

华中科技大学

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 12 月 12 日

目录

1	开发环境	1
1.1	硬件配置	2
1.2	软件配置	2
2	硬件平台—NEMU	3
2.1	前言	4
2.2	系统设计与实现	4
3	运行时环境—AM	15
4	操作系统—nanos-lite	17
A	必答题	19
A.1	PA1	20
A.2	PA2	22
A.3	PA3	24

开发环境

1.1 硬件配置

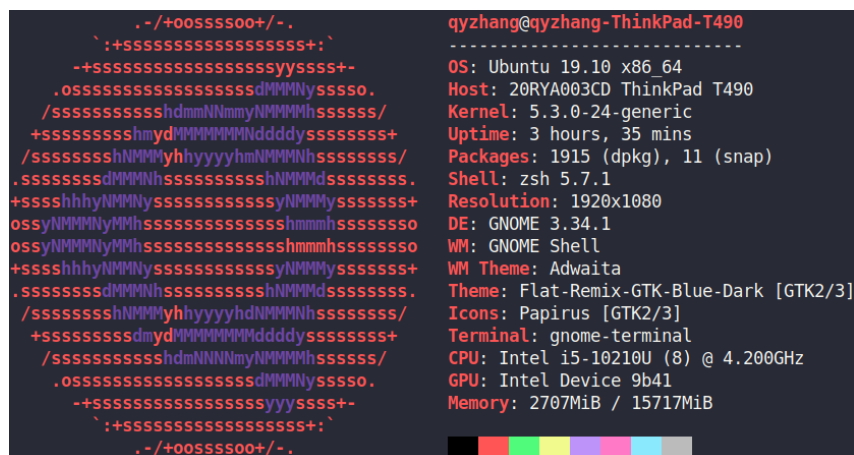


图 1.1: neofetch 命令输出

1.2 软件配置

- gcc 8.3.0
- nvim 0.3.8
- python 3.7.5

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。

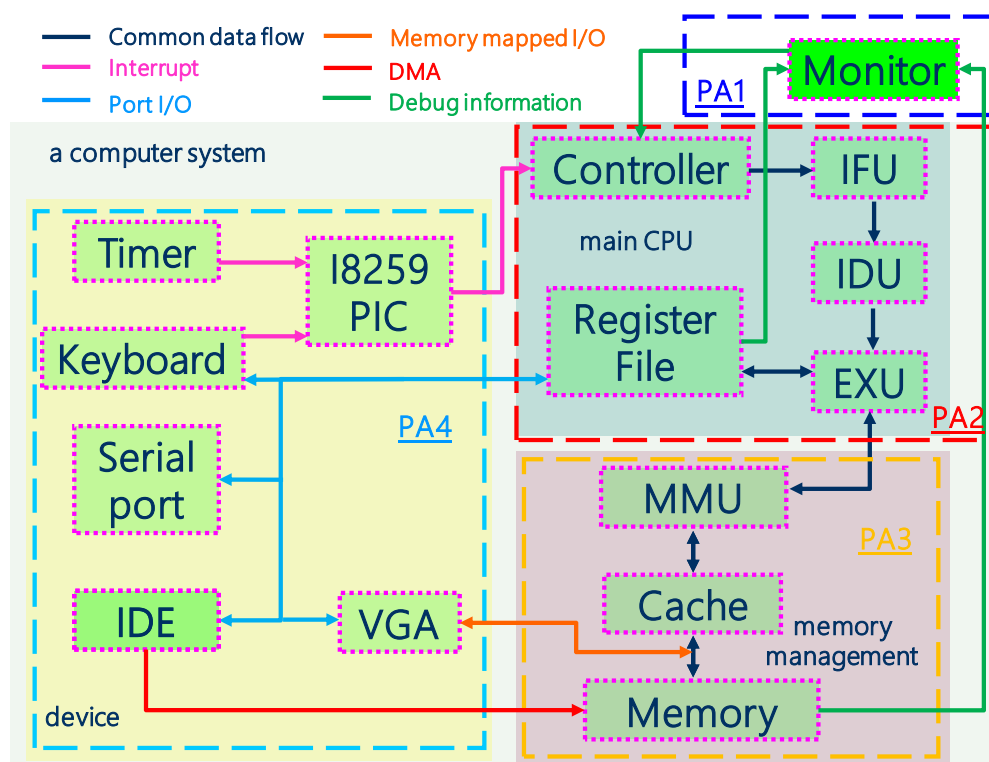


图 2.1: NEMU 总体架构图

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)
│   │   │   │   ├── 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。

