实验报告

211240045 杨镇源

实验进度：

PA2.1 完成了所有必做内容；

PA2.2 除 ftrace（以后会逐步完善）外所有必做内容全部完成，并完成了选做内容 Differential Testing；

PA2.3 完成了所有必做内容。

必做题：

必做题 1：YEMU 可以看成是一个简化版的 NEMU，它们的原理是相通的，因此你需要理解 YEMU 是如何执行程序的。具体地，你需要：

（1）画出在 YEMU 上执行的加法程序的状态机

（2）通过 RTFSC 理解 YEMU 如何执行一条指令

（3）思考一下，以上两者有什么联系?

（1）在 YEMU 上执行加法程序的过程中，仅有 PC，r0，r1 和 z 的值会发生改变。在每个指令执行周期中，都要经过取指令、译码并执行指令，更新 PC 三个过程。在此处，我对每个时钟周期拆解为两部分，前一部分为取指令+译码执行，后一部分为更新 PC，用一个向量（PC，r0，r1，M[z]）来记录状态。

(unknown, unknown, unknown, unknown) → (0, y_value, unknown, unknown) → (1, y_value, unknown, unknown) → (1, y_value, y_value, unknown) → (2, y_value, y_value, unknown) → (2, x_value, y_value, unknown) → (3, x_value, y_value, unknown) → (3, x_value+y_value, y_value, unknown) → (4, x_value+y_value, x_value, unknown) → (4, x_value+y_value, y_value, x_value+y_value) → …

最终 halt=1,程序停止运行，此时 z_value=x_value+y_value，实现了该加法程序。

（2）YEMU 执行一条指令分为三个阶段：

第一个阶段为取出指令，对应程序代码段为：

```
this.inst = M[pc]; // 取指
```

第二个阶段为译码执行，对应程序代码段举例为：

```
case 0b0000: { DECODE_R(this); R[rt] = R[rs];   break; }
```

第三个阶段为更新 PC，对应代码段为：

```
pc ++; // 更新 PC
```

即调用 YEMU 中的 exec_once()函数，即可以执行一条指令。YEMU 就是就是这样机械化地执行程序背后的一条一条固定格式的指令，最终将程序“跑”起来。

（3）程序就是一个状态机，计算机执行一条一条指令的过程，从底层来看便是在每个时钟周期到来的时候, 计算机根据当前时序逻辑部件的状态, 在组合逻辑部件的作用下, 计算出并转移到下一时钟周期的新状态。也可以理解为，计算机通过执行程序背后的一条条指令，推动程序状态的转移，最终达到程序的预定效果。

必做题 2：请整理一条指令在 NEMU 中的执行过程。

（1） 调用 engine_start()函数，机器开始运行。

（2） 若载入了可执行程序，则调用 cpu_exec(n)执行预载入程序。

（3） 在 cpu_exec(n)中调用 execute(n)，表示执行 n 条指令。

（4） 在 execute(n)中有一个 n 次的循环，将 exec_once(&s, cpu.pc)调用 n 次，传入一个 Decode 类型的指针 s，方便译码；同时传入 cpu.pc，即指向当前指令存储位置的指针。

（5） 在 exec_once(&s, cpu.pc)中修改 s→ pc、s→ snpc 为 cpu.pc 的值，再调用 isa_exec_once(s)过程，进入 RISC-V 指令集架构，对指令进行译码。

（6） 在 isa_exec_once(s)中首先调用 inst_fetch(&s→ snpc, 4)，获得待译码变量 s 中指令的真值，即 s→ isa.inst.val；同时将 snpc 更新为 snpc+4（即代码中的下一条指令）。

（7） 后调用 decode_exec(s)开始真正的译码过程。

（8） 在 decode_exec(s)中首先将 s→ dnpc 更新为 s→ snpc，即更新为了 pc+4,**后面偏移可能会用到**。然后对 s→ isa.inst.val 进行模式匹配，判断输入的指令的 opcode 是否与表中 opcode 相同。若相同，则找到了对应指令模板。

（9） 此时调用 decode_operand(s, &dest, &src1, &src2, &imm, TYPE_？)函数，计算 rd，rs1,rs2,并将*dest 更新为计算出的目的寄存器的位置。然后根据对应指令类型，选择更新源操作数（*src1 和*src2）和立即数（立即数有不同的扩展类型，需要指令类型确定）。这就完成了译码操作，得到了指令操作的对象。

（10）然后根据 decode_exec（）中的匹配好的指令模式，执行相应的指令执行操作，通过 C 代码模拟指令真正的行为。到此，指令执行结束。

（11）后一路返回，在 isa_exec_once(s)执行完后（即已经执行一条指令后），及时更新 cpu.pc 为当前 s→ dnpc，即下一条指令的位置。（**s→ dnpc 在指令执行过程中可能需要去维护**）

必做题 3：理解打字游戏如何运行。

(1) 首先进行游戏的初始化，通过调用 ioe_init()函数初始化 VGA，Timer，键盘和串口。

(2) 调用 video_init()初始化游戏界面，及字符基本颜色信息。

(3) 当初始化完毕后，nemu 就开始循环播放 VGA 的画面，并通过串口显示出时间，命中数和游戏帧率。

(4)游戏的逻辑实现是通过一帧中数据的值进行处理的，例如字母的下落是在每一帧中通过更改 character->y 的值来控制的。

(5)当按下字母后，我们输入的键盘信息会通过 nemu 中的 Keyboard 设备获取，然后游戏程序通过 Abstract Machine 来获取 nemu 中键盘设备的数据，具体指的是

