

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:	CS1904 班
学 号:	U201915057
姓 名:	姜家辉
电 话:	13461362877
邮 件:	2860590558@qq. com
完成日期:	2022-10-20



目录

1 课程实验概述	1
1.1 课设目的	1
1.2 课设要求	1
1.3 课设资源	1
2 实验方案设计	2
2.1 实验内容概述	2
2.2 各阶段实验方案设计	2
2.2.1 PA0-世界诞生的前夜：开发环境的配置	2
2.2.2 PA1-开天辟地的篇章：最简单的计算机	3
2.2.3 PA2-简单复杂的机器：冯诺依曼计算机系统	9
2.2.4 PA3-穿越时空的旅程：批处理系统	12
2.2.5 PA4-虚实交错的魔法：分时多任务	16
3 实验结果与结果分析	18
3.1 PA0-世界诞生的前夜：开发环境的配置	18
3.2 PA1-开天辟地的篇章：最简单的计算机	19
1) 实现一个简单的调试器	19
2) 实现表达式求值	19
3) 监视点的实现与观察	20
4) 必答题	20
3.3 PA2-简单复杂的机器：冯诺依曼计算机系统	21
1) 运行第一个 C 程序	21
2) 丰富指令集，测试所有的程序	22
3) 实现 I/O 指令，测试打字游戏	22
4) 必答题	24
3.4 PA3-穿越时空的旅程：批处理系统	26
1) 实现自陷操作_yield()及其过程	26
2) 实现用户程序的加载和系统调用	26
3) 运行仙剑奇侠传并展示批处理系统	28
4) 必答题	30
3.5 PA4-虚实交错的魔法：分时多任务	31
1) 实现基本的多道程序系统	31
4 实验总结	33
参考文献	35

1 课程实验概述

1.1 课设目的

探究“程序在计算机上运行”的机理，掌握计算机软硬协同的机制，进一步加深对计算机分层系统栈的理解，梳理大学 3 年所学的全部理论知识，提升计算机系统能力。

1.2 课设要求

课程要求在已经给出的代码框架中实现一个简化的 RISC-V 模拟器，该指令模拟器应该具备以下几个功能：

- 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

1.3 课设资源

课程资源网站链接：<https://course1.istratus.cn/projects/pa/wiki>

2 实验方案设计

2.1 实验内容概述

该课程将整个实验划分为了五个阶段，分别为：开发环境的配置、简单计算机的实现、冯诺依曼计算机系统的实现、批处理系统的实现以及分时多任务的实现，根据各阶段的完成程度来判定最后的得分，其中每个阶段的具体任务会在各阶段实验方案设计中详细介绍。

2.2 各阶段实验方案设计

2.2.1 PA0-世界诞生的前夜：开发环境的配置

该阶段主要是安装相应的虚拟机，以及配置环境变量，熟悉相应的工具和平台并且最后阅读相应的代码框架。

首先是安装相应的虚拟机，在这里统一使用 **Virtual Box**，并且虚拟机镜像版本为统一的 **Ubuntu 20.04**。具体的安装步骤参考网上教程，这里不再介绍。

安装成功之后，拉取远程仓库的代码框架至本地。在主目录下创建一个新的文件夹“**System_project**”，将代码拉取至该文件夹内。其中，各命令如下：

```
mkdir System_project
cd System_project
git init
git clone https://course.cunok.cn/pa/ics2019.git
```

之后，进入文件“**nemu/Makefile.git**”，修改相应的 **STUID** 和 **STUNAME**，并通过以下两条命令提交至仓库。

```
git add -u
git commit -m 'updated id/name'
```

之后，创建一个新的分支 **pa0**，并且编译该项目文件。

```
git checkout -b pa0
make
```

必做任务：创建一个“**Helloworld**”程序，并且通过一个 **Makefile** 文件来编译该程序，并且通过 **gdb** 来调试程序。

其中 **Makefile** 文件内容如下所示：

```
hello: hello.c
```

```
gcc -o hello hello.c
```

最后尝试使用 `gdb` 进行调试。至此，该阶段已经完成。

2.2.2 PA1-开天辟地的篇章：最简单的计算机

在进行该阶段前一定要执行下面这段代码进行分支整理：

```
git commit --allow-empty -am "before starting pa1"
git checkout master
git merge pa0
git checkout -b pa1
```

该阶段一共包含三个小任务，分别为实现一个简易调试器、实现表达式求值和实现监视点的实现与观察。下面分别进行介绍其具体设计过程。

1) 实现一个简单的调试器

该调试器已经成功实现“`help`”、“`q`”、“`c`”三种功能，在这一步，我们还需要实现单步执行功能（`si [N]`），打印当前寄存器状态（`info r`），扫描内存（`x N EXPR`）三个功能。

单步执行功能的实现较为简单。我们重新定义一个新的函数“`static int cmd_si(char *args);`”，在这里参数就是我们输入的 `N`。这里如果 `args` 为空，也就是需要执行 1 次，如果 `args` 不为空，利用 `sscanf` 读出参数 `num`，然后执行 `num` 次。在这里的执行函数可以直接调用 `cpu_exec(num)`。

打印当前寄存器的状态。由于 `info` 命令既可以查看当前寄存器的状态，也可以查看断点的状态。此时定义如下所示的函数“`static int cmd_info(char *args)`”。在实现该函数的时候，需要对输入参数进行判断，如果为“`w`”，那么开始打印断点状态（先直接返回 0，具体实现下面介绍）；如果为“`r`”，那么开始打印寄存器的状态，然后返回 0；如果为空，那么输出提示语句，提示当前指令输入有错，请重新输入，之后返回 0。

其中打印寄存器状态，需要调用函数“`isa_reg_display()`”，这个函数是需要我们自己实现的。原代码框架中已经定义了所有的寄存器的名称（存放在 `regsl` 数组中），以及相应的结构体 `CPU_state` 和相应的变量 `cpu`。我们只需要调用 `cpu.gpr[i]._32`，即可得出当前编号为 `i` 的寄存器的值。最后打印出寄存器的名称，以及当前寄存器的值的十六进制表示和无符号十进制表示即可。

扫描内存。在这里新定义一个函数如下“`static int cmd_x(char *args)`”；传入的参数中应该包含 `N` 和 `EXPR`（调用 `strtok` 函数切分字符串），也因此函数

具体实现的过程中需要考虑到参数不全, 参数过多, 或则参数类型错误这类问题, 并输出提示信息, 返回 0; 之后考虑参数正确的情况, 这里仍然需要将字符串转换为整型数值, 由于表达式的求值 (`expr(char, bool)`) 还未实现这里先定义一个 `paddr_t` 类型的变量 `addr`, 并初始化为 `0x80100000`。之后调用并打印 `paddr_read(addr, 4)` 函数 `N` 次即可, 返回 0。

该阶段分任务已经基本实现。

2) 实现表达式求值

该小节分为两个任务, 一个是表达式求值的实现, 另一个就是自动生成表达式的实现。

a、其中表达式求值的实现如下:

原有的框架已经给出了部分正规表达式以及 `token` 的识别, 但是由于不符合自己的编码习惯, 这里重新定义了一个新的结构体 `mytoken`, 定义如下:

```
Struct mytoken
{
    char flag; //d 表示十进制, h 表示十六进制, r 表示寄存器, o 表示 ==, !=, &&
    等运算符, 其他只有一位的运算符直接用本身表示: +、-、*...
    char str[32]; //用来存放 token 的内容
    uint32_t num; //在这里存放 token 里面的整数, 以及将寄存器中的值也存
    放在这里
    int op; //这里表示运算符的运算数的个数, 1 表示单目运算符, 2 表示双
    目运算符, 0 表示整数、括号或者寄存器
    int pri; //运算符的优先级: 加减为 1, 乘/分为 2, 正负为 3, () 为 4
    }token[1000];
```

同时也新定义了一个函数 `mymake_token` 用来做 `token` 的识别。另外, 在当前阶段, 只能识别出以下几种 `token`: ‘+’ (加号)、‘-’ (减号)、‘*’ (乘号)、‘/’ (除号)、‘+’ (正号)、‘-’ (负号)、‘==’ (等于)、‘!=’ (不等于)、十六进制整数、十进制整数、寄存器 (在这里将所有寄存器前面都添加了一个 `$` 符号, 除了 `$0`)。通过以下正规表达式来识别:

```
const char *pattern = "^0[Xx][0-9a-fA-F]+|^0-9+|^\\$[a-z0-9]+|^\\-|^\\+|^\\*|^\\/|^%|^\\(|^\\)|^==|^!=|^+";
```

之后根据教程上面的框架, 分别定义了函数 `eval` 用来递归实现表达式的求值; `mymake_token` 用来实现将一个长表达式划分为一个个的 `token`; `in_brackets` 用来判断当前运算符是否被一对括号包围着; `check_parentheses` 用来判断当前表

达式是否恰好全部被一对括号包围着；`findop` 用来查找当前子表达式的主运算符。这五个函数的具体实现思路如下所示：

eval: 函数原型为 “`uint32_t eval(int p,int q);`” 该递归函数的实现可以直接参考教程，这里不再过多介绍。

mymake_token: 函数原型为 “`static int mymake_token(char *e,bool *success);`” 该函数首先需要编译正规表达式（`regcomp`）、将编译后的正规表达式和我们的输入串匹配（`regexexec`），当当前字符串已经匹配结束之后，清空编译好的正规表达式（`regfree`）；观察正规表达式的规则可以发现，每次匹配都一定会是匹配当前串的头，匹配成功之后，删除已匹配的字符形成一个新的字符串，然后接着匹配，直至剩余字符串为空或者某个字符匹配失败；在每一次匹配成功之后，将匹配成功的字符串放入 `token[i].str` 中，然后在这里开始进行判断当前匹配成功的字符串是什么运算符、是不是数字、是不是寄存器（并赋值给 `token[i].flag`），如果是数字，直接使用 `sscanf` 将字符串转换成整型变量，并存入 `token[i].num` 中，如果是寄存器的形式，先判断当前寄存器是否存在，如果存在那么读出寄存器的值（通过调用 `isa_reg_str2val` 来实现），如果是运算符，那么同时要对优先级赋值，这里面最重要的一点就是如何判断这个运算符是单目运算符还是双目运算符，观察 `1+2` 和 `1+-2`，可以看出单目运算符的前一个 `token` 一定是一个运算符，或则它就是第一个 `token`，双目运算符前一个 `token` 一定不是一个运算符（也可能是括号、整数）。

in_brackets: 函数原型为 “`static int in_brackets(int i,int j,int in);`”，首先定义两个整型变量 `num1` 和 `num2`，分别用来表示在 `token[i]` 和 `token[in-1]` 内 ‘(’ 和 ‘)’ 的个数，和 `token[in]`、`token[j]` 内 ‘(’ 和 ‘)’ 的个数。如果是 ‘(’，那么 `num` 加一，如果是 ‘)’，`num` 减一。最后如果 `num1>0` 并且 `num1+num2==0`，那么返回 1，表示此时序号为 `in` 的 `token` 恰好被一对括号包围着，反之返回 0。

check_parentheses: 函数原型为 “`static int check_parentheses(int i,int j);`”，该函数和上面的函数类似，只是需要判断 `token[i]` 和 `token[j]` 是否是一对匹配的括号，如果是，那么返回 1，否则返回 0。

findop: 函数原型为 “`static int findop(int i,int j);`”，该函数是查找主运算符，查找规则：非运算符的 `token` 不是主运算符；出现在一对括号中的 `token` 不是主

