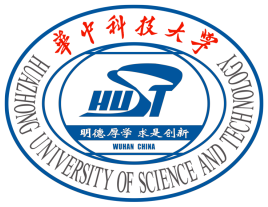

華中科技大學

课 程 实 验 报 告

课程名称：系 统 能 力 培 养

姓　　名：张 丘 洋

学　　号：U201614667

院　　系：计算机科学与技术学院

专业班级：计算机科学与技术 1606 班

指导教师：谭 志 虎

分数	
教师签名	

2019 年 11 月 1 日

目录

1

开发环境

1.1 硬件配置

1.2 软件配置

- gcc 8.3.0
- nvim 0.3.1
- ld 2.32.0
- python 3.6.9

2

硬件平台—NEMU

2.1 前言

编写虚拟机的第一个任务就是去实现其硬件设施。根据冯诺伊曼计算机的思想，一个完整的计算设备需要有运算器、控制器、存储器、输入设备和输出设备，而本章介绍的 NEMU 就是去实现这五大部件。

在今年，南京大学的 NEMU 项目进行了扩展，使 NEMU 提供了三种指令集架构可供选择。一个指令集架构约定了指令的编码方式以及运算器和控制器的解码及执行方式。在 NEMU 中提供了三种指令集架构，分别是：`x86`，`mips32` 和 `riscv32`。在这里我选择的是日常使用中最常见的 `x86` 架构。

