PA2 实验报告

翟笑晨

2024 年 11 月 23 日

目录

1 不停计算的计算机

1.1 理解 YEMU 如何执行程序

状态机 (初始的 pc 值不妨设为 0)：$(pc, R[0], R[1], M[z]) \to (0,0,0,0) \to (1,y,0,0) \to (2,y,y,0) \to (3,x,y,0) \to (4,x+y,y,0) \to (5,x+y,y,x+y)$

1.2 通过 RTFSC 理解 YEMU 如何执行一条指令

YEMU 执行指令的过程也是按照: 取指 → 译码 → 执行 → 更新 PC 的流程进行的.
首先 $this.inst = M[pc]$ 取出 pc 对应的指令, 然后使用 $switch$ 来对操作码、操作数分别译码, 并执行对应的操作, 最后使用 $pc++$ 更新 pc.

1.3 想法和心得

通过学习取指 (IF)、译码 (ID)、执行 (EX) 和更新 PC 的过程，我明白计算机本质上就是一个不断重复这些步骤的状态机。虽然 YEMU 只是个简单的 CPU 模拟器，但它展示了即使是最基础的指令集，通过合理组合也能实现复杂的功能。这让我更加理解了计算机系统的设计之精妙。

2 RTFSC(2)

2.1 回答蓝框问题

2.1.1 立即数背后的故事

由于 Motorola 68k 系列的处理器是大端架构的，如果我们要把 NEMU 的源代码编译成 Motorola 68k 的机器码，我们可以设计一个 reverse 函数将指令中的立即数按字节进行翻转即可。
但是假设我们需要把 Motorola 68k 作为一个新的 ISA 加入到 NEMU 中的话, 我们就需要整个地修改取立即数的逻辑, 具体则是修改译码的逻辑部分, 感觉实现起来不难.

2.1.2 立即数背后的故事 (2)

MIPS32 和 RISCV32 通过将 32 位常数分解成高位和低位，然后通过两个或多个指令合成目标常数来解决这个限制。例如在 RISCV32 中我们使用

lui x5, 0x12345,

addi x5, x5, 0x678

来加载常数 0x12345678.

2.2 RTFSC 理解指令执行的过程

在 NEMU 中，一条指令的执行过程如下：

(1) 取指:

使用 isa_exec_once() 函数传入 $s \to snpc$ 并从内存中读取指令，更新 PC。

(2) 译码:

调用 decode_exec() 函数，解析指令的操作码和操作数。

(3) 执行:

根据模式匹配的指令类型执行相应操作，例如算术运算。

(4) 更新 PC:

更新 $s \to dnpc$ 值，准备执行下一条指令。

(5) 循环执行:

在 exec_once() 函数中循环执行，直到遇到停止条件。

2.3 准备交叉编译环境

2.4 运行第一个客户程序

实现了 addi, ret 等指令之后程序可以 HIT GOOD TRAP.

2.5 实现更多的指令

根据 RISCV 官方手册实现了 RV32I 的所有基本指令.

2.6 想法和心得

学完这一节我有很多想法体会:

1. 做笔记整理细节，配合 GDB 调试加深理解, 通过 RTFM 和 RTFSC 来理解系统.
2. 避免 Copy-Paste 的糟糕习惯，它会带来维护困难和潜在 bug, 注重代码的结构化和解耦，提高可维护性.
3. 尽早进行测试，而不是实现所有功能后才测试 (太重要了!!!)
4. 指令执行是一个复杂的过程，需要理解取指、译码、执行等步骤. 同时代码实现要考虑通用性和可维护性

这些内容不仅对完成 PA 项目有帮助，也是培养良好工程素养的重要经验。

3 程序, 运行时环境与 AM

3.1 回答蓝框题

3.1.1 这又能怎么样呢

抽象在编程中带来的好处还有简化复杂性、提高可重用性、增强可维护性、提高灵活性。

3.2 通过批处理模式运行 NEMU

通过 RTFSC 我们可以通过在 nemu.mk 中的 NEMUFLAGS 中添加一个'-b' 参数开启批处理模式, 从而实现一键回归测试的功能. 的确, 在对之前的 tests 进行一键回归测试来验证 inst.c 的实现是否正确的时候, 倘若我们还是一个一个的 c 的话效率是非常的低, 这也间接地说明了了解项目中文件的构建规则和构建关系是非常重要的.

3.3 实现字符串处理函数

我们暂时只用实现 vsprintf 即可.

3.4 实现 sprintf

参考之前的 vsprintf 实现.