运算符. 注意到这里不会出现有括号包围整个表达式的情况, 因为这种情况已经在 `check_parentheses()` 相应的 if 块中被处理了; 主运算符的优先级在表达式中是最低的. 这是因为主运算符是最后一步才进行的运算符; 当有多个运算符的优先级都是最低时, 根据结合性, 最后被结合的运算符才是主运算符. 一个例子是 $1 + 2 + 3$, 它的主运算符应该是右边的 $+$ (在这里所有的运算符都是右结合运算符)。

b、自动生成表达式的实现:

代码框架如下所示:

```
void gen_rand_expr() {
    switch (choose(3)) {
        case 0: gen_num(); break;
        case 1: gen('('); gen_rand_expr(); gen(')'); break;
        default: gen_rand_expr(); gen_rand_op(); gen_rand_expr(); break;
    }
}
```

我们需要做的就是完善框架, 包括实现 `choose` 函数、`gen_num` 函数、`gen` 函数和 `gen_rand_op` 函数, 其中我们还需要做的是如何保证生成的表达式的长度是一定的, 以及保证生成的表达式中没有除 0 的行为。

上面四个函数的实现较为简单, 这里不再进行介绍。如何保证生成的表达式长度不会无限长: 这里需要定义一个全局整型变量 `flag`, 并初始化为 0, 在每次调用 `gen_rand_expr()` 之前将 `flag` 加一, 当 `flag` 大于 100 的时候, 就不再执行 `switch` 块, 只执行 `gen_num()` 函数 (如果当前的最后一个字符是 `') '`, 那么还需要先执行一次 `gen_op()`); 如何保证生成的表达式中没有除 0 的行为, 这里不在生成表达式的时候考虑这件事, 直接在表达式求值的时候过滤带有除 0 警告的表达式即可。具体将 `code_format` 修改如下:

```
static char *code_format =
    "#include <stdio.h>\n"
    "#include <signal.h>\n"
    "#include <stdlib.h>\n"
    "char *ERR=\"TRUE\";"
    "void ss(int sig){\n"
    "    ERR=\"ERROR\";"
    "    printf(\"%%s\",ERR);\n"
    "    exit(0);\n"
    "}"
```



```

"int main() { "
"  signal(SIGFPE,ss);"
"  unsigned result = %s; "
"  ERR=\"TRUE\";"
"  printf(\"%%s\",ERR);"
"  printf(\"%%u\", result); "
"  return 0; "
"}";

```

这样，如果生成的表达式中有除 0 的行为，那么就会将 ERR 置为“ERROR”，我们也可以通过 ERR 的值来判断是否出现了除 0 的行为，以及是否需要将该表达式写入 input 文件里面，同时也通过这里来判断是否需要再生成一条表达式。以使得生成的合法表达式的数目刚好符合自己的要求。

c、验证表达式求值

在实现表达式求值以及自动生成表达式之后，开始进行表达式求值的验证，我们可以通过修改 nemu 中相应的输入接口，使得打开 nemu 之后直接进行表达式求值，接着再与结果比较，从而验证表达式求值的正确性。由于我的表达式求值在另一个单独的文件中也已经实现，因此我也可以通过直接修改 main 函数中的输入接口来验证表达式求值的正确性，两种方式大致相同。

最后还需要将上一个小节中的扫描内存中的表达式求值补充完整。至此，该小节结束。

3) 监视点的实现与观察

在这一小节，我们一共需要实现三个功能，分别是实现设置监视点（w EXPR）、删除监视点（d N）、打印监视点信息（info w）三个功能，其中打印监视点是对第一小节中打印内存功能的补充。

a、表达式求值功能的完善

首先我们需要对表达式求值的功能进行补充，在上一小节虽然已经成功识别出了 token，但是对于部分 token 的操作还有待完善，在这一小节先完善这些操作，同时完善这些操作之后，不要忘了再次验证表达式求值功能的正确性。

在上一关卡已经成功区分正号和加号、减号和负号、乘号和指针，但是指针的运算还没有实现。指针后面一般紧跟的是一个地址，当然也可以是一个子表达式。这里需要在 eval 函数中进行完善，如果是指针，那么需要调用 paddr_read 函数，将指针指向的地址中的内容显示出来。

b、监视点的实现

在监视点实现之前，我们需要清楚监视点的结构体中需要有哪些元素：首先是一个整型变量 NO，用来表示监视点的序号；一个字符型指针变量 EXPR，用来表示被监视的表达式；一个整型变量 flag，用来判断当前序号的监视点是否被使用，1 表示正在使用，0 表示未被使用；一个 next 指针。

另外再次定义一个新的结构体 my_wp，定义如下：

```
typedef struct my_wp{
    int NO;
    struct my_wp *next;
} MY_WPLIST;
```

接着开始实现监视点，程序已经分配了一个大小为 NR_WP*sizeof(WP)的空间来用作监视池（wp_pool），并且申请了两个链表分别用来存放已经被使用的监视点(head)的序号，和未被使用的监视点(free_)，这两个链表均是 MY_WPLIST 类型。但是由于在声明的结构体中加入了一个 flag 变量来显示该监视点是否被使用，因此就不再需要单独的一个链表来存放未被使用的监视点。之后则是需要实现申请一个未被使用的监视点，释放一个监视点，显示监视点等三个功能。

申请未被使用的监视点：

new_wp()，函数原型为“int new_wp()”，通过遍历 wp_pool 数组，找到该数组中第一个未被使用的监视点，并返回该监视点的序号；如果所有的监视点都正在被使用，输出提示信息。

insert_wp，函数原型为“int insert_wp(int NO,char *EXPR);”，该函数的作用是向 head 链表中插入一个新的结点。直接声明一个新的结点 now，使 now->NO=NO，接着让监视点池中序号为 NO 的监视点中的 EXPR，flag 修改为相应的值。

set_watchpoint，函数原型为“int set_watchpoint(char *args)”，其中 args 表示由用户传入的表达式。该函数首先调用一次 init_wp_pool（整个代码运行期间只需调用一次，否则会清空原有的监视点），接着调用 new_wp()和 insert_wp()，如果没有插入成功，输出错误原因。

释放一个监视点：

delete_wp，函数原型为“int delete_wp(int NO);”，该函数首先判断当前输入的 NO 所代表的监视点是否在监视点池中，以及是否正在被使用，如果否，那

么输出提示信息。否则，需要将 head 链表中相应的结点释放，并且将监视池中的相应监视点恢复初始状态（调用 free_wp）。

free_wp，函数原型为“void free_wp(int i)”，该函数的作用是将序号为 i 的监视点恢复初始状态。直接对 wp_pool[i]进行操作即可，这里不再介绍。

显示监视点：

这里是在 cmd_info 中完善相应的操作，当输入的命令为“info w”的时候，将会调用 show_watchpoints，显示所有的监视点。

show_watchpoints，函数原型为“int show_watchpoints();”，直接遍历链表 head，然后输出相应的信息。

在实现上面的几个基础功能之后，还需要实现监视点的功能，通过调用表达式求值的功能，在每一次执行一条指令之后，就判断监视点中的表达式的值是否发生了变化，如果发生了变化，那么就暂停指令的执行，然后等待下一条命令。

2.2.3 PA2-简单复杂的机器：冯诺依曼计算机系统

同样，该阶段开始之前也需要进行代码的整理，参考 PA1 中的内容。

该阶段同样分为三个小任务：运行第一个 c 程序；丰富指令集，测试所有的程序；实现 I/O 指令，测试打字游戏。下面分别介绍：

1) 运行第一个 C 程序

这一部分指令的实现倒是不难，很多在组原课设中都接触过，难的地方在于要知道每一个函数的作用，要读懂代码。其中每一条指令的执行按照取指令、译码、执行、更新 PC 这四个阶段依次进行。

其中取指令这一阶段已经实现，在取出相应内存的指令之后，调用 exec.h 文件中的 idex()，接着进行指令的译码和执行。其中译码阶段需要根据指令的 Opcode 字段，来确定该条指令的类型，之后针对不同的类型取出里面的 rd,rs1,rs2 等字段，该阶段就是通过 make_Dhelper()函数实现，其中我们一共需要实现六种类型的解码函数，由于 U, S 型指令已经实现，我们还需要实现 I,B,J,R 等四种指令的解码；

由于对于同一个 opcode 字段可能会对多条指令，这个时候，还需要通过每条指令的 func3 字段来具体区分每条指令的操作。其中 store_table 这些数组就是用来区分 func3 字段的，这里也需要我们进行补充。之后根据 func3 字段进行

具体的执行操作，这是用过 `make_Ehelper()` 函数实现的，针对不同的指令，都有一个对应的函数与之匹配。

这里以 `addi` 和 `jal` 这两条指令的实现为例介绍具体的代码流程。

`addi` 指令是一个 I 型指令，我们需要补充 `make_Dhelper(I)`，来进行解码，具体实现如下：

```
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);
}
```

另外，由于其 `opcode` 字段为 4，与其他指令相同，我们还需要添加一个数组，这里将其命名为 `firsti_table`。目前的实现如下：

```
static OpcodeEntry firsti_table[8]={EX(addi),EMPTY,EMPTY,EMPTY,EMPTY,
EMPTY,EMPTY,EMPTY};
```

之后一个相对应的解码函数 `make_EHelper(firsti)`，以及 `make_EHelper(addi)` 的实现可以参考源代码，这里不再介绍。其中不要忘了先在头文件中声明 `make_EHelper(addi)` 这些函数。这个时候 `addi` 这条指令已经实现完毕。

