华中科技大学

2024

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：	CS2103 班
学　　号：	U202115395
姓　　名：	杜启铭
电　　话：	13526877089
邮　　件：	2454235539@qq.com
完成日期：	2024-09-24

计算机科学与技术学院

目 录

1　课程实验概述

1.1 课设目的

本次课程设计通过实现一个经过简化但功能完备的 riscv32 模拟器 NEMU，最终在 NEMU 上运行游戏“仙剑奇侠传”，来探究“程序在计算机上运行”的基本原理。

1.2 课设任务

本次课程设计主要包含下列实验内容。

1. 实现简易调试器、表达式求值、监视点与断点等功能。

2. 运行一个 C 程序、丰富指令集并测试所有程序、实现 I/O 指令并测试打字游戏。

3. 实现系统调用、实现文件系统、运行仙剑奇侠传。

4. 实现分页机制、实现进程上下文切换、时钟中断驱动的上下文切换。

1.3 实验环境

使用教师提供的 virtual box 镜像。

2 PA1-开天辟地的篇章: 最简单的计算机

2.1 简单调试器

我们需要为 nemu 实现一些简单的调试功能，包括单步执行、打印程序状态、扫描内存、表达式求值、扫描内存、设置监视点、删除监视点。

2.1.1 单步执行 si

cpu_exec 函数实现了执行指定次数的 CPU 循，因此单步执行的功能通过直接调用该函数即可实现。

2.1.2 打印程序状态 info

程序状态包括两种状态：寄存器状态和监视点状态。

定义一个名为 cmd_info 的函数，用于处理 info 命令。根据传入的参数 args，如果参数是 r，则调用 isa_reg_display() 显示寄存器信息；如果参数是 w，则调用 display_watchpoints() 显示监视点信息并返回；如果参数无效，则打印 "Wrong argument!" 错误信息。如果没有提供参数，则打印 "Lack argument!" 错误信息。

2.1.3 扫描内存 x

定义一个名为 cmd_x 的函数，用于处理 x 命令。该函数从参数 args 中解析出要读取的内存单元数量 number 和起始地址 index，如果解析失败或参数无效，则打印错误信息 "Wrong argument!" 并返回。否则，它会循环读取指定数量的内存单元，每读取四个单元打印一行，

调用 isa_vaddr_read 函数读取内存地址的值，并以十六进制格式打印地址和值。

2.2 表达式求值

2.2.1 表达式解析

定义一个名为 make_token 的函数，用于将输入字符串 e 分解成一系列的标记（tokens）。函数通过正则表达式逐个匹配输入字符串中的子串，并根据匹配的规则将子串记录为相应类型的标记。对于不同类型的标记，函数会执行不同的操作。

2.2.2 表达式求值

定义一个名为 calculate 的递归函数，用于计算表达式的值。函数根据传入的标记数组 tokens 和索引范围 [i, j] 解析并计算表达式的值。如果 i 和 j 相等且标记是数字或寄存器，则返回其值；如果表达式被括号包围，则去掉括号递归计算；否则，找到主操作符并递归计算其左右操作数的值，并根据操作符类型执行相应的运算。函数通过 success 标志指示计算是否成功，并处理各种运算符和错误情况，如除零错误和无效表达式。

2.3 设置与删除监视点

我们需要实现：监视点的创建、删除、打印、检查。

监视点的创建。定义一个名为 new_wp 的函数，用于分配一个新的监视点（WP）。如果没有可用的空闲监视点，则打印错误信息并触发断言失败。否则，从空闲链表中取出一个监视点，将其插入到活动

监视点链表的头部，并返回该监视点的指针。

监视点的删除。定义一个名为 free_wp 的函数，用于根据监视点编号 NO 从活动监视点链表中删除相应的监视点；如果找到该监视点，则将其从链表中移除并添加到空闲监视点链表中，以便后续复用

监视点的打印。定义一个名为 display_watchpoints 的函数，用于显示当前所有活动的监视点。函数首先打印表头，然后遍历活动监视点链表 head，对于每个监视点，打印其编号 (NO)、表达式 (wp_expr) 和上次计算的结果 (last_value)。