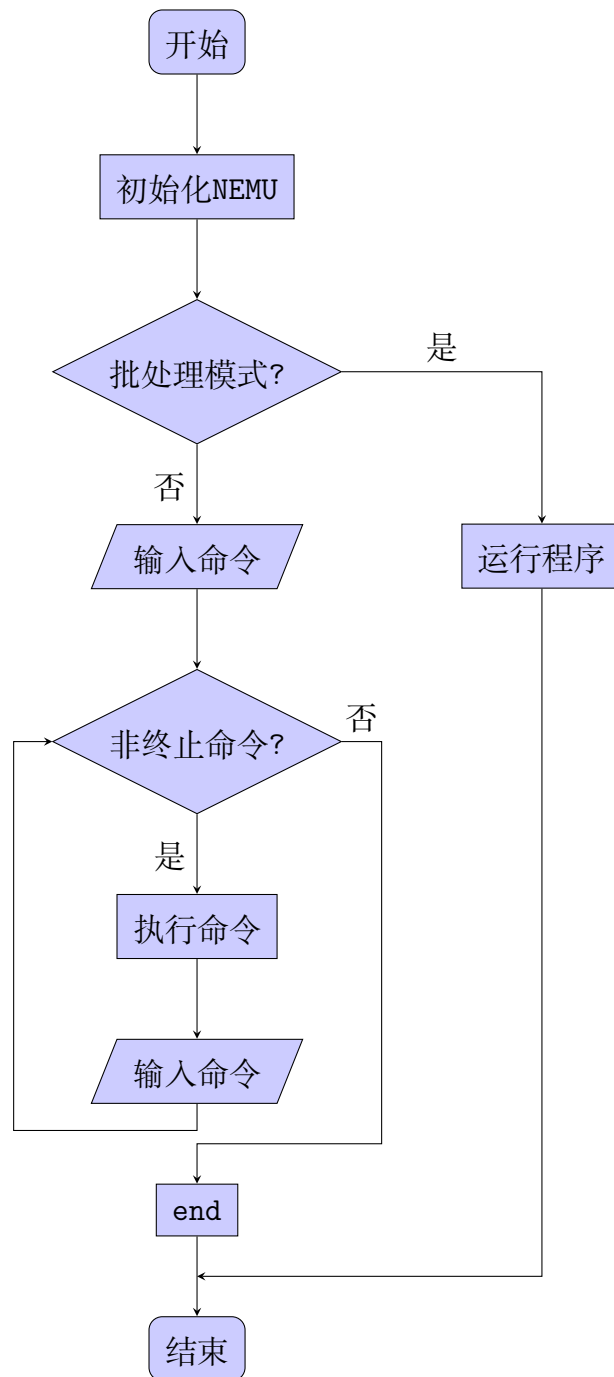


图 2.2: NEMU 整体流程图

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。

2.2.5 x86 架构指令编码

现在我们已经根据 c 命令的执行过程来到了 x86 架构的指令执行函数，为了继续进行我们需要知道 x86 架构指令的编码方式。

x86 的指令编码格式如图 2.3 所示，每个域的作用如表 2.2 所示。

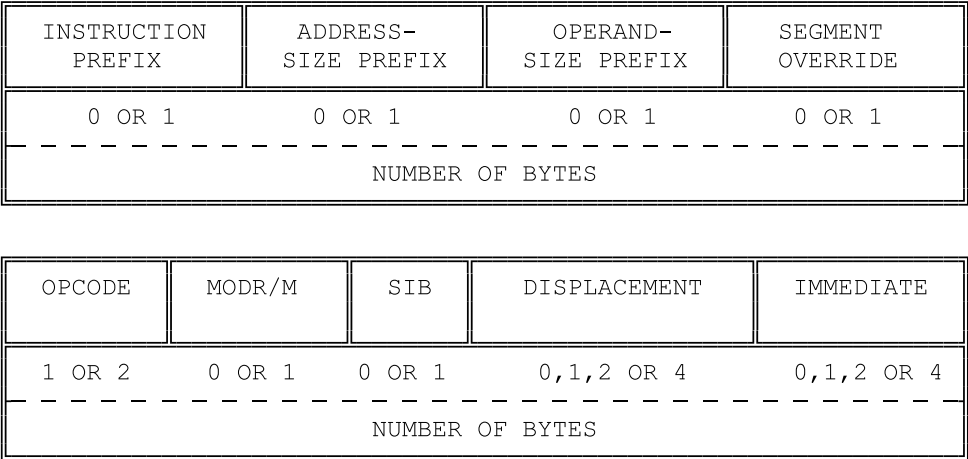


图 2.3: x86 指令编码格式

表 2.2: x86 指令域及其作用

域名	作用	字节数
instruction prefix	改变指令行为 (如 REP)	0 或 1
address-size prefix	改变地址位宽	0 或 1
operand-size prefix	改变操作数大小	0 或 1
segment override	改变寻址使用的段寄存器	0 或 1
opcode	指令操作码	1 或 2
modr/m	指定寻址方式	0 或 1
sib		0 或 1
displacement	内存寻址偏移	0, 1, 2 或 4
immediate	立即数	0, 1, 2 或 4

能够看到在编码格式中包含了很多部分，并且前面有很多部分都是可有可无的。

我们先从简单的来说起。在这些域中，必须存在的就是 opcode 部分。opcode 可以理解为 operation code，也就是操作码，它代表了这条指令的功能，可以说是指令的 id，不同的指令其 opcode 是不同的。举一个最简单的例子，对于 x86 指令 PUSH EAX 来说，该指令仅有一个字节也就是 opcode，为 0x50。你可能会问，opcode 域不是可以有两个

字节吗？我们怎么在译码的时候知道这个 opcode 是一个字节还是两个字节呢？事实上，由于一个字节只有 256 个值，而对于 x86 这样的 CISC 架构是不够的，为了能够拥有更多的指令，x86 采用了变长编码，将 0x0F 作为转义字节，也就是说当第一个 opcode 是 0x0F 是就代表这个 opcode 有两位，类似于赫夫曼编码。比如，SET0 指令是当 OF 标志位为 1 时将对应的操作符设为 1，这个指令的 opcode 是 0x0F 0x90。

