Pa2 实验报告

241300015 秦浩哲

必答题:

1.

NEMU 的核心是有限状态机，执行过程围绕“状态转换”展开，核心逻辑如下:

核心状态:

1. 初始化状态（INIT）: 启动后完成 CPU 寄存器、内存、ELF 加载、设备初始化;

2. 运行状态（RUNNING）: 进入 fetch-decode-execute 循环，持续执行指令（核心工作状态）;

3. 暂停状态（PAUSED）: 响应断点或 `Ctrl+C`，等待用户恢复;

4. 退出状态（EXIT）: 执行退出指令（如 `ecall`）或致命错误（断言失败），终止运行。

状态流转: `INIT → RUNNING ↔ PAUSED → EXIT`，调试功能本质是对状态流转的干预，确保指令执行可控。

2.

一条指令的执行遵循取指-译码-执行-访存-写回五级流程（顺序执行，无流水线并行），核心依赖 `cpu-exec.c`、`ifetch.h`、`decoder.c` 等文件:

1. 取指（Fetch）:

函数: `inst_fetch(addr_t pc, int len)`（`ifetch.h`）;

功能: 根据当前 PC 指针，从模拟内存读取指定长度（如 4 字节 RISC-V 指令）的二进制码，处理内存访问异常。

2. 译码（Decode）:

函数: `decode(uint32_t inst, Decode *dec)`（`decoder.c`）;

功能: 解析指令的 opcode、funct3/funct7、寄存器编号、立即数，结果存入 `Decode` 结构体。

3. 执行（Execute）：

函数：`execute(Decode *dec)`（`cpu/exec.c`）；

功能：按译码结果执行操作（算术逻辑运算、访存地址计算、控制流跳转）。

4. 访存（Memory）：

仅对 `lw`/`sw` 等访存指令生效：加载指令从内存读数据，存储指令向内存写数据。

5. 写回（Write Back）：

功能：将运算结果或加载数据写入目标寄存器（rd），控制流/存储指令无需写回。

核心循环：`cpu.state == RUNNING` 时，重复上述流程，更新 PC 指针（顺序执行 `PC+=4`，跳转则指向目标地址）。

3.

打字小游戏依赖 AM（Application Manager）抽象接口 与 NEMU 交互，运行流程如下：

依赖基础：AM 提供输入（`am_input_key`）、显示（`am_graphic_putchar`）、退出（`am_exit`）接口，NEMU 实现底层模拟外设（键盘、VGA）。

运行流程：

1. 初始化：NEMU 加载游戏 ELF，游戏设置显示、生成目标字符串、初始化得分/计时；

2. 主循环：无限循环执行“等待输入 → 处理输入 → 更新显示 → 判断结束”；

3. 输入处理：通过 `am_input_key` 获取 NEMU 转发的真实键盘输入，比对目标字符串；

4. 显示更新：通过 `am_graphic_putchar` 更新 VGA 缓冲区，实时显示进度、得分；

5. 退出：通关/超时后调用 `am_exit`，NEMU 捕获系统调用并终止运行。

关键链路：`游戏 → AM 接口 → NEMU 模拟外设 → 真实硬件（键盘/屏幕）`，AM 屏蔽底层差异，实现跨平台运行。

4.

`inst_fetch()` 原始定义为 `static inline`，修饰符影响函数的**链接属性**和**编译优化**，不同修改结果如下：

修改方案	编译链接结果	原因分析
去掉 `static`（保留 `inline`）	链接错误（`multiple definition`）	函数变为**外部链接**，多个包含 `ifetch.h` 的 .c 文件生成同名外部符号，链接时冲突
去掉 `inline`（保留 `static`）	编译链接成功	函数仍为**文件内链接**，每个 .c 文件生成独立实例，无符号冲突（仅代码体积增大）
去掉两者（无 `static+inline`）	链接错误（`multiple definition`）	函数为外部链接的普通函数，多个 .c 文件生成同名符号，链接冲突

验证方法：

1. 编译观察错误日志：去掉 `static` 时出现多重定义错误；
2. `nm` 命令查看符号：`static` 函数显示 `t`（文件内链接），无 `static` 显示 `T`（外部链接）；内联函数无符号（已展开）。

5.

