PA1 实验报告

翟笑晨

2024 年 10 月 12 日

1 开天辟地的篇章

1.1 必答题

1.1.1 尝试理解计算机如何计算

计算机根据计算的目标，选择合适的操作数，使用基本的指令修改寄存器的值；根据程序的流程，通过改变程序计数器（PC）的值来控制指令的执行顺序以及循环的过程。

1.1.2 从状态机视角理解程序执行

$(pc, r_1, r_2) : (0, x, x) \to (1, 0, x) \to (2, 0, 0) \to (3, 0, 1) \to (4, 1, 1) \to (5, 1, 1) \to (3, 1, 2) \to (4, 3, 2) \to (5, 3, 2) \to ... \to (3, 4851, 99) \to (4, 4950, 99) \to (5, 4950, 99) \to (3, 4950, 100) \to (4, 5050, 100) \to (5, 5050, 100) \to (6, 5050, 100) \to (6, 5050, 100) \to ...$

1.2 回答蓝框问题

1.2.1 计算机可以没有寄存器吗？

即使没有寄存器，计算机仍然可以运行。但是，这样做会导致每次操作时都需要直接与内存进行交互以获取操作数和存储结果，从而大幅增加执行所需的时间，并且更有可能访问到受限的内存区域。若缺少寄存器，指令需要直接包含操作数在内存中的位置，这将导致指令长度的增加。此外，由于需要频繁地访问内存，编程模型的执行效率及代码效率都将受到影响。

1.3 想法和心得

数电课上有提到过通过枚举系统电路在所有情况下的可能状态并分析彼此之间的转移关系，我们可以得到系统的状态转移表并以此设计时序逻辑电路。现在我们用相同的方法描述程序或计算机状态。计算机就是基于组合逻辑电路和时序逻辑电路实现，用状态图表示程序的状态转换有利于深入理解程序执行的过程。

2 RTFSC

这部分我基本理解了 NEMU 的文件框架。

2.1 回答蓝框问题

程序的执行通常从主入口点开始，这个入口点在不同编程语言中可能有不同的名称。例如，在 C 或 C++ 中，程序的执行始于 main() 函数；在 Python 中，则是从脚本的第一行代码开始执行.
这个入口点标志着操作系统加载程序并开始执行程序代码的位置，它是每个程序必须具备的部分，以确保程序能够被系统正确调用和执行。从这个入口点开始，程序将按照代码中定义的逻辑顺序执行，直到程序终止或遇到结束执行的命令。

2.2 必答题

2.2.1 理解代码框架

2.2.2 优美地退出

报错的原因是未修改框架代码直接退出时，nemu_state.state 的值为 NEMU_STOP，所以在 main 函数中的 is_exit_state_bad() 函数中返回了 1，导致报错。发现当执行过 cmd_q 之后变量值改成了 NEMU_QUIT，于是不会报错，故只需要在执行 cmd_q 时将变量的值改为 NEMU_QUIT。

2.3 回答蓝框问题

2.3.1 究竟要执行多久？

由于 cpu_exec() 的参数是 uint64_t，所以传入-1 相当于传入 uint64_t 中最大的无符号整数, 让 cpu 将程序执行完。

2.3.2 潜在的危胁

不属于未定义行为, 根据 C99 标准，无符号整数不能表示负数。尝试传递-1 给一个 uint64_t 类型的参数，这个-1 实际上会被转换为该无符号类型的最大值。这是由于无符号整型在接收负数时会进行模运算，将负数转换为相应的正数，通常是按照 $2^n - 1$ 的形式，其中 n 是该数据类型的位数。

2.4 想法和心得

在深入这个包含大量文件的工程并解决 PA1 时，我开始有了一些全新的认识:
宏是一个相当强大的预编译工具, 跟不用说 Macro-X 这样的函数预编译工具了。在这个庞大的项目中，宏不仅可以统一定义所有文件中需要的变量，还能避免循环引用、增强代码的可读性等多项功能。
命令行工具也是，现在项目中有许多文件，我常常使用 ls、grep、git 等命令来实现基本的增删查改，这使我更加体会到 Linux 系统工具功能的“专一而强大”。