之后我们来看看 opcode 之前的域。opcode 之前的域都是可有可无的，当这些域中的某一个存在时，就会改变这条指令的行为。比如如果存在 operand-size prefix 域，那么原本一条指令操作数的位宽就会从 32 位变为 16 位。这样就会有一个问题，当我们进行译码时，怎么知道这些域存不存在呢？事实上 opcode 前面的这些域与 opcode 包含的所有编码都是两两不同的，比如 operand-size prefix 域要是存在的话只能是 0x66，而 opcode 域的编码并不包含 0x66，所以在译码时就能够根据当前这个字节的值来判断这个字节是代表着 opcode 域还是其之前的某个域。上面指出了 PUSH EAX 指令的编码为 0x50，而我们又知道 0x66 代表着 operand-size prefix 域，所以编码 0x66 0x50 就代表着 PUSH AX，即将操作数从 32 位的 EAX 变为了 16 位的 AX。

在 opcode 之后是 modr/m 和 sib 域。这两个域可以说是 x86 编码的精华，在这两个域中包含了以下信息：

- 寻址方式或是要使用的寄存器
- 需要用到的寄存器，或是指定指令行为的信息
- 基址、变址以及比例因子信息

总的来说，这两个域说明了这个指令要怎样去找到操作数。比如对于 MOV 指令，这个指令的功能是将数据从一个地方复制到另一个地方，操作数可能在两个寄存器中，可能都在内存中，也可能一个在寄存器一个在内存中。modr/m 和 sib 域就指定了这条 MOV 指令的两个操作数到底在哪，如果在寄存器中会说明在哪个寄存器，如果是在内存中会说明内存的寻址方式以及寻址的基址、变址寄存器等信息。

modr/m 和 sib 各自都有 8 位，他们各自又将这 8 位划分成了三个部分，如图 2.4 所示。

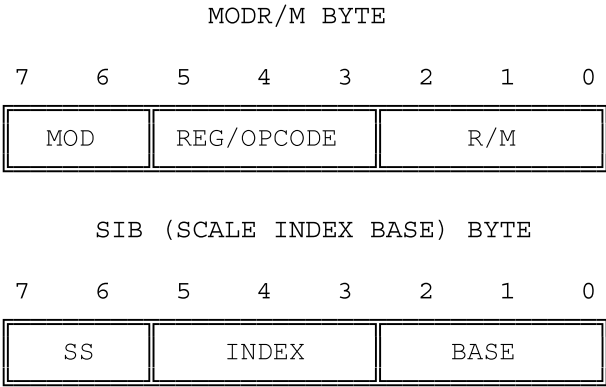


图 2.4: modr/m 和 sib 域的划分

对于 `modr/m` 和 `sib` 的具体编码详见 [?]