这里需要介绍 `jal` 这类指令，这些跳转指令需要对 `pc` 进行操作，因此其执行函数和上面的略微有些不同，具体实现如下所示：

```
make_EHelper(jal){
    s0 = cpu.pc + 4;
    rtl_sr(id_dest->reg, &s0, 4);
    rtl_add(&decinfo.jump_pc, &id_src->val, &cpu.pc);
    interpret_rtl_j(decinfo.jump_pc);
    print_asm_template2(jal);
}
```

这里需要调用 `interpret_rtl_j()` 函数来更新 PC。

这一阶段至少需要实现以下几条指令：`addi/auipc/jal/jalr` 等四条指令

2) 丰富指令集，测试所有的程序

阶段二一共需要完成两部分内容，第一部分是实现更多的指令功能；第二部分是库函数的实现，这里面的库函数都是一些常见的函数。

a、指令的实现

这一阶段的指令实现和上一阶段类似，就不在这里进行赘述，这里至少需要

实现的指令包括：add/sub/mul/sltu/mulhu/xor/div/or/rem/sra/srl/divu/sll/mulh/and/slt、sltiu/srli/srai/andi/xori/slli、beq/bne/bge/blt/bltu、lb/lh、sb/sh/sw 等指令，这些指令的实现大部分都可以调用 rti 伪指令。只是需要注意的是，在 R 型和 I 型指令中存在一些指令，它们的 opcode 字段和 funct3 字段都相同，因此我们还需要通过 funct7 字段进行区分。

这里以 add/sub/mul 指令为例进行介绍。它们的 opcode 字段均为 c，funct3 字段均为 0，只能在 funct7 字段进行区分。我们可以在 exec.c 文件中新建一张 c_0_table 用来区分每条指令，之后为每一条指令单独实现一个 exec_name() 函数；也可以直接在 make_EHelper(add) 中，通过区分 funct7 字段，在同一个函数中通过 switch-case 块实现这三条指令的功能。我是选用的第二种方式。

b、库函数的实现

这里的库函数只在 string.c 和 stdio.c 文件中，我们需要实现的有 strcmp/strcat/strcpy/memset/memcmp/sprintf 等几个函数，这是通关的基本要求，之后还应该将 memcpy 等其他几个函数补充完整。

这里只介绍 sprintf 的实现。sprintf 实现的一个最大的难点就是参数数目是可变的。但是我们可以通过使用 stdarg.h 文件中定义的宏来获取每个参数，具体的用法通过“man stdarg”指令来查看，这里面还提供了一个例子以供参考。之后就是对每一个参数的处理，这里过关的最低要求是要实现 %s 和 %d。这里可以定义几个函数分别针对这几种情况进行处理：_print() 相当于解码函数，分离出参数的类型；_printc() 将字符输入进指定的字符串；_prints() 将 %s 中的参数内容写入指定的字符串；_printd() 将 %d 中的参数内容写入指定的字符串。

之后将库函数补充完整后，再次运行测试程序，验证程序的正确性。

3) 实现 I/O 指令，测试打字游戏

在当前关卡，为了测试打字游戏，还需要实现串口、时钟、键盘、VGA 等四种设备，这里部分设备的实现还需要进行指令集的完善，这里不再进行介绍。

其中串口的实现主要是 printf 函数的实现，注意在实现 printf 函数的过程中主要调用的就是 _putc() 函数，但是这里要注意一点，一定要先定义了宏 HAS_IOE 之后才能够正常调用 _putc 函数，否则就会报错。至于 printf 函数的实现可以参考 sprintf 函数的实现，这里不再介绍。

之后时钟的实现，需要完善寄存器_DEVREG_TIMER_UPTIME 的实现。这个只是需要从 RTC_ADDR 中读出两次内容即可。

键盘的实现，需要完善寄存器_DEVREG_INPUT_KBD，这里也是只需要读出 KBD_ADDR 中的内容，再对通码断码进行一些操作即可。

VGA 的实现，这个时候就需要补全 string.c 这个库函数文件中的 memcpy() 函数。其中在讲义中明确介绍了两个寄存器，这两个寄存器的实现依然可以参考 native 中对应寄存器的实现。最后还需要补充 io_handler() 函数即可，也就是 TODO 的位置添加相应内容。

在实现结束上面四个设备之后，可以运行打字游戏和幻灯片播放。测试结果见第三节。

2.2.4 PA3-穿越时空的旅程：批处理系统

该阶段开始之前也需要进行代码的整理，参考 PA1 中的内容。

该阶段依旧分为三个小任务：实现自陷操作_yield()及其过程；实现用户程序的加载和系统调用；运行仙剑奇侠传并展示批处理系统。下面分别介绍：

1) 实现自陷操作_yield()及其过程

根据讲义的介绍，自陷操作的实现一共需要经历以下五个阶段：设置异常入口地址、触发自陷操作、保存上下文、事件分发、恢复上下文。

其中设置异常入口地址没有需要我们实现的地方，但是在这里注意观察可以发现异常程序入口地址为 0x80100324。

接着在触发自陷操作阶段，这里需要实现一个新函数：raise_intr()函数，在这里实现异常响应机制，这里只是进行一些简单的赋值操作和跳转到指定地址即可，在这里用到的 stvec 这些寄存器可以自己定义。之后还需要实现本阶段需要的新指令（ecall,csrrw,csrrs,sret 等）。

在保存上下文阶段，我们需要完成的就是重新组织_Context 的成员。通过观察 trap.s 文件中的内容，我们可以发现，它将 cause/status/epc 依次存放进了 t0/t1/t2。这里需要修改_Context 中成员的次序，同时将宏定义中的 GPR2/3/4 补充完整即可。

在事件分发阶段，我们需要在__am_irq_handle()中通过异常号来识别出自陷事件，并将其打包为编号为_EVENT_YIELD 的自陷事件；同时在 do_event()中识

别出 `_EVVENT_YIELD`，并输出提示信息。

最后在恢复上下文阶段，我们需要实现新的指令 `sret`。在这里注意根据异常类型的不同，有时候 `sret` 返回的地址需要加 4，有时候不需要。因此我们需要对这种情况进行特殊处理。针对自陷这种异常，我们可以在 `__am_irq_handle()` 中识别出异常的时候，将 `epc` 的值加 4 即可。

2) 实现用户程序的加载和系统调用

a、用户程序的加载

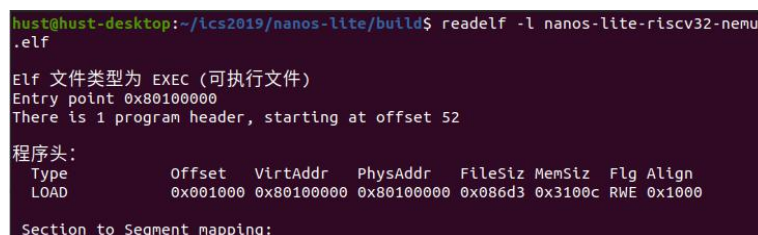
实现用户程序的加载，这一环节是需要将 `dummy` 程序放在正确的内存位置。根据讲义的内容，要加载一个用户程序，我们需要解决以下四个问题：

可执行文件在哪里？代码和数据在可执行文件的哪个位置？代码和数据有多少？"正确的内存位置"在哪里？

在这个实验中，加载用户程序实际上将用户程序先编译成一个 `ramdisk.img` 文件。而在这里，可执行文件就在 `ramdisk` 的偏移位置为 0 处。那么接下来的三个问题就和 ELF 文件格式有关。编译结束之后，我们进入 `nanos-lite/build` 文件夹下可以发现出现了一个名为 `nanos-lite-riscv32-nemu.elf` 的文件，这个文件也就是编译后的 `dummy` 的可执行文件。我们通过以下命令可以查看可执行文件的内容：

```
readelf -l nanos-lite-riscv32-nemu.elf
```

之后查看 ELF header 的信息如下：



```
hust@hust-desktop:~/ics2019/nanos-lite/build$ readelf -l nanos-lite-riscv32-nemu.elf
Elf 文件类型为 EXEC (可执行文件)
Entry point 0x80100000
There is 1 program header, starting at offset 52

程序头:
  Type           Offset  VirtAddr  PhysAddr  FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x80100000 0x80100000 0x086d3 0x3100c RWE 0x1000

Section to Segment mapping:
```

图 2-1 ELF 文件结构

我们可以看到这个 ELF 文件中只包含了一个程序头，并且有 `offset/VirtAddr/PhysAddr/FileSiz/MemSiz` 等数据，实际上，这个程序头就是我们要找的代码和数据，由于 `Type` 为 `LOAD` 的程序头只有一个，因此代码和数据也只有一个。我们需要将这些内容放到正确的内存位置。根据讲义上的内容，正确的内存位置就是虚拟内存地址。也就是说，这个 `segment` 使用的内存就是 `[VirtAddr, VirtAddr + MemSiz)` 这一连续区间，然后将 `segment` 的内容从 ELF 文件中读入到这一内存区

间, 并将[VirtAddr + FileSiz, VirtAddr + MemSiz)对应的物理区间清零。

之后理解了这四个问题就可以开始 loader()函数的编写。其中首要的问题就是我们如何在程序中获得上图所示的程序头的内容呢? 幸好, 给我们提供了 Elf_Ehdr 和 Elf_Phdr 这两个宏以及 ramdisk_read()这些从 ramdisk 中读取数据的函数。其中上述两个宏的具体定义可以参考代码, 这里不再详细介绍。

loader()实现的伪代码如下:

```
读入 ELF header
程序入口地址 = e_entry
定位程序头表, 根据 Elf_Ehdr 的 e_entry + e_phoff 找到程序头表
for 程序头表的每一项 (e_phnum)
    if Type == PT_Load then
        执行装载到内存, 如果 MemSiz>FileSiz, 还要将大于的部分清零
    end if
end for
return 程序入口地址
```

读入 ELF header 的方式如下:

```
Elf_Ehdr elf; ramdisk_read(&elf, 0, sizeof(Elf_Ehdr));
```

读入程序头的方式也和上面类似, 这里不再进行介绍。之后运行程序即可。

b、系统调用

在上一小节实现自陷操作的过程中, 我们曾遇到过 __am_irq_handle() 和 do_event() 这两个函数, 这两个函数主要是进行事件分发的的工作。我们曾处理了 yield 这样的情况, 这里需要添加一个 syscall 的情况, 具体实现过程可以参考讲义的内容。

之后 do_syscall() 函数的实现也是针对不同的系统调用类型进行处理。在这里我们只需要考虑系统调用号为 SYS_yield 和 SYS_exit 这两个的具体实现, 其中 SYS_yield 的实现只是需要调用 _yield() 同时将 GPRX 赋为 0 即可; SYS_exit 的实现只需要调用 halt() 函数即可。到这里系统调用的部分大致实现成功。