监视点的检查。定义一个名为 check_watchpoints 的函数，用于检查所有活动的监视点是否发生变化。函数遍历监视点链表 head，计算每个监视点表达式的当前值 res，如果计算失败则打印错误信息并触发断言失败；如果当前值与上次记录的值 last_value）不同，则打印监视点信息和新旧值，并更新 last_value。函数返回一个布尔值 result，指示是否有监视点发生了变化。

2.4 运行结果

简单调试器的功能已基本实现。

2.5 必答题

这里我们来对实验文档中的必答题进行逐一的解答。

1.我选择的 ISA 是__riscv32__。

2.用于调试的时间为 10 小时，实现简单调试器后的调试时间将降低为 2 小时，由此可见，通过实现一定的基础设施，可以有效的减

少我们在后续工作中 debug 的工作量。

3. riscv32 的指令格式有：R 型（Register）、I 型（Immediate）、S 型（Store）、B 型（Branch）、U 型（Upper Immediate）、J 型（Jump）

LUI（Load Upper Immediate）指令将一个 20 位的立即数加载到寄存器的高 20 位，低 12 位填 0。

mstatus 包含字段：MIE 机器模式全局中断使能、MPI 机器模式中断使能前值、MPP 机器模式前模式、FS 浮点状态、XS 扩展状态、SD 状态脏位。

4.

（1）nemu/目录下的所有.c 和.h 文件总共有多少行代码?

使用以下命令统计代码行数：

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

（2）和框架代码相比, 你在 PA1 中编写了多少行代码?

两次使用（1）中的命令并计算结果。

（3）将统计代码行数的命令写入 Makefile 中。

在 Makefile 添加以下内容：

```
count:
    find nemu/ -name *.c -o -name *.h | xargs wc -l
```

运行 make count 即可统计代码行数。

（4）除去空行之外, nemu/目录下的所有.c 和.h 文件总共有多少行代码?

使用以下命令统计除去空行的代码行数：

```
find nemu/ -name *.c -o -name *.h | xargs grep -v ^\s*$$ | wc -l
```

5. -Wall 和-Werror 是 GCC 编译器中的两个常用选项：

-Wall：

（1）启用所有常见的警告选项，帮助开发者发现潜在的代码问题。

（2）包括未使用的变量、未初始化的变量、隐式函数声明等警告。

-Werror：

（1）将所有警告视为错误，编译器在遇到警告时会停止编译。

（2）强制开发者修复所有警告，确保代码质量。

为什么要使用-Wall 和-Werror：

（1）提高代码质量：通过启用警告，开发者可以发现并修复潜在的代码问题。

（2）强制修复警告：将警告视为错误，确保所有警告都被修复，避免潜在的运行时错误。

（3）保持代码整洁：减少代码中的潜在问题和不良实践，保持代码库的整洁和可维护性。

3　PA2-简单复杂的机器：冯诺依曼计算机系统设计

3.1 在 NEMU 中运行第一个 C 程序 dummy

首先浏览 dummy-riscv32-nemu.txt，找到需要实现的指令有：li、auipc、addi、jal、mv、sw、jalr。

为了实现一条新指令，需要：

（1）在 opcode_table 中填写正确的译码辅助函数，执行辅助函数、操作数宽度。

（2）用 RTL 实现正确的译码辅助函数和执行辅助函数。使用 RTL 伪指令时要遵守小型调用约定。

下面以 LUI 指令为例，介绍译码辅助函数和执行辅助函数的实现及 opcode_table 的填写。LUI（Load Upper Immediate）指令用于将一个立即数加载到寄存器的高 20 位。

opcode_table 是一个包含 32 个 OpcodeEntry 的数组，用于根据指令的 opcode 字段选择相应的译码辅助和执行辅助函数。每个 OpcodeEntry 包含一个译码辅助函数和一个执行辅助函数。

LUI 指令的 opcode 为 0b0110111，对应 opcode_table 中的第 13 个条目。

```
1. static OpcodeEntry opcode_table [32] = {
2.   // 其他条目省略...
3.   /* b01 */ IDEX(st, store), EMPTY, EMPTY, EMPTY, IDEX(R, r), IDE
   X(U, lui), EMPTY, EMPTY,
4.   // 其他条目省略...
5. };
```