最后还有 `displacement` 和 `immediate` 域, 这两个域比较简单, `displacement` 域代表了内存寻址时的偏移, 要和 `modr/m` 以及 `sib` 配合使用。而 `immediate` 域则是代表了立即数, 是否存在该域要看具体的 `opcode`。

2.2.6 x86 指令译码与执行设计

在大致知道了 x86 指令的编码方式之后, 我们就能够进行译码的实现。再次回到 `isa_exec()` 函数, 在函数体中首先在 `pc` 处取出一个字节, 这时我们大概能够推断出取出的这一个字节是 `opcode`(当然也有可能是 `opcode` 前面的那些域, 我们后面再来说明如果是前缀域要怎么办), 取出了 `opcode` 之后调用了一次 `set_width()` 函数, 从函数名字来看这个函数是设置宽度, 能够推断出这个函数的大概作用是设置操作数的宽度, 我们先忽略掉, 之后就调用了 `idex()` 函数。`idex()` 在名字上能够看出来是 `id` 和 `ex`, 其中 `id` 指的是 `instruction decode`, 即指令译码, `ex` 指的是 `execute`, 也就是指令执行。那么 `idex()` 的作用我们能推断出是译码以及执行的意思了。

虽然大概知道了 `isa_exec()` 函数的执行流程, 但还是很不清晰, 还是不了解 `set_width()` 的具体实现, 以及译码时 `opcode` 之前的域和之后的域都是怎么推断出来的。

为了对译码部分的实现有更清晰的认识, 我们进入到 `idex()` 里面看看其是怎么执行的。`idex()` 函数在 `nemu/include/cpu/exec.h` 中定义, 函数也非常简单, 有两个参数, 一个是 `pc` 我们很好理解, 另一个是 `OpcodeEntry` 结构指针。在函数体内先判断 `OpcodeEntry` 是否存在一个名为 `decode` 的函数指针, 如果存在则调用该函数。之后调用 `OpcodeEntry` 下名为 `execute()` 的函数。也就是说, 在 `idex()` 函数中只是简单地调用了这个指令对应的 `decode()` 函数以及 `execute()` 函数。这里比较重要的是 `OpcodeEntry` 这个参数, 能够看到这个参数中包含了对应指令的译码及运行的实现函数指针。我们回到 `isa_exec()` 函数, 发现这个 `OpcodeEntry` 参数传入的值是 `&opcode_table[opcode]`。那么我们就看看 `opcode_table` 是一个什么数组。进入文件 `nemu/src/isa/x86/exec/exec.c`, 能够看到在这个文件里用很长的代码定义了静态数组 `opcode_table`, 这个数组的类型就是 `OpcodeEntry`。而 `OpcodeEntry` 结构在 `nemu/include/cpu/exec.h` 中定义, 其定义为:

```
1 typedef void (*EHelper) (vaddr_t *);
2 typedef void (*DHelper) (vaddr_t *);
3 typedef struct {
4     DHelper decode;
5     EHelper execute;
6     int width;
7 } OpcodeEntry;
```

能够看到该结构只有三个成员，一个是 `decode` 函数指针，一个是 `execute` 函数指针，另一个是 `width`。根据 `OpcodeEntry` 的名字以及其成员和出现的位置我们能够知道，这个结构的含义是指令译码和执行的入口。由于 x86 指令是变长编码，不同指令有不同的宽度，并且不同的指令有不同的功能，所以我们在对指令进行译码和执行时就不能一概而论，必须要根据具体指令来选择其对应的译码和执行的实现，而区分指令则是根据 `opcode` 域，所以当我们得到了一条指令的 `opcode` 之后就需要转入该 `opcode` 对应指令的译码和执行函数。`opcode_table` 就是保存所有 `opcode` 对应实现函数的表，这个表一共有 512 个成员，前 256 个代表了 `opcode` 长度为 1 的那些指令，后 256 个代表了 `opcode` 长度为 2 的那些指令。