之后在处理标准输出的时候还需要我们完善 SYS_write 的系统调用, 这里我们只考虑 fd 等于 1 或者 2 这两种情况, 遇到这两种情况, 直接通过 _putc() 将 buf 为首地址的 len 字节输出到串口。同时在 _write() 中调用系统调用函数, 最后不要忘记在这里处理它的返回值。

最后还需要完善堆区管理, 即系统调用号位 SYS_brk。在这里我们对于

sys_brk()函数暂时直接返回 0 即可。主要要处理的地方是_sbrk()函数。我们首先需要通过 man 3 end 查看_end 的调用方式。之后根据讲义上的工作方式进行代码编写即可，这部分比较简单，这里不再展示。

到了这里 PA3.2 也就完成了。

3) 运行仙剑奇侠传并展示批处理系统

到了最后一步，要实现文件系统和批处理系统。

a、简易文件系统的实现

我们在加载第一个用户程序的时候曾写过一个 loader()函数，这里文件系统主要就是对 loader()函数进行重写。

由于我们在加载第一个用户程序的时候只有一个文件，因此不需要 filename 这个参数，但是当我们实现文件系统的时候会遇到大量的文件，因此这个时候就需要使用 filename 这个参数了。

根据讲义的要求，fs_open()函数的功能就是根据输入的参数 filename，查找 file_table 中有没有对应的文件，如果有，返回相应的序号，否则报错。

我们正在加载第一个用户程序的时候，直接使用了 ramdisk_read()和 ramdisk_write()这两个函数进行文件的读写操作。而在文件系统中我们将这两个函数分别封装进 fs_read()和 fs_write()这两个函数中来使用。由于文件的大小是固定的，在实现 fs_read(), fs_write()和 fs_lseek()的时候，注意偏移量不要越过文件的边界，也就是注意 offset、len、和 size 之间的大小关系。最后不要忘记更新文件的 offset。

fs_lseek()函数主要是来调整文件的偏移量，也就是 offset，根据参数是 SEEK_SET/SEEK_CUR/SEEK_END 来对 offset 做相应的修改。

最后的 fs_close()在这里直接返回 0 即可。但是由于我们之后还需要进行批处理系统的实现，这里还应该使得相应文件的 offset 置为 0，否则运行批处理系统的时候会直接输出“HIT BAD TRAP”，而不会输出一个菜单。

上面几个函数实现完之后，将 loader()函数进行调整，同时完善系统调用。

b、虚拟文件文件系统的实现

为了支持一些特殊文件的实现，我们还需要对上面编写的几个函数进行扩展。在这一步，需要用到 Finfo 结构体中的两个读写函数指针 read 和 write。扩

展也就主要是针对 `fs_read()` 和 `fs_write()` 这两个函数。

当 `read` 和 `write` 指向的是 `NULL` 的时候，只需要将文件作为一个普通文件通过 `ramdisk_read` 和 `ramdisk_write()` 进行读写即可。否则需要调用他们指向的具体函数。比如在上一个阶段，我们是直接将 `stdout` 和 `stderr` 通过 `_putc()` 函数输出，这里我们需要实现 `nanos-lite/src/device.c` 文件下的 `serial_write()` 函数。在这个函数里面调用 `_putc()` 函数即可。最后不要忘记在 `file_table` 中将相应的读文件指针指向 `serial_write()`。

之后我们还需要在 `file_table` 中添加一个 `/dev/events` 文件，该文件不可写，读指针指向 `nanos-lite/src/device.c` 中的 `events_read()`，这个函数主要是读取键盘信息和时间信息等。这里的具体实现可以参考 `nexus-am/apps/typing/keyboard.c` 中 `keyboard_event()` 的实现。实现成功之后敲击键盘就会出现相应的相应信息，等待过程会输出时间信息。

最后我们还需要将 `VGA` 显存抽象成文件，这里的实现步骤可以参考讲义上的六条步骤。其中比较麻烦的就是在 `fb_write()` 中通过 `offset` 来计算坐标。在 `navy-apps/libs/libndl` 目录下有两个文件，其中 `bmpptest` 中调用的函数也都来自这两个文件，其中有个函数为 `NDL_Render()`，在这里他调用了 `fseek` 来设置 `offset`，并且将 `offset` 的值设置为 $(y * \text{width} + x) * \text{sizeof}(\text{uint32_t})$ 。我们可以通过这个式子的逆运算推导出用 `offset` 计算坐标。实现完成之后，运行文件，此时会显示一张照片。

c、批处理系统

在实现上面的内容之后，下载仙剑奇侠传的数据文件并放置到指定位置，这个时候已经可以初步运行该游戏了，但是在尝试的过程中会出现报错，观察报错信息，发现有一条未实现的指令 `slti`。

之后的批处理系统其实就是新添加了一个系统调用 `SYS_execve()`，我们只需要在这里调用 `naive_uoload()` 即可。另外，需要修改 `SYS_exit`，原先我们是直接调用了 `_halt()`，但是在这里我们需要调用 `naive_uoload(NULL, "/bin/init")`。

最后测试运行结果。

2.2.5 PA4-虚实交错的魔法：分时多任务

这里也是分为了三个子任务：实现基本的多道程序系统、实现支持虚存管理

的多道程序系统、实现抢占式分时多任务系统，并提交完整的实验报告。

1) 实现基本的多道程序系统

基本的多道程序系统实现比较简单，根据讲义的要求，我们一共需要实现两个部分，一个是上下文切换的实现，另一个是创建用户进程上下文的实现。

a、上下文切换的实现

在这里我们一共需要实现 CTE 中的 `_kcontext()` 函数、Nanos-lite 中的 `schedule()` 函数、以及修改 `do_events()` 中对自陷事件的处理、最后修改 `__am_asm_trap`，使得程序从 `__am_irq_handle()` 退出后，跳转到新进程的上下文结构，最后才恢复上下文。

其中 `_kcontext()` 函数，我们只需要新声明一个 `_Context` 类型的变量，使其恰好位于 `stack` 的尾端，并且将 `entry` 保存进结构体的 `epc` 变量中，最后返回结构体的指针即可。

`schedule()` 函数的实现已经在讲义中介绍。

`do_events()` 函数中，我们原先在处理自陷事件的时候是直接输出了一句话，在这里我们为了实现上下文的切换，还需要在处理自陷事件的时候直接返回 `schedule()` 函数。

`__am_asm_trap` 的修改，程序在跳转到 `__am_irq_handle()` 中的时候，如果遇到了自陷事件，它就会调用 `schedule` 函数，然后获得一个 `_Context` 类型的指针，并且将这个指针作为一个返回值保存到寄存器 `a0` 中，因此我们需要做的就是将当前栈顶指针指向 `a0` 寄存器中的内容，用反汇编实现就是 “`sw sp,a0`”，添加在 `jal __am_irq_handle()` 的后面。

最后不要忘了修改 `init_proc()` 的内容。

b、创建用户进程上下文的实现

在实现了上一小节的内容之后，再观察本小节需要实现的内容，一共有三个文件需要改动，其中 `init_proc()` 和 `schedule()` 中需要添加的内容已经给出。我们只需要实现 `_ucontext()` 即可，这个函数的实现和 `_kcontext()` 的实现大致相同，这里不再介绍。

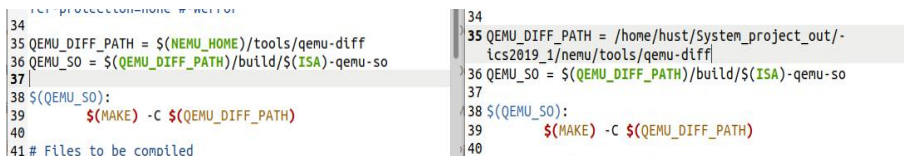
最后运行程序。

3 实验结果与结果分析

3.1 PA0-世界诞生的前夜：开发环境的配置

该关卡主要是开发环境的配置，同时也编写了一个最简单的 HelloWorld 的程序，并将其通过一个 Makefile 文件的形式进行编译，并运行。最后尝试使用 gdb 进行调试。

在这一关卡中我认为有两个地方是需要注意的：其一，通过 git 直接克隆的项目的地址。在刚开始实验的时候，是新建了一个名为 System_project_out 的文件夹来作为项目地址，但是这个时候在通过 make 初始化项目的时候就会出现报错，根据提示信息发现原来是文件的地址出现了错误，打开 nemu 文件夹下的 Makefile 文件，将其中的路径修改如下：



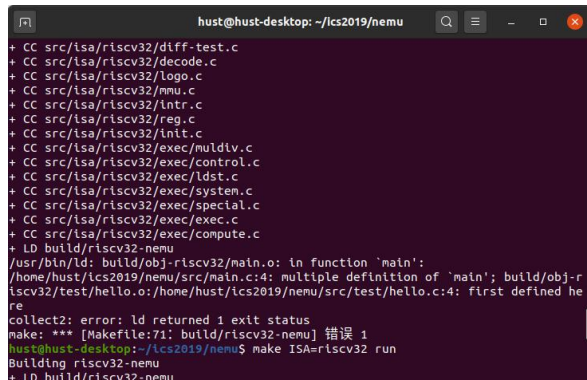
```
34 QEMU_DIFF_PATH = $(NEMU_HOME)/tools/qemu-diff
35 QEMU_SO = $(QEMU_DIFF_PATH)/build/$(ISA)-qemu-so
36
37 $(QEMU_SO):
38     $(MAKE) -C $(QEMU_DIFF_PATH)
39
40 # Files to be compiled
```

```
34 QEMU_DIFF_PATH = /home/hust/System_project_out/-
35   ics2019_1/nemu/tools/qemu-diff
36 QEMU_SO = $(QEMU_DIFF_PATH)/build/$(ISA)-qemu-so
37
38 $(QEMU_SO):
39     $(MAKE) -C $(QEMU_DIFF_PATH)
40
```