2.2 系统设计与实现

下面我们就正式进入冯诺伊曼机 NEMU 的设计与实现。由于 NEMU 源文件较多，为了能够更好的叙述设计和实现，这里采用自顶向下的方式来阐述。

2.2.1 NEMU 总体架构

由于 NEMU 模拟器是一个冯诺伊曼机，所以其整体架构遵循冯诺伊曼机。NEMU 的总体架构图如图 2.1 所示。

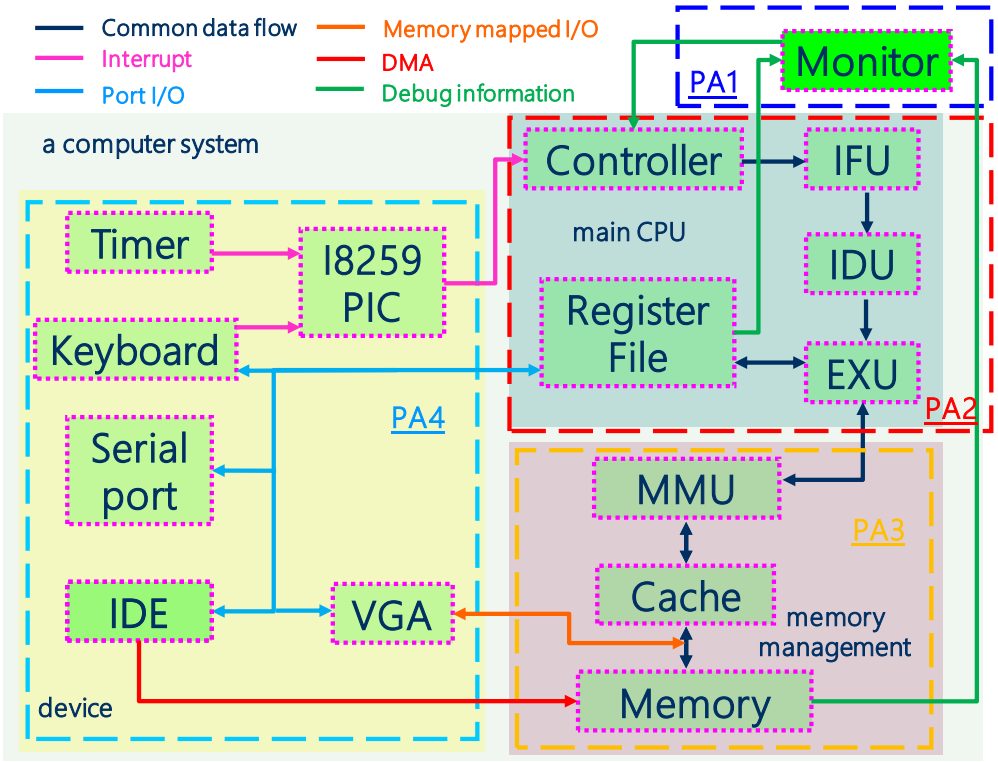

图 2.1: NEMU 总体架构图

2.2.2 框架代码结构

```
nemu/
├── include/
│   ├── cpu/
│   ├── device/
│   ├── memory/ ..... 内存访问有关
│   ├── monitor/ ..... 监视器有关
│   ├── rtl/ ..... 通用 rtl 指令定义
│   ├── common.h ..... 公用头文件
│   ├── debug.h
│   ├── macro.h
│   └── nemu.h
├── src/
│   ├── cpu/ ..... CPU 执行有关
│   ├── device/ ..... IO 设备实现
│   ├── isa/ ..... 指令集架构封装
│   │   ├── mips32/ ..... mips32 指令集
│   │   ├── riscv32/ ..... riscv32 指令集
│   │   └── x86/ ..... x86 指令集
│   ├── memory/ ..... 内存访问实现
│   ├── monitor/
│   │   ├── debug/ ..... 调试器实现
│   │   │   ├── expr/ ..... 表达式解析 (这里与给定的框架代码不同，是个人修改的)
│   │   │   │   ├── def.h ..... 表达式解析有关函数定义
│   │   │   │   ├── lex.l ..... 表达式解析词法规则定义
│   │   │   │   └── parser.y ..... 表达式解析语法规则定义
│   │   │   ├── log.c ..... Log 信息输出
│   │   │   ├── ui.c ..... 监视器交互命令实现
│   │   │   └── watchpoint.c ..... 监视点实现
│   │   ├── diff-test/
│   │   ├── cpu-exec.c
│   │   └── monitor.c
│   └── main.c
├── tools/ ..... 测试及调试用工具
└── Makefile
```

```
├── Makefile.git ..... git 版本控制相关
└── runall.sh ..... 一键测试脚本
```

2.2.3 NEMU 执行流

为了能够了解 NEMU 的工作方式，我们来看看 NEMU 整体的一个执行流程。

进入 `nemu/src/main.c` 文件，能够看到里面定义了 `main()` 函数。在 `main()` 函数中只有两行，第一行调用 `init_monitor()` 函数对 NEMU 进行各项初始化，并根据调用参数来判断本次程序运行是否是批处理模式。第二行调用 `ui_mainloop()` 函数。`ui_mainloop()` 函数在 `nemu/src/monitor/debug/ui.c` 中定义，该函数是监视器与用户进行 IO 交互的主函数。在该函数中首先判断程序是否是批处理模式，若是批处理模式则直接运行在命令行中指定的程序，不会出现与用户的交互界面。若不是批处理模式则会进行循环，在循环体中首先等待用户的命令，之后根据用户所输入的命令调用相应的处理函数。

由此我们可以得到 NEMU 的总体流程图，如图 2.2 所示。

在 NEMU 的交互界面中，一共设定了九个命令，这些命令对应的字符串以及其含义如表 2.1 所示。

表 2.1: NEMU 调试器指令

命令名称	命令描述	处理函数
help	显示帮助	`cmd_help()`
c	继续运行程序	`cmd_c()`
q	退出 NEMU	`cmd_q()`
p expr	打印表达式值	`cmd_p()`
info r	打印寄存器信息	`cmd_info()`
info w	打印监视点信息	`cmd_info()`
w	添加监视点	`cmd_w()`
d	删除监视点	`cmd_d()`
s	单步执行	`cmd_s()`

在这些命令中，最重要的就是 `c` 命令，该命令使程序继续运行，在没有设置监视点的情况下输入 `c` 命令会使程序一直执行到结束为止。下面我们就来看看 `c` 命令的实现。