3 基础设施

这部分我完成了所有内容

3.1 遇到的问题和思考

找 cpu.gpr 找了好久, 致敬传奇“从 sdb 找到 nemu-main” :(
扫描内存时，发现无法引用函数 guest_to_host()，选择向头文件中添加 paddr.h 文件。在一个项目中，处理好文件之间的引用关系、熟悉各个变量和函数的作用范围也更重要。

3.2 回答蓝框问题

3.2.1 如何测试字符串处理函数？

打印经过处理的字符串和处理后得到的变量，观察是否符合程序后续执行的需求。

3.3 想法和心得

读懂代码的重要性日渐凸显，我逐渐认识到，这不仅仅是一个挑战，更是一项关键技能。深入理解现有的代码是编写新代码的前提。只有当我们完全理解一个程序的工作原理时，我们才能有效地进行扩展或优化。此外，阅读并分析精心设计的 PA 代码框架，可以提高我们的编程技术，使我们能够创造出既高效又易于维护的软件解决方案。这种能力，使得编程不仅仅是一种技术行为，更是一种艺术。

4 表达式求值

这部分我完成了所有内容

4.1 遇到的问题和思考

在处理字符串时使用指针需要格外小心，以避免超出界限的问题。例如，使用一个简单的变量进行加减操作来检查括号是否匹配，虽然这种方法涉及多个判断条件，但它能有效解决问题。在这个阶段，确保代码的功能性比追求完美更重要，这不仅保证了程序的正确执行，还便于后续的调整和优化。

此外，对于不符合规则的表达式，必须实现严格的错误处理机制，否则可能引发难以预料的 bug，甚至导致程序因内存访问错误而意外崩溃。对于复杂的项目，代码需要综合考虑所有潜在的异常情况，以确保系统的整体稳定性和可维护性。

在测试功能时，如果直接在 main 函数中调用 expr 函数可能会遇到 “Segmentation fault” 的错误。这通常是因为数组或指针未被初始化而直接访问了非法内存。通过将测试过程放置在 sdb 调试环境中，并通过定义一个新

指令 t 来专门测试 expr 函数，可以确保所有必要的变量在 sdb 环境下已正确定义，从而安全地校验表达式的有效性。

4.2 回答蓝框问题

4.2.1 为什么 printf() 函数的输出要换行？

不换行的话下一次输出会直接在这一行的后面输出，导致输出端格式混乱和信息冗杂。

4.2.2 表达式生成器如何获得 c 程序的打印结果？

在 sdb.c 中插入这么一个函数:

```
void test_expr() {
  FILE *fp = fopen("ics2024/nemu/tools/gen-expr/input", "r");
  if (fp == NULL) perror("test_expr error");

  char *e = NULL;
  uint32_t correct_res;
  size_t len = 0;
  size_t read;
  bool success = false;

  while (true) {
    if(fscanf(fp, "%u ", &correct_res) == -1) break;
    read = getline(&e, &len, fp);
    e[read-1] = '\0';

    uint32_t res = expr(e, &success);

    assert(success);
    if (res != correct_res) {
      puts(e);
      printf("expected: %u, got: %u\n", correct_res, res);
```

```
      assert(0);
    }
  }

  fclose(fp);
  if (e) free(e);

  Log("expr test pass");
}
```

4.3 想法和心得

原文提到的程序设计课程通常侧重于算法的编写与实现，而现在的挑战在于如何将这些算法有效地融入到整个项目中。这要求我们不仅要关注算法本身的优劣，还要考虑到项目的整体架构和各个部分的协同工作能力。因此，处理所有可能的情况成为了关键。
在之前的程序设计课程中，重点主要放在算法的开发和实现上。然而，当前的任务更为复杂，要求我们不仅要掌握算法本身，还要能够将其顺利集成到整个项目中。这一过程考验的不仅是对算法的理解，还有对项目结构和各模块之间相互作用的把控能力。因此，全面应对各种情况变得尤为重要。