U 是译码辅助函数，用于解析 U 型指令。

lui 是执行辅助函数，用于执行 LUI 指令。

译码辅助函数用于解析指令并提取操作数。对于 LUI 指令，我们需要解析立即数和目标寄存器。

```
make_DHelper(U) {
  // 解析立即数并左移 12 位
  decode_op_i(id_src, decinfo.isa.instr.imm31_12 << 12, true);
  // 解析目标寄存器
  decode_op_r(id_dest, decinfo.isa.instr.rd, false);
  // 打印操作数信息
  print_Dop(id_src->str, OP_STR_SIZE, "0x%x", decinfo.isa.instr.imm31_12);
}
```

执行辅助函数用于执行指令的具体操作。对于 LUI 指令，我们需要将立即数加载到目标寄存器的高 20 位。

```
static inline void exec_lui(DecodeExecState *s) {
  // 将立即数加载到目标寄存器
  rtl_li(&s->dest->val, s->src1->imm);
}
```

3.2 实现指令，运行所有 cputest

1. 剩余指令的实现

与 LUI 指令的实现类似，剩余指令的实现大致相同。指令的实现很繁琐，需要的时间很多。

2.字符串处理函数的实现。

待实现的都是比较常见的函数，不再赘述。

3.printf 的实现。

printf 函数

```
int printf(const char *fmt, ...) {
  va_list ap;
  va_start(ap, fmt);
  char buf[1024] = {0};
  int cnt = vsprintf(buf, fmt, ap);
```

```
6.    for(int i = 0; i < cnt; i++) {
7.      _putc(buf[i]);
8.    }
9.    va_end(ap);
10.   return cnt;
11.}
```

（1）使用 va_list 类型定义 ap 来存储可变参数。

（2）调用 va_star 初始化 ap。

（3）定义一个缓冲区 buf 来存储格式化后的字符串。

（4）调用 vsprintf 将格式化字符串和参数写入 buf。

（5）使用_putc 将 buf 中的字符逐个输出。

（6）调用 va_end 结束可变参数处理。

（7）返回写入的字符数。

vsprintf 函数

```
1. int vsprintf(char *out, const char *fmt, va_list ap) {
2.   int cnt = 0;
3.   for(int i = 0; fmt[i]; i++) {
4.     if(fmt[i] != '%') {
5.       out[cnt++] = fmt[i];
6.     } else {
7.       i++;
8.       switch(fmt[i]) {
9.         case 'd': ...
10.        case 's': ...
11.        case 'c': ...
12.        case 'x': ...
13.        case 'u': ...
14.        default: break;
15.      }
16.    }
17.  }
18.  out[cnt] = '\0';
19.  return cnt;
20.}
```

（1）初始化字符计数器 cnt

（2）遍历格式化字符串 fmt。

（3）如果当前字符不是%，直接将其写入 out。如果是%且下一个字符是 d/s/c/x/u，相应地处理后续的格式化字符。

sprintf 函数和 snprintf 函数的实现与 printf 函数的实现大致相同。

3.3 输入输出

（1）实现_DEVREG_TIMER_UPTIME 的功能

实现两个主要函数：__am_timer_read 和 __am_timer_init。

__am_timer_read 根据传入的寄存器地址读取定时器信息，支持读取系统启动时间_DEVREG_TIMER_UPTIME 和当前日期 _DEVREG_TIMER_DATE。

__am_timer_init 初始化定时器，记录系统启动时的时间。通过访问硬件寄存器 RTC_ADDR，这些函数获取并处理时间信息。

（2）实现_DEVREG_INPUT_KBD 的功能

实一个函数 __am_input_read，用于读取键盘输入。当寄存器地址为 _DEVREG_INPUT_KBD 时，它从键盘地址 KBD_ADDR 读取键盘状态，并将按键状态和按键码存储到 _DEV_INPUT_KBD_t 结构中。

（3）实现_DEVREG_VIDEO_INFO 的功能

实现三个主要函数：

__am_video_read：读取视频信息。根据寄存器地址读取屏幕宽度和高度，并存储在 info 结构中。

__am_video_write：将帧缓冲区中的像素数据写入视频内存。根据寄存器地址判断操作类型。将像素数据逐行拷贝到视频内存。如果需要同步显示，则调用同步函数。

__am_vga_init：初始化 VGA 显示。计算屏幕的总像素数。将帧缓冲区中的每个像素设置为其索引值。调用同步函数以更新显示。

3.4 运行结果

riscv32 大部分指令的功能基本实现。三个输入输出接口成功实现。

3.5 必答题

1. nemu 中一条指令的执行流程可以分为四个步骤：取指、译码、执行、更新 PC。

2. 去掉 static 后，函数的链接属性变为外部链接。这意味着该函数在多个源文件中可能会有多个定义，从而导致链接器错误（重复定义）。

去掉 inline 后，函数不再是内联函数，编译器可能会为每个调用生成一个函数调用，从而增加函数调用的开销。但不会导致编译或链接错误。

3.