由于每个 `opcode` 对应的实现是已知的，所以 `opcode_table` 表是静态的，应在编译时就设置好其成员，也就是说需要在代码文件中填写好。为了方便填写 `opcode_table` 表，在 `nemu/include/cpu/exec.h` 中定义了一些宏，如下所示。

```
1 #define IDEXW(id, ex, w) {concat(decode_, id), concat(exec_, ex), w}
2 #define IDEX(id, ex)    IDEXW(id, ex, 0)
3 #define EXW(ex, w)      {NULL, concat(exec_, ex), w}
4 #define EX(ex)          EXW(ex, 0)
5 #define EMPTY           EX(inv)
```

这些宏最主要的一个就是 `IDEXW`，这个宏有三个参数，其中 `id` 代表了译码函数名，`ex` 代表了执行函数名，`w` 代表了 `OpcodeEntry` 中的 `width` 成员。`IDEX` 宏与其类似，只不过将 `width` 成员设为 0。`EXW` 宏是将译码实现函数设为 `NULL`，`EX` 宏是仅设置执行函数，`EMPTY` 宏的定义是 `EX(inv)`，而 `inv` 代表的是 `invalid`，也就是说 `EMPTY` 代表这个指令还未实现。在刚开始时，表项中大多数都是 `EMPTY`，也就是还未实现，需要自己去增加。

看到这可能还会有一个问题，为什么有些指令的宽度和译码函数没有呢？指令中的宽度代表的是什么意思呢？我们先说为什么有些指令的译码函数没有。由于 x86 是变长指令，所以不同指令的长度不同，从指令的格式上来看 `opcode` 域之后的那些（比如 `modr/m` 和 `sib`）都是可有可无的，并且当他们存在时字节数也不一定。其实 `opcode` 域之后的那些域都是由 `opcode` 决定的。`opcode` 决定了具体的指令，而具体的指令又能决定操作数的信息，而 `opcode` 后面的那些域正是代表着操作数的信息，所以当 `opcode` 知道了之后就知道是否需要后面的某些域来确定操作数信息。那么 `OpcodeEntry` 中的 `width` 成员代表着什么意思呢？这个成员代表的是指令操作数的长度，有些 `opcode` 决定了这条指令操作数的长度，这时我们就需要在 `OpcodeEntry` 中设置，而有些则并不确定。还记得指令格式中的 `operand-size prefix` 域吗？有些指令靠他来确定操作数的位宽，当没有这个前缀时位宽是 32，当存在时位宽是 16。需要知道指令操作数位宽的原因是在我们取操作数时需要知道，比如在内存中取操作数时，操作数字节数是 0、1、2 或 4 时取出来的值是不同的（当然在极个别情况下可能会相同）。这时我们就能够解释

`isa_exec()` 函数中为什么要调用 `set_width()` 函数了。`set_width()` 函数就是根据是否存在 `operand-size prefix` 域来设置操作数的位宽到底是 32 还是 16。

最后还有一个问题，那就是 `opcode` 之前的域到底是怎么处理的？我们在前面假设指令取出的第一个字节就是 `opcode`，但如果在 `opcode` 之前有其他的域，比如 `operand-size prefix` 域，那要怎么办呢？事实上，由于前面提到 `opcode` 的编码与 `opcode` 之前的域的编码是两两不同的，所以在 NEMU 中也将 `opcode` 之前的域当作 `opcode` 处理了，他们在 `opcode_table` 中也存在着表项，并且这个表项中只包含执行函数。而在其执行函数中，会在 `decinfo` 这个全局译码信息中记录下这个前缀域要改变的指令行为，之后会再次调用 `isa_exec()` 函数。注意这里的调用并不代表是进入到下一个指令的执行，而是重新开始这一条指令的译码及执行。由于在前缀域的执行函数中已经记录下了这次执行需要改变的行为，所以在真正指令译码和执行的过程中就可以通过 `decinfo` 来判断是否需要改变某些行为，比如当存在 `operand-size prefix` 时译码过程中在取操作数时会将操作数的位宽变为 16。

