n"，keyname[keycode & 0xfff]);

} else {

len = sprintf(buf，"ku %s\n"，keyname[keycode & 0xfff]);

}

return len;

}

把VGA显存抽象成文件，首先要在device.c中实现fb\_write()和fbsync\_write()。fbsync\_write()用于同步绘制，直接调用IOE的相应API即可。fb\_write()用于写入图形数据，函数首先计算每个像素的坐标，再调用draw\_sync()确保图形绘制同步，最后调用draw\_rect()以矩形的形式绘制buf中的数据，并返回写入数据的长度。代码如下：

size\_t fb\_write(const void \*buf，size\_t offset，size\_t len) {

int x = offset / 4 % screen\_width();

int y = offset / 4 / screen\_width();

draw\_sync();

draw\_rect((uint32\_t\*)buf，x，y，len / 4，1);

return len;

}

1. 实现批处理系统

下载仙剑奇侠传的数据文件，并放到navy-apps/fsimg/share/games/pal/目录下，更新ramdisk之后，就可以在Nanos-lite中加载并运行游戏了。

为了展示批处理系统，需要添加系统调用SYS\_execve来实现开机菜单程序， 它的作用是结束当前程序的运行，并启动一个指定的程序。参考前面系统调用的实现，只需在do\_syscall()中添加SYS\_execve的case，并调用sys\_execve()处理。代码如下：

int sys\_execve(const char \*fname，char \* const argv[]，char \*const envp[]) {

printf("%s\n"，fname);

naive\_uload(NULL，fname);

}

实现开机菜单程序后，还需要修改SYS\_exit的实现，让它调用SYS\_execve来再次运行/bin/init，而不是直接调用\_halt()来结束整个系统的运行。这样在一个用户程序结束的时候，操作系统就会自动再次运行开机菜单程序，来让用户选择一个新的程序来运行。修改后的sys\_exit()如下：

void sys\_exit(uintptr\_t arg){

// \_halt(arg);

sys\_execve("/bin/init"，NULL，NULL);

}

## **结果分析**

**4.2.1 实现自陷操作**

实现自陷所需的新指令和函数后，实验结果如图4.1所示。系统能够显示在do\_event()中输出的信息“Self trap!”，并触发了main()函数末尾设置的panic()。

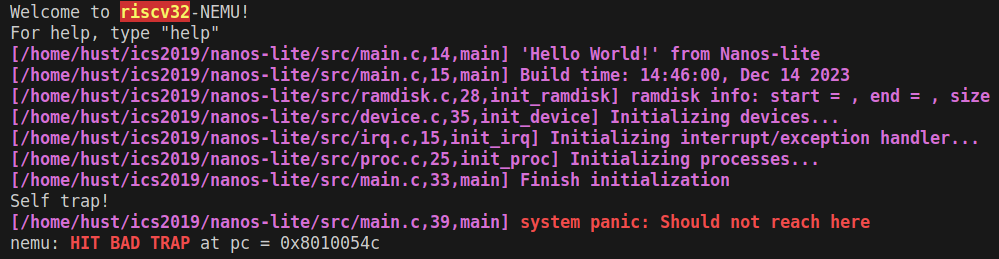


图4.1 实现自陷操作结果

**4.2.2 实现用户程序和系统调用**

在init\_proc()中调用naive\_uload(NULL，NULL)检测用户程序的实现，结果如图4.2所示。执行dummy程序时在Nanos-lite中触发了一个未处理的1号事件，说明loader已经成功加载dummy。