图 3-1 Makefile 文件修改前后对比

之后，再次执行就不会报错。但是在实际实验的过程中，又重新拉取了项目文件直接到 hust 文件夹下，避免了修改 Makefile 文件。

其二：在新建一个 HelloWorld 文件的时候是直接在 nemu/src/test 下面创建了一个 main 函数，但是这样的操作在 PA0 阶段的时候不会出错，但是在 PA1 阶段的时候就会出现如下所示的报错信息：



```
hust@hust-desktop: ~/ics2019/nemu
+ CC src/isa/riscv32/diff-test.c
+ CC src/isa/riscv32/decode.c
+ CC src/isa/riscv32/logo.c
+ CC src/isa/riscv32/mmu.c
+ CC src/isa/riscv32/intr.c
+ CC src/isa/riscv32/reg.c
+ CC src/isa/riscv32/init.c
+ CC src/isa/riscv32/exec/muldiv.c
+ CC src/isa/riscv32/exec/ctrl.c
+ CC src/isa/riscv32/exec/ldst.c
+ 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
/usr/bin/ld: build/obj-riscv32/main.o: in function 'main':
/home/hust/ics2019/nemu/src/main.c:4: multiple definition of 'main'; build/obj-riscv32/test/hello.o:/home/hust/ics2019/nemu/src/test/hello.c:4: first defined here
collect2: error: ld returned 1 exit status
make: *** [Makefile:71: build/riscv32-nemu] 错误 1
hust@hust-desktop: ~/ics2019/nemu$ make ISA=riscv32 run
Building riscv32-nemu
+ LD build/riscv32-nemu
```

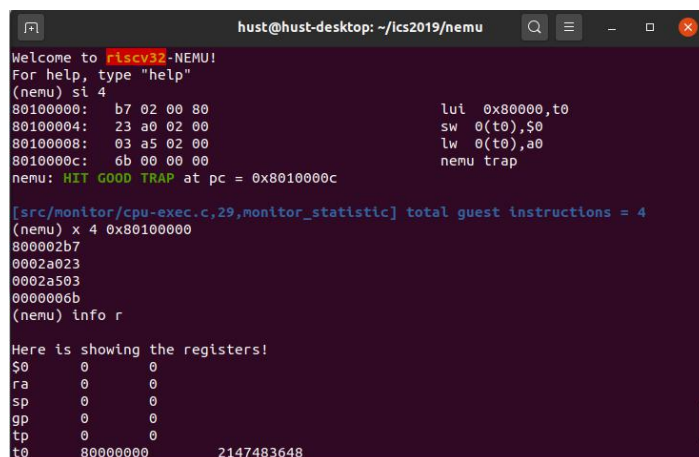
图 3-2 PA0 阶段新建一个 HelloWorld 文件后的隐患

当然，解决方法也很简单，直接删除该文件即可。

3.2 PA1-开天辟地的篇章：最简单的计算机

1) 实现一个简单的调试器

这里实现了单步执行功能（si [N]），打印当前寄存器状态（info r），扫描内存（x N EXPR）三个功能。展示如下：



```
hust@hust-desktop: ~/lcs2019/nemu
Welcome to riscv32-NEMU!
For help, type "help"
(nemu) si 4
00100000: b7 02 00 80          lui 0x800000,t0
00100004: 23 a0 02 00          sw 0(t0),S0
00100008: 03 a5 02 00          lw 0(t0),a0
0010000c: 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 4 0x80100000
000002b7
0002a023
0002a503
0000006b
(nemu) info r

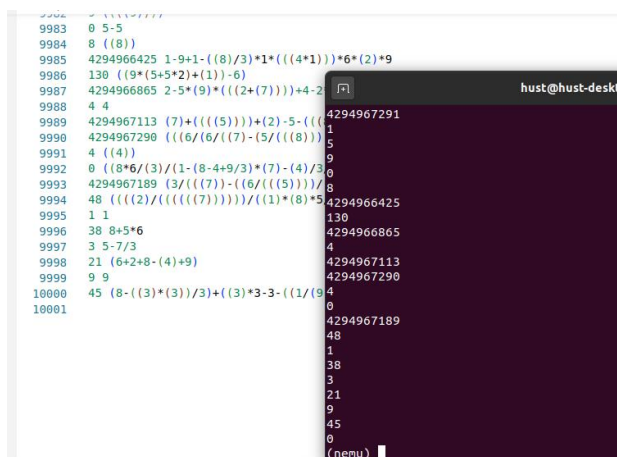
Here is showing the registers!
$0      0      0
ra      0      0
sp      0      0
gp      0      0
tp      0      0
t0      80000000 2147483648
```

图 3-3 简易调试器实现结果

其中单步执行功能的命令为 si 4，表示单步执行 4 步；扫描内存的命令为 x 4 0x80100000，表示从 0x80100000 扫描 4*4 个字节的内存，由于这一步还没有实现表达式的求值，因此使用了一个十六进制的数代替；打印当前寄存器状态的命令为 info r，直接显示出所有寄存器当前的值。

2) 实现表达式求值

表达式的求值和自动生成表达式的功能实现后开始进行验证，最后的验证结果如下所示：



```
9983 0 5-5
9984 8 ((8))
9985 4294966425 1-9+1-(((8)/3)*1*(((4*1))) *6*(2)*9
9986 130 ((9*(5+5*2)+(1))-6)
9987 4294966865 2-5*(9)*(((2+(7))))+4-2
9988 4 4
9989 4294967113 (7)+(((5))) + (2)-5-(((1
9990 4294967290 (((6/(6/((7)-(5/((8))))
9991 4 ((4))
9992 0 ((8*6/(3)/(1-(8-4+9/3)*(7)-(4)/2
9993 4294967189 (3/(((7))-((6/((5))))/
9994 48 (((2)/(((7)))))/((1)*(8)*2
9995 1 1
9996 38 8+5*6
9997 3 5-7/3
9998 21 (6+2+8-(4)+9)
9999 9 9
10000 45 (8-((3)*(3))/3)+((3)*3-3-((1/5)
10001
```

图 3-4 表达式求值的验证

其中图片左侧是自动生成表达式的部分内容，可以看到，恰好生成了 10000 条表达式，右侧则是执行结果，这里只截取了后面的一部分，可以看到输出的结果和自动生成表达式的计算结果相同，最后一行输出的 0 表示自己计算的结果和自动生成表达式计算的结果不相同的结果个数，这里表示没有不相同的，也就表示表达式求值的功能基本正确。

3) 监视点的实现与观察

这里实现的时候将监视池的结构体中添加了一个字符指针，但是在实际运行的时候会报错，因此，这里将这里修改成了一个大小为 32 的字符数组。

之后在测试能否计算寄存器的时候，发现能正确输出结果，但是之后的简单表达式的计算会出错，如下图所示：

```

hust@hust-desktop: ~/ics2019/nemu
114 0 0 0 $t0
0 0 0 0
0 0 0 0
0
(nemu) si 1
80100000: b7 02 00 80          lui  0x80000,t0
(nemu) p $t0
ui:$t0
expr:$t0
114 -2147483648 0 0 $t0
0 0 0 0
0 0 0 0
2147483648
(nemu) p 10+5
ui:10+5
expr:10+5
100 100 0 0 100
43 0 2 1 +
100 5 0 0 5
105
(nemu) p 1+2
ui:1+2
expr:1+2
100 100 0 0 100

```

图 3- 5 表达式求值完善过程中的错误

这时候发现是 str 数组的问题，初始化的时候出错，因此将初始化的语句修改为 `memset(token[i].str, '0');` 即可，最后表达式求值完善成功。

最后的监视点结果如下：

```

Welcome to riscv32-NEMU!
For help, type "help"
(nemu) w $t0
(nemu) si 4
80100000: b7 02 00 80          lui  0x80000,t0
the program is stopped case of a watchpoint!
(nemu) q
hust@hust-desktop: ~/ics2019/nemu$

```

图 3- 6 监视点的实现结果

4) 必答题

问题一：我选择的 ISA 是 riscv32;

问题二：调试次数为 $500 \times 0.9 = 450$ 次，如果没有实现简易调试器，那么需要 $450 \times 20 \times 30s = 270000s = 75h$ ；如果实现了简易调试器，那么只需要 $75/3 = 25h$ 。

问题三：查阅手册。

Riscv 有哪几种指令格式：在参考文献第 23 页，一共有六种基本指令格式，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。

LUI 指令的行为是什么？在参考文献 1 的第 151 页，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。

mstatus 寄存器的结构是怎么样的？在参考文献【1】的第 101 页。

问题四：shell 命令：通过”find -name “*[.h|.c]” -type f | xargs cat |wc -l”命令查看当前文件夹下所有.c 和.h 文件的行数，一共为 5804 行。这个时候可以直接切换到 pa0 分支，再次运行命令，输出 5009 行，因此自己一共添加了 795 行。

如果想筛选掉空行，只需要 ”find -name “*[.h|.c]” -type f | xargs cat|grep -v ^\$|wc -l”即可。这时候一共有 4824 行代码。

添加到 Makefile 文件中，在 run:后面的位置添加如下内容，即可通过 make count 统计代码行数。

```
count:
    find -name “*[.h|.c]” -type f | xargs cat |wc -l
```

问题五：-Wall，打开 gcc 的所有警告,-Werror，他要求 gcc 将所有的警告当成错误进行处理。使用这两个保证了程序的正确运行。

3.3 PA2-简单复杂的机器：冯诺依曼计算机系统

1) 运行第一个 C 程序

```
hust@hust-desktop: ~/ics2019/nexus-am/tests/cputest
hust@hust-desktop:~/ics2019/nexus-am/tests/cputest$ make ARCH=riscv32-nemu ALL=d
dummy run
Makefile:17: 警告: 覆盖关于目标“lnage”的配方
/home/hust/ics2019/nexus-am/arch/platform/nemu.mk:20: 警告: 忽略关于目标“lnage”
的旧配方
Makefile:18: 警告: 覆盖关于目标“run”的配方
/home/hust/ics2019/nexus-am/arch/platform/nemu.mk:27: 警告: 忽略关于目标“run”
的旧配方
# Building dummy [riscv32-nemu] with AM_HOME (/home/hust/ics2019/nexus-am)
# Building lib-am [riscv32-nemu]
# Building lib-klib [riscv32-nemu]
# Creating binary lnage [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/tes
ts/cputest/build/dummy-riscv32-nemu.bin
[src/memory/memory.c,16,register_pmen] 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 gener
ated to record every instruction NEMU executes. This may lead to a large log fil
e. If it is not necessary, you can turn it off in include/common.h.
[src/monitor/monitor.c,28,welcome] Build time: 17:12:00, Sep 8 2022
Welcome to riscv32-NEMU!
For help, type "help"
nemu: HIT GOOD TRAP at pc = 0x80100030

[src/monitor/cpu-exec.c,31,monitor_statistic] total guest instructions = 13
dummy
hust@hust-desktop:~/ics2019/nexus-am/tests/cputest$
```

图 3- 7 dummy 程序测试结果

如图所示，命令行输出“HIT GOOD TRAP”，表示该程序中的指令都已经正确实现。

2) 丰富指令集，测试所有的程序

```
hust@hust-desktop: ~/ics2019/nemu
NEMU compile OK
compiling testcases...
testcases compile OK
add-longlong PASS!
add PASS!
bblt 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$
```