4 基础设施 (2)

这部分我完成了所有内容

4.1 遇到的问题和思考

在 utils/tools 目录下单独写了一个 itrace.c 文件, 但是在 make run 的时候 cpu_exec() 函数老是报错找不到 itrace.c 中定义了的函数, 在 RTFSC 无果之后求助有爱的助教和有类似经历的同学之后发现竟然只是一个小小的引用头文件 utils.h 就能解决的事情, **只能说, 不能只阅读理解有 TODO 的地方, 而是需要尽可能多地理解阅读 Project 中的所有代码.**

4.2 实现 iringbuf

实现难度不大, 就用一个结构体数组即可

4.3 实现 mtrace

我们在 itrace.c 中实现 pread 和 pwrite 两个函数, 在访存函数 paddr_read() 和 paddr_write() 中调用即可实时 trace 是否有越界行为.

4.4 回答蓝框题

4.4.1 消失的符号

在 am-kernels/tests/cpu-tests/tests/add.c 中，宏 NR_DATA、局部变量 c 和形参 a、b 在符号表中之所以找不到对应的表项，是因为局部变量 c、形参 a、b 的作用域仅限于 add() 函数内部，函数结束后它们被销毁，因此不会在符号表中记录。而宏 NR_DATA 在预处理阶段被展开，不会在符号表中生成条目。
而符号是程序中定义的变量、函数等的名称。只有全局作用域或文件级别的符号会出现在符号表中。局部符号在函数外不可见，因此不会被记录。

4.4.2 寻找"Hello World!"

在字符串表中，“Hello World!”字符串的地址通常会显示为一个偏移量，这个偏移量是相对于字符串表的起始位置的。具体位置可能会因编译器和

系统的不同而有所变化，但一般来说，**字符串表的偏移量会在 ELF 文件的.strtab 节中**。

4.5 实现 ftrace

我们需要在 itrace.c 中添加分别用来追踪函数调用和返回的函数, 根据文档的提示，call（调用）和 ret（返回）都需要记录指令所在的地址，用参数 pc 表示。注意: 为了便于阅读 ftrace 的输出，需要将函数地址转换为函数名显示. 同时我们还需要对 jal 和 jalr 指令的译码部分进行修改.

4.6 回答蓝框题

4.6.1 不匹配的函数调用和返回

在函数调用时，call 指令会将返回地址压入栈中。若在函数内部发生了其他的 call，而没有正确的 ret 配对，可能会导致返回到错误的地址。同时编译器在优化过程中可能会将某些函数内联，或者在某些情况下改变函数的调用顺序。这可能导致 call 和 ret 之间的匹配关系被打破。

4.6.2 冗余的符号表

编写并编译一个 Hello World 程序后，使用 strip 命令丢弃符号表，生成的可执行文件仍然可以成功运行。这是因为符号表主要用于调试和链接，程序的执行不依赖于符号表的存在。但是当对目标文件 hello.o 使用 strip 命令丢弃符号表后，再进行链接时，可能会遇到问题。目标文件中的符号表用于链接器解析符号引用，如果符号表被丢弃，链接器将无法找到函数和变量的定义，导致链接失败。
这可能是因为可执行文件中符号表存在与否不影响程序执行, 而是影响调试; 但是目标文件中的符号表对链接是至关重要的.

4.7 想法和心得

1. 调试思维的重要性 通过 trace 工具（如 itrace、mtrace、ftrace）来追踪程序执行, 善用 GDB 等调试工具，不要盲目修改代码
2. 工程化思维 注重代码的可维护性和可移植性. 建立良好的基础设施，提高开发效率

这些内容不仅对完成当前项目有帮助，更是培养良好工程素养的重要经验。通过这些实践，我们能更好地理解计算机系统的工作原理。

5 输入输出

这部分我完成了除了声卡以外的所有内容.

5.1 回答蓝框问题

5.1.1 理解 volatile 关键字

volatile 关键字在 C 语言中是为了确保程序能够正确地与外部环境（如硬件设备）交互。它的存在是为了防止编译器在优化时做出错误的假设，确保每次访问变量时都能获取最新的值。

(1) 带有 volatile 的情况：

编译器会生成代码，确保每次循环都从内存中读取 *p 的值，可以看到以下的汇编指令：

```
movb _end, %al
cmpb $0xff, %al
jne .L1
```

(2) 去掉 volatile 的情况：

编译器可能会优化掉对 *p 的重复读取，生成的代码可能会是：

```
movb $0, _end
jmp .L1
```

在这种情况下，编译器可能会认为 *p 的值在循环中不会改变，因此只会在循环开始时读取一次，导致程序在某些情况下无法正常工作。

5.1.2 理解 mainargs

在 $AM_HOME/scripts/platform/nemu.mk 中

```
CFLAGS += -DMAINARGS=\"$(mainargs\)"
```

gcc 的编译选项-D 定义宏 MAINARGS, 并且将宏的值设为"$mainargs", 也就是字符串 “I-love-PA” .

5.2 运行 Hello World

千万别忘记在 NEMU 的 menuconfig 中开 Device 的开关, 不然总是会报错 $0xa0000000$ 地址越界, 事实上这个是设备 IOE 的地址!