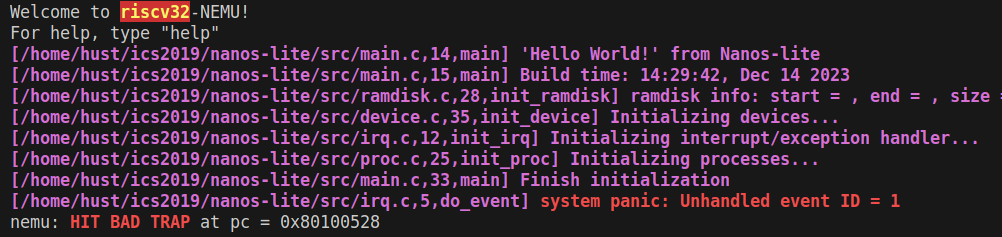


图4.2 实现用户程序结果

实现SYS\_yield和SYS\_exit系统调用后，再次运行dummy程序，系统能够正确输出“HIT GOOD TRAP”信息，如图4.3所示。

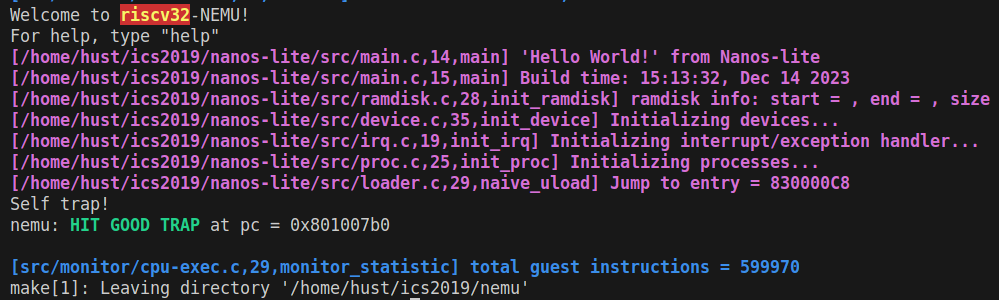


图4.3 实现系统调用结果

继续实现系统调用SYS\_write和SYS\_brk，然后把Nanos-lite上运行的用户程序切换成hello程序，结果如图4.4所示。

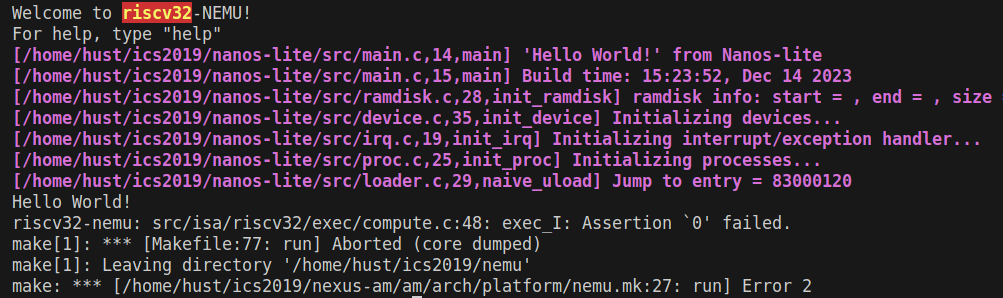


图4.4 hello运行结果1

查看报错信息发现，系统中有I指令未实现而导致程序退出。查看反汇编文件并修改后，能够打印正确信息，如图4.5所示。

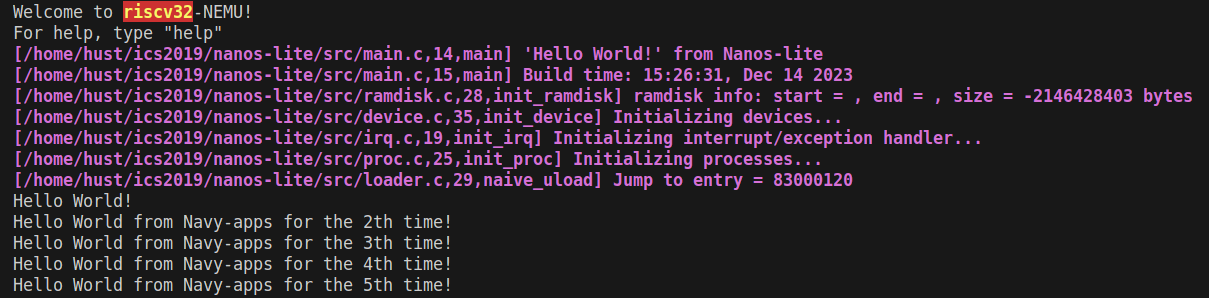


图4.5 hello运行结果2

**4.2.3 实现文件系统和批处理系统**

实现文件操作后，运行测试程序/bin/text，结果如图4.6所示，程序成功输出“PASS!!!”的信息。

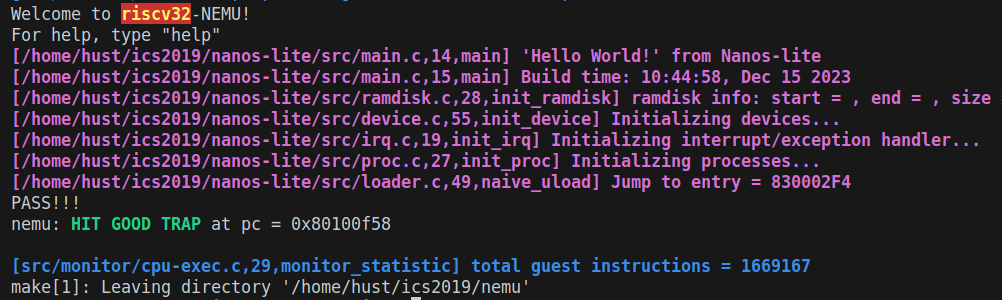


图4.6 text运行结果

把串口和设备输入抽象成文件后，在Nanos-lite加中载/bin/events，结果如图4.7所示，无法输出时间事件和按键事件的信息。

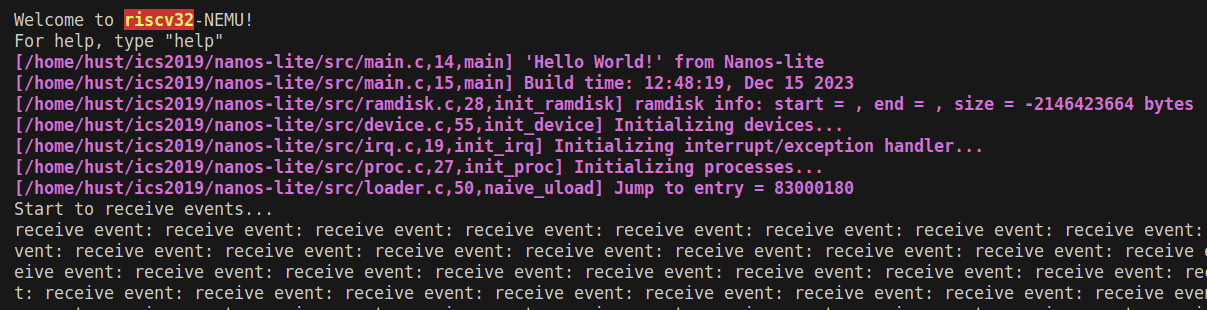


图4.7 events运行结果1

经过调试后发现，由于PA2中实现的sprintf()和printf()有问题，导致此处的输出出错。修改后的正确结果如图4.8所示。

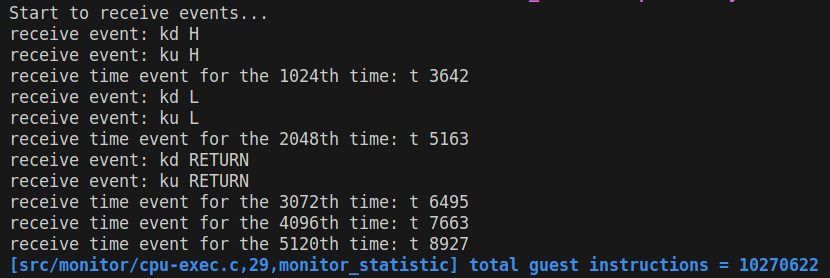