图 3- 8 所有测试程序测试结果

在完善了所有程序的指令和库函数之后，再次进行测试，最后的结果如上所示，表示所有的指令都已经正确实现。

3) 实现 I/O 指令，测试打字游戏

在这里测试串口的时候，需要完善 printf()函数，其中需要读入%c 类型的数据，这个时候按照习惯应该通过 va_arg(ap,char)获取参数，但是实际上应该通过 va_arg(ap,int)来获取参数内容。

之后测试时钟，查看跑分结果如下，这里只展示最后一个程序。


```
hust@hust-desktop: ~/lcs2019/nexus-am/apps/microbench
x8600063]
[src/monitor/monitor.c,25,welcome] Debug: OFF
[src/monitor/monitor.c,28,welcome] Build time: 21:34:57, Sep 22 2022
Welcome to riscv32-NEMU!
For help, type "help"
===== Running MicroBench [input *ref*] =====
[qsrt] Quick sort: * Passed.
  min time: 740 ms [691]
[queen] Queen placement: * Passed.
  min time: 908 ms [518]
[bf] Brainf**k interpreter: * Passed.
  min time: 5221 ms [453]
[fib] Fibonacci number: * Passed.
  min time: 12031 ms [235]
[sieve] Eratosthenes sieve: * Passed.
  min time: 9939 ms [396]
[15pz] A* 15-puzzle search: * Passed.
  min time: 1638 ms [273]
[dinic] Dinic's maxflow algorithm: * Passed.
  min time: 2050 ms [530]
[lzip] Lzip compression: * Passed.
  min time: 1922 ms [395]
[qsrt] Suffix sort: * Passed.
  min time: 721 ms [624]
[md5] MD5 digest: * Passed.
  min time: 9758 ms [176]
=====
MicroBench PASS      429 Marks
                        vs. 100000 Marks (1.7780K @ 4.20GHz)
Total time: 51539 ms
nemu: HIT GOOD TRAP at pc = 0x801041e0
[src/monitor/cpu-exec.c,31,monitor_statistic] total guest instructions = 1865085
432
make[1]: 离开目录“/home/hust/lcs2019/nemu”
```

图 3- 9 microbench 跑分测试

测试键盘:

```
[src/monitor/monitor.c,28,welcome] Build time: 21:34:57, Sep 22 2022
Welcome to riscv32-NEMU!
For help, type "help"
Try to press any key...
Get key: 43 A down
Get key: 43 A up
Get key: 31 E down
Get key: 31 E up
Get key: 42 CAPSLOCK down
Get key: 42 CAPSLOCK up
Get key: 58 C down
Get key: 58 C up
Get key: 67 LCTRL down
Get key: 67 LCTRL up
^Z
[1]+ 已停止                  make run ARCH=riscv32-nemu mainargs=k
```

图 3- 10 键盘测试

VGA 的动画效果测试:

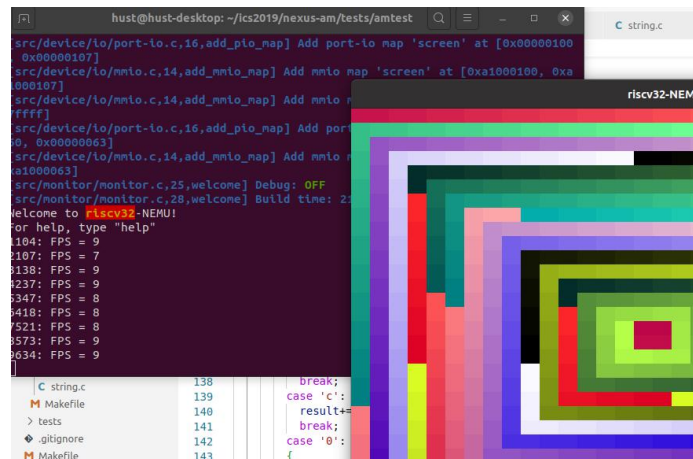


图 3- 11 VGA 动画效果测试

幻灯片的播放截图如下所示:

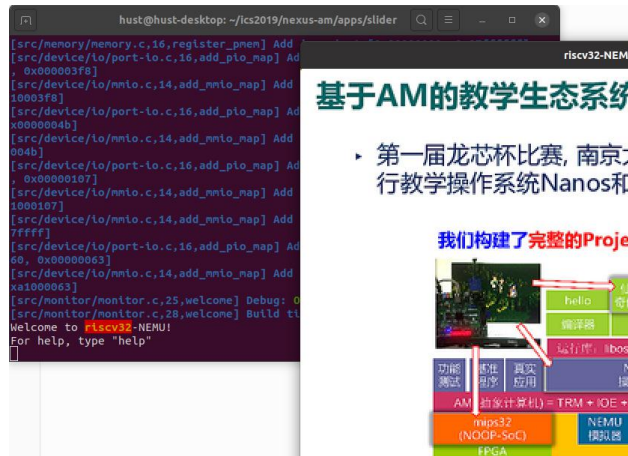


图 3- 12 幻灯片播放展示截图

打字游戏的测试截图：

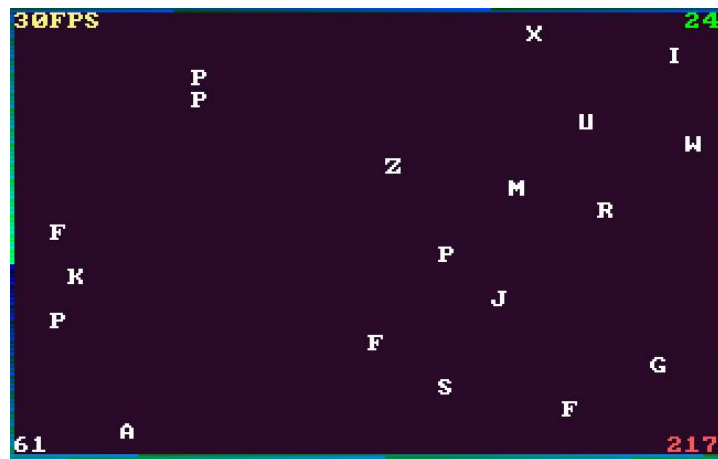


图 3- 13 打字游戏测试截图

4) 必答题

第一题：指令在 `nemu` 中的执行过程。其中一条指令的执行大致需要经过取指令、解码、执行、更新 PC 等四个阶段。具体来说一条指令的执行首先需要调用 `nemu/src/isa/riscv32/exec/exec.c` 文件中的 `isa_exec()` 函数，在该函数中，通过调用 `instr_fetch()` 函数来获取相应存储位置的指令内容，之后通过调用 `nemu/include/cpu/exec.h` 文件中的 `idex()` 函数来进行指令的解码、执行。如果该指令对应的有相应的解码函数，那么会调用 `nemu/include/cpu/decode.h` 文件中的 `make_DHelper()` 函数进行指令的解码，读出指令中的寄存器内容、立即数内容等等。解码结束之后，开始进行指令的执行，这里如果同一个 Opcode 对应多条指令，还需要调用 `nemu/src/isa/riscv32/exec/exec.c` 文件中的 `make_EHelper()` (一类指令) 函数，将每一条指令具体的指向一个执行函数 `make_EHelper(具体指令名)`，该函数的实现位于和 `exec.c` 同文件夹下的其他文件。在执行结束之后，依据实际

情况进行 PC 的更新，即调用 `exec.h` 文件中的 `update_pc()` 函数。至此一条指令执行结束。

第二题：在这里删除的是 `rtl_ad` 类中的 `inline` 修饰符，但是删除之后程序依然可以正确执行；如果删除 `static` 修饰符，依然可以正常运行；如果二者同时删除，这个时候会编译错误。这里编译错误的原因：因为同时存在 `interpret_rtl_add` 和 `rtl_add` 等重复实现的函数，这里去掉了 `static inline`，导致这些函数会报重复定义的错误，因此编译失败（`static` 本身的作用之一：其他文件中可以定义相同名字的函数，不会发生冲突。）。而如果只是单独删去某一个修饰符，有可能依然可以保证其他文件中定义同名函数，因此可能不会编译失败。

第三题：第一问，在 `common.h` 文件中添加 “`volatile static int dummy`” 后，再次编译文件中一共有 34 个 `dummy` 变量的实体。在 `ics2019/nemu/build/obj-riscv32` 文件夹下通过以下命令查找出编译后的 NEMU 中的 `dummy` 变量的实体个数：

```
hust@hust-desktop:~/ics2019/nemu/build/obj-riscv32$ grep -r -c 'dummy' ./* | grep '\.o:[1-9]' | wc -l
34
```

图 3-14 查找 `dummy` 变量的个数

第二问，在 `debug.h` 文件中添加了一句 “`volatile static int dummy`” 后，再次编译 NEMU，此时依然有 34 个 `dummy` 变量的实体。`dummy` 变量的实体数目未变化，`debug.h` 文件中已经包含了 `common.h`，对于变量的重复声明仅作用一次，因此变量的实体数目未发生变化。

第三问，如果为这两处 `dummy` 进行初始化，再次编译就会报错。出现了重定义错误，因为 `debug.h` 包含了 `common.h`，导致对 `dummy` 进行了两次初始化。

```
hust@hust-desktop:~/ics2019/nemu$ make ISA=riscv32
Building riscv32-nemu
+ CC src/monitor/diff-test/diff-test.c
In file included from ./include/nemu.h:4,
                 from src/monitor/diff-test/diff-test.c:3:
./include/common.h:35:21: error: redefinition of 'dummy'
   35 | volatile static int dummy=0;
      | ~~~~~
In file included from ./include/common.h:32,
                 from ./include/nemu.h:4,
                 from src/monitor/diff-test/diff-test.c:3:
./include/debug.h:8:21: note: previous definition of 'dummy' was here
    8 | volatile static int dummy=0;
      | ~~~~~
make: *** [Makefile:50: build/obj-riscv32/monitor/diff-test/diff-test.o] 错误 1
```

图 3-15 初始化 `dummy` 后再次编译 NEMU 的结果

第四题：`make` 是一个命令工具，是用来解释 Makefile 中指令的工具。当我们执行 `make` 命令的时候，`make` 会首先在当前的目录下查找 Makefile 文件，执

行对应的操作。其中 `make` 会自动判断源代码是否发生了变化，而自动更新执行档。

编译链接的过程：首先指定要编译的源文件，对着源文件进行预处理，然后对于每个 `.c` 和 `.h` 文件进行编译、汇编，最后将上一步编译产生的 `obj` 文件进行链接，生成最终的可执行文件。