（1）在 nemu/include/common.h 中添加 volatile static int dummy; 并重新编译。

结果，由于 dummy 变量被定义为 static，它在每个包含 common.h 的源文件中都是一个独立的实体。因此，编译后的 NEMU 将包含多个 dummy 变量的实体，每个包含 common.h 的源文件都会有一个 dummy 变量。

可以通过以下步骤验证：

1）在 common.h 中添加 volatile static int dummy;。

2）运行 make clean 清理之前的编译结果。

3）运行 make 重新编译 NEMU。

4）使用 nm 命令查看生成的目标文件，检查 dummy 变量的定义。

（2）在 nemu/include/debug.h 中添加 volatile static int dummy; 并重新编译。

结果，由于 dummy 变量被定义为 static，它在每个包含 debug.h 的源文件中也是一个独立的实体。因此，编译后的 NEMU 将包含更多的 dummy 变量的实体，每个包含 common.h 和 debug.h 的源文件都会有一个 dummy 变量。

与上题相比，添加到 debug.h 中的 dummy 变量会增加 dummy 变量的实体数量，因为 debug.h 可能被更多的源文件包含。

（3）为两处 dummy 变量进行初始化

将 common.h 和 debug.h 中的 dummy 变量初始化为 0。然后重新编译 NEMU。

结果，编译器可能会报错，提示重复定义 dummy 变量。这是因为 static 变量在每个包含它的源文件中都是独立的，但初始化会导致链接器错误。

之前没有出现这样的问题是因为 static 变量在每个源文件中都是独立的实体，不会在链接阶段冲突。但一旦进行初始化，编译器会尝试在多个源文件中初始化同一个变量，从而导致链接器错误。

4. 在 nemu 目录下执行 make 后，make 程序会根据 Makefile 的规则组织 .c 和 .h 文件，最终生成可执行文件 nemu/build/$ISA-nemu。以下是一个简要的过程描述：

（1）读取 Makefile

make 程序首先读取 Makefile 文件，并解析其中的变量和规则。

（2）设置目标 ISA

根据 Makefile 中的条件判断，设置目标 ISA（指令集架构）。

```
1. ifneq ($(MAKECMDGOALS),clean) # ignore check for make clean
2. ISA ?= riscv32
3. ISAS = $(shell ls src/isa/)
4. $(info Building $(ISA)-$(NAME))
```

这段代码会设置 ISA 变量为 riscv32，并列出 src/isa/目录下的所有 ISA。

（3）查找源文件

Makefile 会定义源文件的路径和模式，通常会使用通配符查找所有的 .c 文件。

```
1. SRCS = $(wildcard src/**/*.c)
```

这行代码会查找 src/ 目录及其子目录下的所有 .c 文件，并将它们存储在 SRCS 变量中。

（4）编译源文件

Makefile 会定义如何编译每个源文件。通常会使用一个编译规则。

```
1. OBJS = $(SRCS:.c=.o)
2. %.o: %.c
3.     $(CC) $(CFLAGS) -c $< -o $@
```

这段代码会将所有的 .c 文件编译成对应的 .o 文件。

（5）链接目标文件

Makefile 会定义如何将所有的目标文件链接成最终的可执行文件。

```
1. nemu/build/$(ISA)-nemu: $(OBJS)
2.     $(CC) $(LDFLAGS) -o $@ $^
```

这行代码会将所有的 .o 文件链接成最终的可执行文件 nemu/build/$(ISA)-nemu。

（6）生成可执行文件

最终，make 程序会执行所有的编译和链接命令，生成可执行文件 nemu/build/$(ISA)-nemu。

4　实验结果与结果分析

参考文献