1：仅 `common.h` 添加 `volatile static int dummy;`

实体数目：等于包含 `common.h` 的**编译单元（.c 文件）个数**。

原因：`static` 变量为文件内链接，每个包含该头文件的 .c 会生成独立实例（未初始化存于 BSS 段）；`common.h` 是公共头文件，被大量 .c 包含。

验证：`nm nemu | grep dummy` 显示多个 `b dummy`（`b` 表示 BSS 段文件内符

号)。

2：`common.h` + `debug.h` 均添加 `volatile static int dummy;`

实体数目：`包含 common.h 的编译单元数 + 包含 debug.h 但不包含 common.h 的编译单元数`。

与上题对比：数目增加。

原因：两处 `dummy` 是独立文件内链接变量，无关联；若同一 .c 同时包含两个头文件，会生成两个独立实例（同名但作用域隔离)。

3：两处 `dummy` 初始化为 `volatile static int dummy = 0;`

问题现象：编译错误（`redefinition of 'dummy'`)。

原因：

1. 未初始化时是暂定定义，C 允许同一 .c 中多个暂定定义合并为一个实例；
2. 初始化后是完整定义，C 禁止同一 .c 中多个同名完整定义；
3. 同时包含两个头文件的 .c 会出现重复定义，触发错误。

之前无问题的原因：未初始化时为暂定定义，可合并，无冲突。

6. 了解 Makefile：`make ARCH=$ISA-nemu` 生成可执行文件的过程

该过程分为 Makefile 解析 和 编译链接 两部分，最终生成 `hello-$ISA-nemu.elf`：

（1）Makefile 工作方式

1. 变量与包含：

   顶层 `am-kernels/Makefile.inc` 定义 `CC`、`LD`、`CFLAGS` 等全局变量；

   hello 目录 `Makefile` 包含 `../Makefile.inc`，继承变量并定义本地目标。

2. 目标与依赖：默认目标为 `build/hello-$ISA-nemu.elf`，依赖链：`ELF → 目标文件（.o）→ 源文件（.c）+ 链接脚本（linker.ld）`。

3. 隐式规则重写：

   GNU Make 内置 `.c.o` 隐式规则（.c 生成 .o)；

重写该规则，添加架构相关选项（如 `-march=rv32imac`、`-ffreestanding`），适配 NEMU。

4. `-n` 选项：`make -n` 打印所有执行命令（预处理、编译、链接），不实际生成文件，用于调试 Makefile。

（2）编译链接过程

1. 预处理：`$(CC) $(CFLAGS) -E src/main.c -o build/obj/main.i`，展开头文件、替换宏、删除注释；

2. 编译：`$(CC) $(CFLAGS) -c build/obj/main.i -o build/obj/main.o`，将预处理文件编译为目标文件（二进制指令，未解析符号）；

3. 汇编：若有汇编文件（.s），`$(AS) $(ASFLAGS) src/start.s -o build/obj/start.o`，转换为目标文件；

4. 链接：`$(LD) $(LDFLAGS) -T linker.ld build/obj/*.o -o build/hello-$ISA-nemu.elf`，解析符号、合并段（.text/.data/.bss）、分配地址，生成 ELF 可执行文件。

此外，我在实现 PA2 时也遇到了不少困难与挑战，所幸都得以一一解决。

一 1：`printf`输出触发断言失败（最直接报错）

1.1 表现

LiteNES 模拟器调用`printf`时，触发`src/engine/interpreter/hostcall.c:56`断言失败（`Assert fail: 0`），但简单计算、CPU 测试均通过，说明核心执行逻辑正常，仅输入输出模块存在问题。

1.2 背后原因

通过之前的报错分析和代码修改，定位到 3 个关键诱因：

缓冲区溢出：原始`sprint_buf`缓冲区仅 1024 字节，若`printf`输出超长字符串（如模拟器调试日志、游戏批量显示内容），会超出缓冲区长度，破坏内存结构触发断言；

格式化字符串处理不当：未处理`printf(NULL)`（传递 NULL 格式化字符串）、空输出（`printf("")`）等边界场景，断言条件过于严格（如要求`vfprintf`返回值>0，而空输出返回 0）；

缺乏长度限制：使用`vsprintf`（无长度限制）直接写入缓冲区，一旦输出长度超界，会

导致内存越界，触发断言校验。

1.3 解决方法（对应之前的代码修改方案）

增大缓冲区容量：将`sprint_buf`从 1024 字节扩容至 4096 字节，适配更长的输出场景（如模拟器调试信息、游戏多行显示）；

替换安全的输出函数：用`vsnprintf`替代`vsprintf`，强制限制输出长度为缓冲区大小，避免溢出；

补充边界场景处理：对`printf`的 NULL 格式化字符串，自动替换为`"(null)"`（参考标准库行为）；将断言条件从`>0`修正为`>=0`，允许空输出的合法场景；