3.4 PA3-穿越时空的旅程：批处理系统

1) 实现自陷操作 `_yield()` 及其过程

这里在 `do_event()` 函数中，如果遇到了自陷事件，那么输出 “`yield ok`” 的字样，并且 `dummy.c` 的程序会正确触发中断，显示如下：

```

hust@hust-desktop: ~/ics2019/nanos-lite
For help, type "help"
[/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: 21:31:14, Oct 17 2022
[/home/hust/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = p, end = p, size = -2146421188 bytes
[/home/hust/ics2019/nanos-lite/src/device.c,73,init_device] Initializing devices...
[/home/hust/ics2019/nanos-lite/src/irq.c,21,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
yield ok
[/home/hust/ics2019/nanos-lite/src/main.c,39,main] system panic: simao, Should not reach here
nemu: HIT BAD TRAP at pc = 0x80100ea4
[src/monitor/cpu-exec.c,31,monitor_statistic] total guest instructions = 491330
make[1]: 离开目录 "/home/hust/ics2019/nemu"
hust@hust-desktop: ~/ics2019/nanos-lite$

```

图 3-16 自陷操作的实现

2) 实现用户程序的加载和系统调用

在实现完 `loader()` 函数后，进行测试，最后运行结果如下所示：

```

[/home/hust/ics2019/nanos-lite/src/loader.c,41,naive_uoload] Jump to entry = -7cf fff38
[/home/hust/ics2019/nanos-lite/src/irq.c,13,do_event] system panic: Unhandled event ID = 1
nemu: HIT BAD TRAP at pc = 0x801007a0
[src/monitor/cpu-exec.c,31,monitor_statistic] total guest instructions = 595670
make[1]: 离开目录 "/home/hust/ics2019/nemu"
hust@hust-desktop: ~/ics2019/nanos-lite$ make ARCH=riscv32-nemu run

```

图 3-17 用户程序加载测试

最后通过上图看出，成功实现用户程序的加载之后依然会出现一个报错，但是观察发现报错信息是：出现一个 ID 为 1 的未处理事件。而这个事件就是我们在下一步系统调用的时候需要处理的内容。

接着我们开始测试系统调用的功能，这时候根据讲义上面的步骤，已经实现了 `SYS_yield` 系统调用，在实现正确的情况下，应该会出现一个新的报错内容，但是实际上此时的运行结果如下：

```

./
[/home/hust/ics2019/nanos-lite/src/loader.c,41,naive_uoload] Jump to entry = -7cf
fff38
yield ok
yield ok
yield ok
yield ok
yield ok

```

图 3- 18 SYS_yield 出错展示

发现此时会不停的输出“yield ok”这个内容，而这个内容只有在 do_event() 函数中出现，也就是说现在陷入了死循环，当程序执行完 yield() 函数之后，没有跳出该函数，反而重复执行。因此想到这可能是返回地址的问题。再考虑到自己在 PA3.1 实现的过程中，只针对自陷这一种过程的返回地址进行了加 4 的操作，因此在这里对于系统调用这个过程，也将他们的返回地址进行加 4 的操作，最后再次执行之后，发现出现了一个新的报错，如下图所示：

```

[/home/hust/ics2019/nanos-lite/src/loader.c,41,naive_uoload] Jump to entry = -7cf
fff38
yield ok
[/home/hust/ics2019/nanos-lite/src/syscall.c,18,do_syscall] system panic: Unhand
led syscall ID = 0
nemu: HIT BAD TRAP at pc = 0x801007a0

[src/monitor/cpu-exec.c,31,monitor_statistic] total guest instructions = 596230
make[1]: 离开目录"/home/hust/ics2019/nemu"
hust@hust-desktop:~/ics2019/nanos-lite$ make ARCH=riscv32-nemu run

```

图 3- 19 SYS_exit 未实现造成的报错

而这个报错则是因为 SYS_exit 没有实现造成的。在实现之后，再次运行程序，最后发现运行成功。展示如下：

```

./
[/home/hust/ics2019/nanos-lite/src/loader.c,41,naive_uoload] Jump to entry = -7cf
fff38
yield ok
nemu: HIT GOOD TRAP at pc = 0x801007ac

[src/monitor/cpu-exec.c,31,monitor_statistic] total guest instructions = 594886
make[1]: 离开目录"/home/hust/ics2019/nemu"
hust@hust-desktop:~/ics2019/nanos-lite$

```

图 3- 20 SYS_exit 系统调用成功实现后展示结果

查看 SYS_write 的系统调用的实现结果：

```

./
[/home/hust/ics2019/nanos-lite/src/loader.c,41,naive_uoload] Jump to entry = -7cf
fffe0
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!

```

图 3- 21 SYS_write 系统调用实现结果

最后查看 SYS_brk 系统调用的实现结果，为了方便观察实现前后结果对比，在 sys_write() 中添加一句 Log(“sys_write\n”)；可以观察到实现 SYS_brk 前，用户程序通过 printf() 输出的时候，确实是逐个字符地调用 write() 来输出的；而在实现 SYS_brk 后，用户程序通过 printf() 输出的时候，将格式化完毕的字符串通过一次 write() 系统调用进行输出，而不是逐个字符地进行输出。结果如下：


```

Hello World!
[/home/hust/ics2019/nanos-lite/src/syscall.c,27,sys_write] sys_write
H[/home/hust/ics2019/nanos-lite/src/syscall.c,27,sys_write] sys_write
e[/home/hust/ics2019/nanos-lite/src/syscall.c,27,sys_write] sys_write
l[/home/hust/ics2019/nanos-lite/src/syscall.c,27,sys_write] sys_write
l[/home/hust/ics2019/nanos-lite/src/syscall.c,27,sys_write] sys_write
o[/home/hust/ics2019/nanos-lite/src/syscall.c,27,sys_write] sys_write
[/home/hust/ics2019/nanos-lite/src/syscall.c,27,sys_write] sys_write
W[/home/hust/ics2019/nanos-lite/src/syscall.c,27,sys_write] sys_write
o[/home/hust/ics2019/nanos-lite/src/syscall.c,27,sys_write] sys_write

```

图 3-22 SYS_brk 实现前 printf 输出

```

Hello World!
[/home/hust/ics2019/nanos-lite/src/syscall.c,27,sys_write] sys_write
Hello World from Navy-apps for the 2th time!
[/home/hust/ics2019/nanos-lite/src/syscall.c,27,sys_write] sys_write
Hello World from Navy-apps for the 3th time!
[/home/hust/ics2019/nanos-lite/src/syscall.c,27,sys_write] sys_write
Hello World from Navy-apps for the 4th time!
[/home/hust/ics2019/nanos-lite/src/syscall.c,27,sys_write] sys_write
Hello World from Navy-apps for the 5th time!
[/home/hust/ics2019/nanos-lite/src/syscall.c,27,sys_write] sys_write

```

图 3-23 SYS_brk 实现后 printf 输出

3) 运行仙剑奇侠传并展示批处理系统

我们在实现了简易的文件系统之后，通过 `naive_upload()` 调用 “/bin/text”，最后出现如下所示的结果，表明程序实现正确。

```

[/home/hust/ics2019/nanos-lite/src/irq.c,21,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/loader.c,71,naive_upload] Jump to entry = -7cffffd0c
PASS!!!
nemu: HIT GOOD TRAP at pc = 0x80100c1c
[src/monitor/cpu-exec.c,31,monitor_statistic] total guest instructions = 1663530
make[1]: 离开目录“/home/hust/ics2019/nemu”
hust@hust-desktop: ~/ics2019/nanos-lite$

```

图 3-24 /bin/text 运行结果

之后来测试 “/bin/events”，测试结果如下所示，程序实现正确。

```

hust@hust-desktop: ~/ics2019/nanos-lite
...
[/home/hust/ics2019/nanos-lite/src/irq.c,21,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/loader.c,71,naive_upload] Jump to entry = -7cffffd0c
Start to receive events...
receive event: kd A
receive event: ku A
receive event: kd S
receive event: ku S
receive time event for the 1024th time: t 4249
receive time event for the 2048th time: t 5091
receive time event for the 3072th time: t 5914
receive time event for the 4096th time: t 6726
receive time event for the 5120th time: t 7548
receive time event for the 6144th time: t 8487
receive time event for the 7168th time: t 9342
receive event: kd X
receive event: ku X
receive time event for the 8192th time: t 10221
receive time event for the 9216th time: t 11253
receive time event for the 10240th time: t 12366

```

图 3-25 /bin/events 运行结果

最后测试/bin/bmptest，如下所示，正确显示出一张图片。

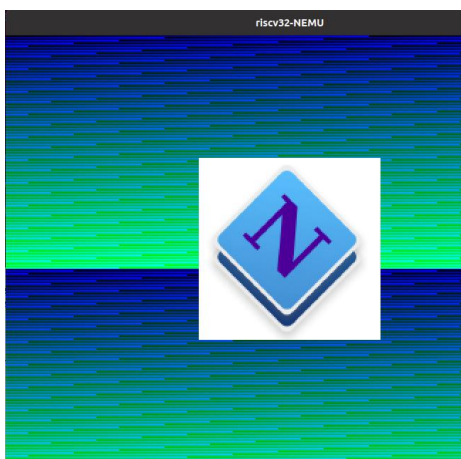


图 3- 26 /bin/bmptest 运行结果

测试运行仙剑奇侠传，在第一次测试运行的时候，发现运行速度特别慢，在关掉 DUBUG 之后，再次测试运行仙剑奇侠传，发现运行速度快了很多，但是仍然还会出现卡顿。在这里只展示一个封面的截图，如下所示：



图 3- 27 仙剑奇侠传运行截图

最后尝试批处理的运行，在刚开始实现的时候直接运行“/bin/init”，发现会报错，而不是出现一个菜单。一步步调试发现是自己的 `fs_close()` 函数出现错误。就像在上面实验方案设计阶段说的那样。如果直接返回 0，而没有将 `offset` 置为 0，就会出现这种错误。修改之后再次运行，结果如下所示，其中 0 和 1 不能运行，其他可以正常运行。