5 监视点

5.1 遇到的问题和思考

这部分遇到很多链表操作，但是还好增加删除操作 (new & free) 只是将一个链表上的 Node 节点转到另一个上, 感觉还好。

5.2 回答蓝框问题

5.2.1 温故而知新

在框架代码中，定义 wp_pool 等变量时使用了 static 关键字，这意味着这些变量的作用域限制在它们被定义的文件内，其他文件无法直接访问。
在这种情况下，使用 static 的原因主要有两个。首先，它有助于维护模块之间的独立性，确保变量不会被外部文件误用或篡改。其次，通过限制变量的作用域，代码的可维护性和可读性得以提升，因为程序员可以更清晰地理解哪些部分依赖于哪些变量，而不必担心其他模块的干扰。因此，在 watchpoint.c 文件中定义检查监视点变化的函数，并在 sdb.h 中声明该函数后，在 cpu_exec 中调用该函数，进一步保持了文件间的清晰引用关系。

5.2.2 如何提高断点的效率？

由于每次指令执行完成之后都要逐个判断是否满足断点的条件，这会大大降低程序运行的速度。可以考虑指令执行结束之后只检测指令涉及到的寄存器相关的断点。

5.2.3 一点也不能长？

在 x86 架构中，int3 指令的长度为 1 个字节。假设在变种架构 my-x86 中，int3 指令的长度改为 2 个字节，断点机制依然可以正常工作，原因如下：(1) 指令解码：只要 my-x86 能够识别 int3 的操作码，并将其解析为断点指令，系统就能正确处理断点。(2) 调试工具：调试工具可以适应新的 int3 指令长度，只需确保在设置和移除断点时处理的是 2 字节的 int3 指令。(3) 中断处理：断点触发的中断不依赖于指令长度，而是依赖于操作码和中断向量配置。只要配置正确，断点机制就能正常运作。因此，my-x86 中的断点机制可以正常工作，只要能够正确处理新的 int3 指令长度。

6 如何阅读手册

6.1 必答题

6.1.1 程序是个状态机

上文已经回答

6.1.2 理解基础设施

约大于 9000 秒

6.1.3 RTMF

1. riscv32 有 6 种指令类型，R、I、S、B、U、J。
2. LUI 用于将一个 20 位的立即数加载到寄存器的高 20 位，并让其低 12 位清零
3. mstatus 结构：
31：MPP 先前特权级
30：MPIE 先前中断使能
29：SPV 是否为虚拟模式
28：UBE 用户端大端模式使能
27：XS 扩展状态位
26：FS 浮点状态位
25：MIE 当前机器中断使能
24：SIE 当前监控中断使能
23：UIE 当前用户中断使能
0-22：保留

1. **RISC-V 32 的指令格式**
   - **阅读范围**: 第 2.2 节（Instruction Formats）
2. **LUI 指令的行为**
   - **阅读范围**: 第 2.3.2 节（LUI - Load Upper Immediate）
3. **mstatus 寄存器的结构**
   - **阅读范围**: 第 3.1.4 节（mstatus Register）

6.1.4 shell 命令

pa1 总代码行数：296393 行
命令：find . -name "*.c" -o -name "*.h" | xargs wc -l
通过 git checkout pa0 可以切换到 pa0 最终提交时的状态
pa0 总代码行数：295793 行

编写了大概 600 行代码。
除去空行，pa1 总代码行数：259785
命令：find . -name "*.c" -o -name "*.h" | xargs grep -v '$' | wc -l

6.1.5 RTFM

1. **-Wall**
   - **作用**: 启用大多数编译警告，帮助发现潜在的代码问题，如未使用的变量和可能的逻辑错误。
   - **原因**: 提高代码质量和可维护性，及早识别并解决问题。
2. **-Werror**
   - **作用**: 将所有警告视为错误，任何警告都会导致编译失败。
   - **原因**: 强制开发者解决所有警告，确保代码达到高标准，避免潜在错误。

6.2 想法和心得

感谢王润希同学，葛韩飞同学在我倒数第二个样例 10 次没过时即将走火入魔时候的耐心引导。