同步优化辅助函数：增大`snprintf`/`vsnprintf`的临时缓冲区，保持长度一致性，避免中间过程溢出。

二、2：输入输出的底层交互逻辑不清晰（打字小游戏运行）

2.1 表现

不清楚打字小游戏如何获取键盘输入、更新屏幕显示，无法理解“应用程序→模拟器→真实硬件”的联动逻辑，导致输入无响应或显示异常。

2.2 背后原因

缺乏对 AM（Application Manager）接口抽象层的认知：不知道输入输出功能需通过 AM 接口封装，而非直接操作硬件；

不了解 NEMU 的外设模拟机制：不清楚 NEMU 如何捕获真实键盘事件、如何将显示数据映射到 VGA 缓冲区。

2.3 解决方法（对应打字小游戏运行机制的探究）

理解“三层交互链路：明确输入输出的核心流转逻辑：`应用程序（游戏）→ AM 接口 → NEMU 模拟外设 → 真实硬件（键盘/屏幕）`；

调用标准 AM 接口：输入依赖`am_input_key`（获取 NEMU 转发的真实键盘事件），显示依赖`am_graphic_putchar`（更新 VGA 显示缓冲区），退出依赖`am_exit`（触发模拟器终止）；

依赖 NEMU 的底层实现：无需关注 NEMU 如何模拟键盘中断、VGA 映射，只需调用 AM 提供的抽象接口，即可实现跨模拟器/硬件的输入输出功能。

三、3：编译链接阶段的输入输出相关冲突

3.1 表现

修改`inst_fetch()`函数的`static`/`inline`修饰符后，出现`multiple definition`（多重定义）链接错误；

在输入输出相关头文件（如`common.h`/`debug.h`）中添加静态变量后，编译失败或出现变量实体冗余。

3.2 背后原因

对`static`（链接属性）和`inline`（编译优化）的作用理解不足：`static`确保函数/变量仅当前文件可见（文件内链接），`inline`建议编译器内联展开，去掉`static`后函数变为外部链接，导致多文件包含时符号冲突；

混淆 C 语言中“暂定定义”和“完整定义”：头文件中未初始化的`static`变量（`volatile static int dummy;`）是暂定定义，可合并；初始化后（`=0`）变为完整定义，多文件包含时触发重复定义。

3.3 解决方法（对应编译链接的实验探究）

正确使用`static`和`inline`修饰符：

头文件中的工具函数（如`inst_fetch()`）必须保留`static`，避免多文件包含时外部符号冲突；

保留`inline`以减少函数调用开销（输入输出函数频繁调用，内联可提升性能）；

规范头文件静态变量的使用：

避免在公共头文件（如`common.h`）中定义初始化的静态变量，若需使用，需确保仅被单个编译单元包含；

未初始化的静态变量（暂定定义）可在多文件中包含，但需注意实体冗余（不影响功能，仅增大代码体积）。

四、4：输入输出的边界场景未覆盖

4.1 表现

除了`printf(NULL)`和空输出，还可能遇到`printf`参数不匹配（如`printf("%d")`少传参数）、非法内存地址访问（如格式化字符串指向非法地址）等场景，导致程序崩溃或断

言失败。

4.2 背后原因

输入输出模块的“防御性编程”不足：未对参数合法性、内存指针有效性进行校验；

断言条件设计过于严格：将合法的边界场景（如参数个数不匹配、非法指针）直接触发断言，而非优雅处理。

4.3 解决方法（对应断言失败的根源分析）

补充参数校验逻辑：在`hostcall.c`的`printf`处理逻辑中，解析格式化字符串的参数个数，与实际传递的参数个数比对，不匹配时输出警告并返回，避免断言；

添加内存指针合法性校验：对格式化字符串指针，先判断是否在应用程序的合法地址空间内，非法则替换为`"(invalid ptr)"`并警告，而非直接触发断言；

优化断言的使用场景：仅在“致命错误”（如内存分配失败、硬件模拟异常）时使用断言，边界场景采用“警告+降级处理”，提升程序健壮性。

这些困难的解决，本质上是从“功能实现”到“健壮性+可移植性”的升级过程——不仅要让 PA2“能用”，还要应对各种边界场景、适配编译链接规则，最终实现稳定可靠的交互功能。