图4.8 events运行结果2

把VGA显存抽象成文件后，在Nanos-lite中加载/bin/bmptest，结果如图4.9所示，屏幕上成功显示Project-N的logo。

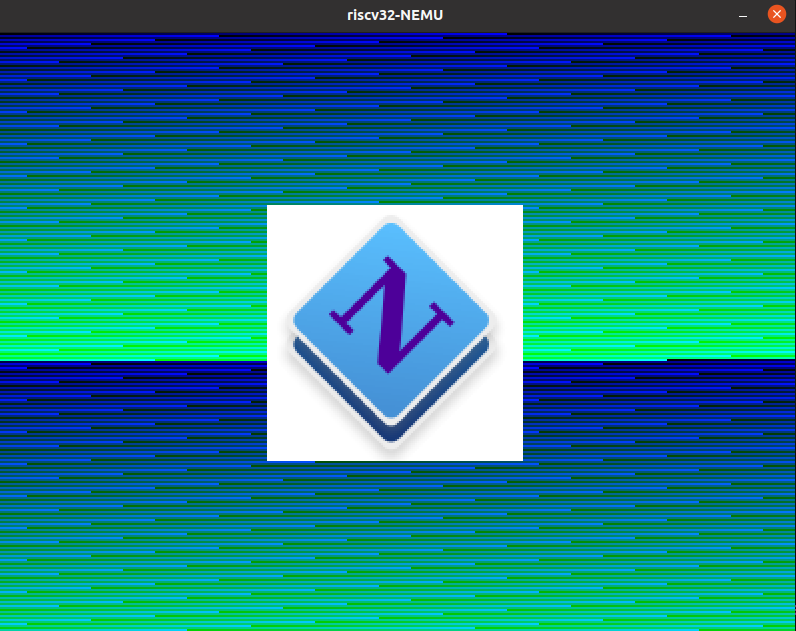


图4.9 bmptest运行结果

接着下载仙剑奇侠传的数据文件，并放到navy-apps/fsimg/share/games/pal/目录下，在Nanos-lite中加载并运行/bin/pal，游戏运行效果如图4.10所示。



图4.10 游戏运行结果

最后实现实现开机菜单程序，运行/bin/init展示批处理系统，结果如图4.11所示。



图4.11 init运行结果

## **实验必答题**

问：运行仙剑奇侠传时会播放启动动画，通过navy-apps/apps/pal/src/main.c中的PAL\_SplashScreen()函数来播放，像素信息存放在数据文件mgo.mkf中。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，并且更新到屏幕上的？

答：①读取游戏存档：首先调用libc中的fread()函数，再进行一系列的系统调用，最终传递到\_syscall\_()进行真正的系统调用。在\_\_am\_irq\_handle()中，会将其封装成\_EVENT\_SYSCALL事件，并转发给nanos-lite处理。nanos-lite在 do\_syscall()函数中接收到该事件，根据系统调用号知道是read系统调用，从而执行sys\_read()帮助函数，在该函数里会调用真正的文件系统操作函数fs\_read()，并进行真正的文件读取操作。 ②更新屏幕：在fs\_write()函数中，判断出要写的文件是/dev/fb设备文件，接着调用draw\_rect()函数，最终转发给\_\_am\_video\_write()。在该函数中会执行out 汇编指令，将数据传送给VGA设备中。接收到数据后会保存在定义的显存中，当之后NDL库向/dev/fbsync 设备文件中写入时，VGA设备最终会调用SDL库来更新画面。

## **心得体会**

在本次PA3中，我实现了自陷操作\_yield()，完成了用户程序的加载和系统调用，并实现了文件系统和批处理系统，最终成功运行了仙剑奇侠传游戏。在完成Nanos-lite子项目的过程中，我深刻理解了操作系统是如何与硬件交互，提供支持给用户程序的运行的。

通过实现\_yield()操作，我了解了操作系统中的进程调度机制。这个机制是操作系统负责管理和切换不同进程的重要组成部分，确保每个进程都能有机会执行。同时也更深入理解了自陷的概念和实现方式，尤其是保存上下文、事件分发等阶段的具体过程。

通过实现loader()等函数，我学会了如何加载用户程序，并将其载入内存执行，重新温习了ELF文件格式的解析、加载程序的地址空间等概念。通过实现简单的系统调用，我理解了系统调用的机制，以及如何通过软中断实现用户态和内核态的切换。

通过实现fs\_open()等文件操作函数，实现一个简易的文件系统，我理解了文件系统的组织结构和文件的读写操作，并通过虚拟文件系统，更深刻地体会到“一切皆文件”的思想。

总体而言，通过PA3实验，我对操作系统的内部机制有了更加深入的认识，学到了如何与硬件进行交互、管理进程、加载和执行用户程序、实现文件系统等基本概念。

# 参考文献

[1]KaiHWang,王鼎兴.高等计算机系统结构[M].清华大学出版社,1995.

[2]袁春风,余子濠.计算机系统基础[M].机械工业出版社,2018.

[3]谭志虎.计算机组成原理[M].人民邮电出版社,2021.