游戏程序将使用 Abstract Machine 中的接口，接口在 ISA 下转换成对应的 riscv32 机器指令回传给 nemu，在 nemu 中执行指令，进行运算。
(6)随后游戏程序处理相应的逻辑，若命中字符，游戏程序将字符的颜色信息通过 Abstract Machine 中的 API 传给 nemu 中的设备，设备将信息处理后又通过 device_update()判断是否需要更新 VGA，最终将颜色变化通过 VGA 显示在窗口上。

必做题 4： static 与 inline。

单独去掉 static 和 inline 程序都不会发生错误，但是当两者同时去掉则会出现 multiple definition of ‘inst_fetch’的报错。这是由于在链接时候因为有多个同名符号，进行重定位时无法确定地定位 inst_fetch 符号的位置而导致的。

static 静态关键字能让符号存储在静态存储区，函数只能在本文件中调用，以此可将其他文件中的同名符号区分开来。

inline 关键字会让函数不在栈区调用，程序执行时会将其内容直接作为指令执行，和宏的用法类似，因此该函数不被视为一个符号。

必做题 5： 编译与链接

(1) 在 nemu/include/common.h 中添加一行 volatile static int dummy; 然后重新编译 NEMU. 请问重新编译后的 NEMU 含有多少个 dummy 变量的实体? 你是如何得到这个结果的?

答： 一个，通过查看 nemu 的预编译代码得到的。

(2) 添加上题中的代码后, 再在 nemu/include/debug.h 中添加一行 volatile static int dummy; 然后重新编译 NEMU. 请问此时的 NEMU 含有多少个 dummy 变量的实体? 与上题中 dummy 变量实体数目进行比较, 并解释本题的结果。

答： 两个，因为都是静态变量，只在当前文件中能够被调用，且它们都存储在.bss 段（未初始化，是弱符号）中，所以链接时不会发生符号混淆的情况。

(3) 修改添加的代码, 为两处 dummy 变量进行初始化:volatile static int dummy = 0; 然后重新编译 NEMU. 你发现了什么问题? 为什么之前没有出现这样的问题?

答： 程序链接时报错，multiple definition of ‘dummy’，声明重定义。。因为有 volatile 关键词存在，该变量的数值只能通过内存地址调用，再考虑 dummy 变量已进行初始化，它们都储存在.data 段（是强符号）中。由于在同一个数据段中出现了重复的定义，所以报错。

由于在之前的操作中 dummy 并未初始化，所以不会报错。

必做题 6： 了解 Makefile

描述你在 am-kernels/kernels/hello/目录下敲入 make ARCH=$ISA-nemu 后 make 程序如何组织.c 和.h 文件, 最终生成可执行文件

am-kernels/kernels/hello/build/hello-$ISA-nemu.elf。

答：

(1)am-kernels/kernels/hello/目录下的 Makefile 会 include $AM_HOME/目录下的 Makefile，$AM_HOME/目录下的 Makefile 则会 include 对应架构的($ARCH).mk 文件，然后分别 include 硬件架构和软件平台的两份.mk 文件。

(2)在这两份.mk 文件中，为编译添加各种与架构相关的编译选项(即各种 FLAGS)。另外，在 riscv32.mk 文件中确定了交叉编译选项 CROSS_COMPILE=riscv64-linux-gnu。

(3)再在$AM_HOME/目录下的 Makefile 中根据 ARCH，SPLIT 出的 ISA 和 PLATFORM 确定 CFLAGS、CXXFLAGS、ASFLAGS 等编译选项，并在后面确定了源文件和目标文件之间的依赖关系。

(4)根据这些依赖关系和编译选项，调用$(MAKE)，得到编译后的目标文件。

实验心得：

PA2.1 主要检查了我对 RISC-V 架构的掌握，只有在正确理解这一份指令集架构的前提下才能正确实现机器指令的译码，完成 PA2.1 中对不同类型指令的实现。另外在 PA2.1 部分，以代码框架中对不同类型指令的处理方式为例，如何用利用宏的特性写出一份规范、可读性高的代码也很值得学习。

PA2.2 实现了基本的字符串库函数和调试的基础设施。其中在各种 trace 的实现过程中，我熟悉了文件读入读出的写法，并逐渐理解了应该用什么样的方法获得程序动态运行时的踪迹。在实现各种库函数的过程中，我粗略理解了不同库函数的底层实现方式，并再次体会到认真读手册的重要性。

PA2.3 锻炼了我对抽象层之间的关联能力，AM 是一个封装的 API 合集（一个库）又称为运行时环境，是为软件提供调用硬件的一层抽象层，能够调用 NEMU 中的设备。在实现各种设备的过程中，我初步认识到了不同设备的工作原理，并进一步理解了 Abstract-Machine 是如何协同 NEMU 进行工作的，这对理解“程序是如何在计算机上运行”很有好处。