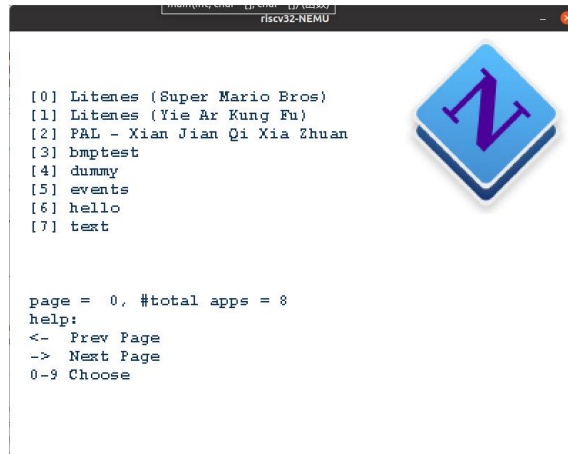


图 3-28 批处理运行结果

4) 必答题

问题一：理解上下文结构体的前世今生

这个 `c` 指向的上下文结构实际上是在触发 `raise_intr()` 后，将上下文保存到了 `_Context` 中。在触发 `raise_intr()` 后，会有以下动作：将当前 PC 值保存到 `sepc` 寄存器、在 `scause` 寄存器中设置异常号、从 `stvec` 寄存器中取出异常入口地址、跳转到异常入口地址。之后将上下文结构保存到了 `_Context` 中，这也就是上下文结构的来源。而每个成员具体是在哪里赋值的，赋值的部分在 `trap.s` 中，如下所示的一段指令。

```

dmi_dmi_trap:
addi sp, sp, -CONTEXT_SIZE

MAP(REGS, PUSH)

mv t0, sp
addi t0, t0, CONTEXT_SIZE
sw t0, OFFSET_SP(sp)

csrr t0, scause
csrr t1, sstatus
csrr t2, sepc

sw t0, OFFSET_CAUSE(sp)
sw t1, OFFSET_STATUS(sp)
sw t2, OFFSET_EPC(sp)

```

图 3-29 上下文结构赋值的代码段

`riscv32-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？刚才在 NEMU 中实现的新指令主要是为了完成自陷过程，然后调用 `raise_intr()` 函数，将上下文结构保存到 `_Context` 中，其中在 `riscv32-nemu.h` 中定义了 `_Context` 结构体，并且根据 `trap.s` 文件中的代码片段修改了结构体成员的顺序。同时通过 `trap.s` 将上下文结构保存到结构体中。

问题二：理解穿越时空的旅程

`_yield()`调用的具体过程：在我们调用`_yield()`之后，代码就会执行反汇编代码“`li a7,-1`”和“`ecall`”指令；执行`ecall`指令的时候，除了基础的取指令、译码，接着会跳转到`make_EHelper(system)`，然后识别出`ecall`指令后会跳转到`raise_intr()`函数；在这个函数中，会将当前PC值保存到`sepc`寄存器、在`scause`寄存器中设置异常号、从`stvec`寄存器中取出异常入口地址、跳转到异常入口地址等四个操作；之后就会进入到`__am_asm_trap`中，将上下文结构保存到`_Context`中；接着会执行`__am_irq_handle()`，在这里会根据`cause`识别出中断原因，并设置相应的事件类型；最后就会跳转到`do_event()`函数，在这里对相应的事件类型进行处理；最后恢复上下文（通过`sret`指令），自陷过程结束；然后执行下一条指令。

问题三：hello 程序是什么，它从而何来，要到哪里去

`hello.c`本身是一个C语言程序，在经过编译后，形成了一个elf文件，这时候文件的数据、代码等内容都保存到了`ramdisk`中，这个镜像文件从偏移地址为0的地方开始就是`hello`的内容，只是在前52个字节存放的是程序头的内容，在这里有程序的入口地址等关键内容。之后根据程序头中的内容，我们可以找出这个ELF文件中所有需要加载到内存中的数据，以及应该加载到内存中的哪个位置等等。通过`ramdisk_write()`函数，将这些数据加载到虚拟内存的相应位置（这里一般是`0x83000000`开始），文件大小等。其中`hello`程序会调用`printf`这个函数，在调用这个`printf()`函数将数据输出到显示屏的时候，代码会跳转到`printf()`函数的位置，执行结束会再次跳转到`0x83000000`周围，这样反复执行。

问题四：仙剑奇侠传究竟如何运行

仙剑奇侠传在运行的时候会播放一个动画，而这个动画中的仙鹤的信息时存放在`mgo.mkf`中的。在这个过程中，通过我们原先已经实现的`fs_open()`、`fs_read()`等函数读取文件的内容，读取完成之后设置仙鹤的位置，之后将其显示到显示屏上。

3.5 PA4-虚实交错的魔法：分时多任务

1) 实现基本的多道程序系统

在实现了上下文的切换之后，尝试加载`hello_fun`这个程序，运行批处理系

统，在菜单页面选择 dummy 程序运行，在 dummy 程序运行结束之后，理论上应该跳转到 hello_fun()函数执行，观察执行结果如下所示，表示程序实现成功。

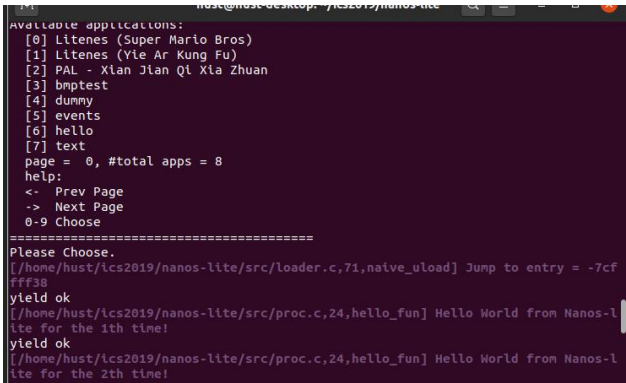


图 3-30 hello_fun()运行结果

之后在实现了_ucontext()函数之后，尝试一遍运行仙剑奇侠传，一遍运行 hello_fun()程序，但是这个时候仙剑奇侠传运行的速度非常慢，动画截图如下所示：

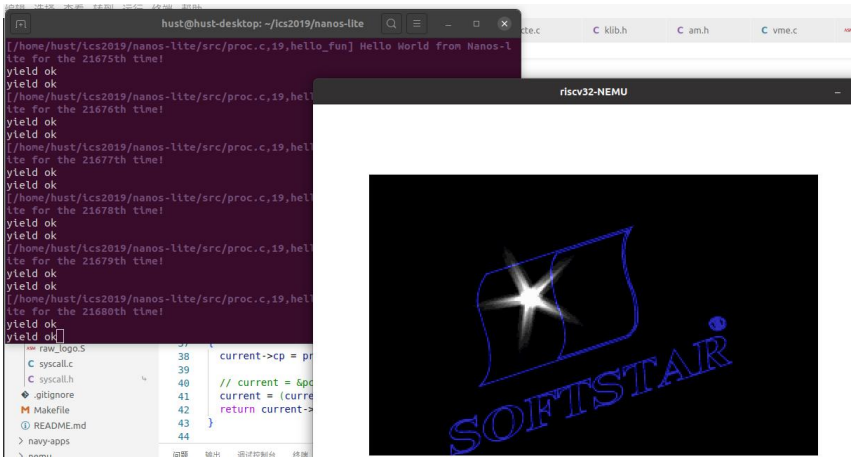


图 3-31 hello_fun()和仙剑奇侠传同时运行结果

4 实验总结

在这个系统能力培养课程中实现了一个简单的指令模拟器。但是分时多任务的部分内容还没有实现。从整体来看，这次实验的任务量不是很大，自己也是一天实现一小部分功能。前三个小节的功能全部实现，其中 PA4 只实现了一个简单的多道程序系统。

其中在 PA1 阶段主要是实现了一个简易的调试器，具有设置断点、表达式求值等基础的功能。其中 PA1 阶段最难的部分应该就是表达式求值的实现。在原有的代码框架中已经给出了一些代码，但是由于不符合自己的编码习惯，就重新写了一份，但是重写的这部分代码对表达式求值时输入的字符串的长度有限制，这算是代码的一个缺陷。除此之外，在自动生成表达式的时候如何保证自己生成的表达式不会出现除 0 这种非法情况也算是一个比较麻烦的地方，不过可以在网上找到类似的类似的解决方法。在这里是通过 `signal()` 函数，如果出现除 0 的非法行为就会捕获，从而输出提示信息给主程序，主程序根据提示信息决定是否将当前表达式写入文件以及是否需要再次生成一个新的表达式。

之后在 PA2 阶段，花费了很多时间在实现指令上，不过指令的实现比较简单，基本上都是重复的工作，同时和组原课设的指令实现有类似的地方。一条指令的执行过程包括取指令、解码、执行、更新 PC 四个阶段。其中需要我们实现的是解码和执行这两个部分，具体来说就是 `make_DHelper()` 和 `make_EHelper()` 这两类函数的实现。另外在这里需要补充一下，在更新 PC 阶段，有一个 `update_pc()` 函数，这个按照讲义的内容是需要我们完善内容的，但是自己在实现的时候直接忽略了这部分内容，而是直接在那些跳转指令的实现过程中直接调用了 `rtl` 伪指令实现了 PC 的更新。另外 VGA 的实现也是一个难点，需要自己精读代码。

而在 PA3 阶段，主要的工作就是实现系统调用等。其中我觉得最麻烦的一个地方就是在实现 `fb_write()` 函数的时候，根据 `offset` 参数计算坐标，这个也是需要自己反复读代码。之后系统调用的实现也几乎都是一个套路，但是自己在实现的时候犯了好几次同样的错误：比如，在 `SYS_open` 系统调用的时候我们需要将 `_syscall_()` 的返回值作为 `_open()` 的返回值，但是由于自己的失误，总是直接返回

0，导致程序陷入死循环。

最后的 PA4 的阶段，自己只实现了一小部分，而且根据讲义的提示即可完成，没有太大的难度。

做完这个实验，最大的一个感触就是一定要仔细阅读讲义，要不然就会出一些奇怪的错误：好几次在终端执行的时候明明代码看了好几遍都没有发现问题，但是总是报错，最后才发现是自己终端运行的指令的参数选错了。但是也不能仅仅只依靠讲义，比如我们在实现 PA3 阶段的时候写过 `fs_close()` 这样一个函数，讲义上面说的直接返回 0 即可，但是实际上我们还需要考虑到 `open_offset` 的问题，这个如果只是返回 0，就会导致后面的批处理系统运行失败。这个时候代码的实现就还需要添加自己的理解。

参考文献

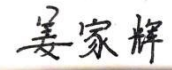
- 【1】勾凌睿、黄成、刘志刚 翻译。RISC-V-Reader-Chinese-v2p1.pdf 2018

一、原创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

特此声明！

作者签字：姜家辉

A handwritten signature in black ink, reading '姜家辉' (Jiang Jiahui), written in a cursive style. The signature is positioned to the right of the printed name '姜家辉'.