5.3 实现 printf & 运行 alu-tests

前面 inst.c 的粗心错到这里之后被无限放大, debug de 到让人绝望, 从反汇编到 nemu 到 am 无缝切换, 只能说前面埋下的祸根总会在背后使坏.(T_T)

5.4 实现 IOE

这一部分帮助我完成了 printf 实现中的 bug, **天知道为什么我每每实现后面的功能的时候都在修前面代码的 BUG**. 心累.

5.5 看看 NEMU 跑多快

最开始在经过 3968136.272ms 之后, 我的 NEMU 在 microbench 中跑分结果为 5 分, 非常不合理。但是根据将讲义上的指示修改了时钟之后发现跑分为 205, 正常，因此在查阅之后允许 vDSO 访问时钟解决了当前问题。

5.6 实现 dtrace

我们直接在 itrace.c 中添加两个函数 dread 和 dwrite, 随后在 map_read 和 map_write 中调用即可追踪设备调用.

5.7 实现 IOE(2)

参考 native 版本的 ioe 即可.

5.8 实现 IOE(3) & 实现 IOE(4)

通过 RTFSC 可以知道, ___am_gpu_config() 函数用于读取 VGA 参数, 而 ___am_gpu_fbdraw() 函数则是将像素点的 RGB 数据搬运到 frame buffer 中, 然后通过 vga_update_screen() 函数来实现屏幕的更新. 理解这些原理之后, 我们就可以在 nemu 的 ioe 模块中的 gpu.c 中添加上述配置.

5.9 声卡

由于虚拟机很难甚至不能调用宿主机的声卡, 因此我跳过了这一步骤.

5.10 游戏是如何运行的

NEMU 模拟键盘硬件捕获按键事件，将按键扫描码写入键盘控制器寄存器。而 ISA 层通过指令读取按键状态。AM 层将按键事件封装为高层接口，通过 io_read(AM_INPUT_KEYBRD) 提供给程序。最后程序检测按键，调用 check_hit() 检查命中。

5.11 编译与链接

结果：删除 static 或 inline 都不会导致编译错误，但若两者同时删去则会出现错误。错误原因在于多个文件声明了 inst_fetch 函数，未加 static 和 inline 时，inst_fetch 头文件中的代码被多个文件包含，每个文件都定义了一个全局的 inst_fetch 函数，从而引发重复定义错误。添加 static 后，函数变为局部函数，外部文件无法访问，从而避免重复定义。添加 inline 后，inst_fetch 成为内联函数，编译器会在调用处展开汇编代码，不会为该函数生成独立的汇编代码，因此也不会导致重复定义。

5.12 编译与链接 (2)

1. 重新编译后含有 35 个 dummy。我们在终端中使用 findbuild-name ” *.o” | xargs readelf–symbols | grep ” dummy” | wc-l 指令, 结果为 35.
2. 数目仍为 35 个。这是因为 common.h 中已经引用过了 dubug.h, 未初始化的相同名字的局部变量在符号表中只有一个实体。
3. 对变量进行初始化之后再编译会出现 redefinition 错误。这是因为赋了初值的 static 变量会被认为是一个强符号。而 C 语言中不允许有两个相同的强符号被定义。

5.13 了解 Makefile

hello 目录的 Makefile 首先配置 SRC 和 NAME 变量，然后导入 AM 的 Makefile，将 AM 的源文件加入 SRC，并按隐含规则将其编译成.o 文件存

放于构建目录（build）中，同时更新 OBJ 和 LIB 目标。接着根据 ARCH 变量加载 script 目录下对应的 mk 文件（如 riscv32-nemu.mk），设置编译链接参数。之后将自定义库函数（klib）打包成归档文件，编译 SRC 中的源文件并链接成 elf 可执行文件。最后通过 nemu.mk 生成指令文件（txt）和镜像文件（bin），并通过 –image 参数将镜像加载到 nemu 内存中执行。

5.14 想法和心得

在实现输入输出这一部分的过程中, 我深刻地领会到了 “先完成后完美” 这一原则在系统开发中地重要性, 在系统开发中，首先确保功能的完整性和正确性比过早优化更重要, 而过早追求局部完美可能会影响全局目标的实现.

同时我们要采用 “动静结合的学习方法”, 即通过 RTFSC 静态理解代码和通过 trace 等工具动态观察程序行为两种方法结合, 我们才能深入理解程序如何运行.

最重要的一点是在实现某些功能的时候一定要及时进行**单元测试**, 防止对后续的功能实现产生影响. 具体来说, 比如我在写 inst.c 的时候有些指令的模式匹配码写错了, 某些指令的执行功能写混了, 这导致我在后面进行 alu-test, string-test 等测试的时候报了很多次错, 导致我不得不花大量的时间来寻找错误, 浪费了很多本可以用于写 PA2 的时间.

最后非常感谢陈奕睿同学能够和我一起找 bug 的根源并且不断地测试纠错, 以及非常感谢助教的耐心和帮助.