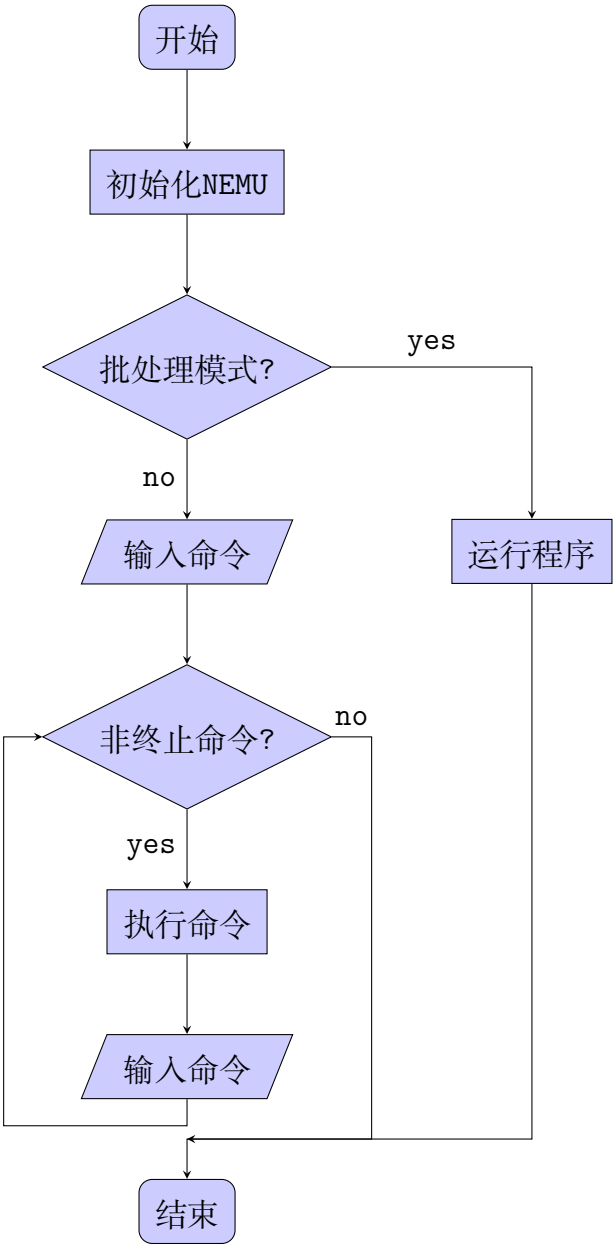

图 2.2: NEMU 整体流程图

2.2.4 NEMU 内程序运行流程

c 命令的执行函数是 cmd_c(),该函数在 nemu/src/monitor/debug/ui.c 中定义,该函数的函数体只有一行，即 cpu_exec(-1)。这个函数在文件 nemu/src/monitor/cpu-exec.c 中定义，该函数有一个参数，这个参数代表了要执行的虚拟机程序的指令条数。在 cmd_c() 中传入的参数值是-1，由于参数值是按照无符号数来解析的，所以-1 会被认为是最大的整数，也就是说想让程序一直执行直到程序退出为止。

nemu/src/monitor/cpu-exec.c

虽然一般情况下程序的指令数量都不会超过最大整数的值，但如果超过了那么就会发生错误。为了解决这个可能会出现的问题，我们可以在 cpu-exec() 函数中做一点修改，只要让当参数值为最大整数 (也就是-1) 时一直运行即可，可以把 for 循环中更新部分的 n--变为 (n == (uint64_t)-1) ? n : n --。

继续观察 cpu-exec() 函数，该函数使用了状态机来判断虚拟机的运行状态，我们可以先不管。函数内主体是一个 for 循环，在循环体内首先记录下旧的 pc 值，之后调用 exec_once() 函数，从这个函数的名字和出现的位置我们可以推断出来这个函数的功能是让虚拟机执行下一条指令。执行了一条指令之后就继续进入下一个循环 (可以先忽略掉条件编译部分)。

接下来我们就需要去看看 exec_once() 函数了。这个函数在文件 nemu/src/cpu/cpu.c 中定义，函数体也比较简单，先调用 isa_exec() 函数再调用了 update_pc() 函数。由于 NEMU 代码框架中定义了三种指令集架构，对于每一种架构来说其执行指令的过程都是不同的，所以能够看出来 isa_exec() 函数的存在是为了屏蔽掉不同架构带来的差异，相当于一个接口，不同的架构只需要实现这个接口就可以达到架构无关的目的。update_pc() 函数从名字上就能看出来是更新指令计数器 pc 的值。

下面我们就要进入到 isa_exec() 函数内看看了。由于这个接口对于不同的架构来说实现是不同的，因为本次选择的架构是 x86，所以我们就要去看 x86 下的实现。打开文件 nemu/src/isa/x86/exec/exec.c，在文件的末尾定义了 isa_exec() 函数。在该函数体中，首先通过 instr_fetch() 函数取出一个字节的 opcode，并设置好 decinfo 全局译码结构相关的成员，之后调用了一次 set_width() 函数，最后调用 idex() 函数。

看到这里你可能会感觉有点懵，按理来说让 CPU 执行一条指令应该先从 pc 处取出一条指令，之后对这个指令进行解码执行即可，为什么这里取出指令的长度只有一个字节? 这里的 set_width() 函数又是有什么作用呢? 要解决这个问题就要涉及到 x86 架构指令的编码方式了。

3

运行时环境—AM

4

操作系统—nanos-lite