由于 NEMU 比较简单，所以在所有的指令中只会出现 `operand-size prefix` 这一种前缀域，该域的执行函数在 `nemu/src/isa/x86/exec/prefix.c` 中定义，如下所示。

```
1 // 先忽略掉 make_EHelper, 这里主要关注函数体
2 make_EHelper(operand_size) {
3     decinfo.isa.is_operand_size_16 = true;
4     isa_exec(pc);
5     decinfo.isa.is_operand_size_16 = false;
6 }
```

好了，现在总结一下，在 NEMU 中，x86 架构在执行一条指令时，会先在 `pc` 处取出一个字节，这个字节就是这一条指令的 `opcode` (前面说了如果是前缀域则也会按照 `opcode` 一样处理)，然后以 `opcode` 作为下标从 `opcode_table` 中取出这一条指令对应的入口，之后如果入口中定义了译码函数则会调用译码函数，在译码函数中会根据这一条指令的编码规定去取出 `opcode` 后面的域，并且根据这些域以及指令的内容去取出操作数，并设置目的操作数的信息。之后会调用入口中的执行函数，对译码阶段中设置的源操作数进行运算，并将结果放入目的操作数中。

x86 译码和执行的流程图如图 2.5 所示。**注意：**在流程图中将前缀域的译码单独拉出来了，但是在实现时要记得是当作 `opcode` 一样处理的。

2.2.7 x86 指令译码实现

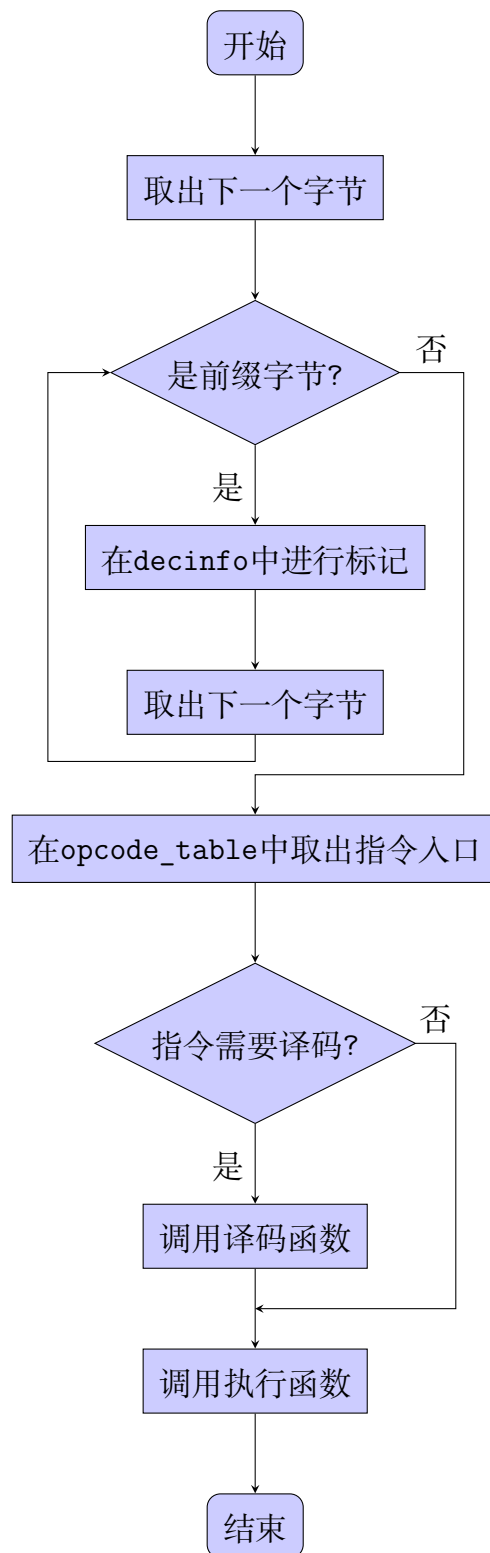


图 2.5: x86 译码与执行流程图

运行时环境—AM

操作系统—nanos-lite

附录 A

必答题

A.1 PA1

问：

理解基础设施。我们通过一些简单的计算来体会简易调试器的作用。首先作以下假设：

- 假设你需要编译 500 次 NEMU 才能完成 PA
 - 假设这 500 次编译当中, 有 90% 的次数是用于调试.
 - 假设你没有实现简易调试器, 只能通过 GDB 对运行在 NEMU 上的客户程序进行调试. 在每一次调试中, 由于 GDB 不能直接观测客户程序, 你需要花费 30 秒的时间来从 GDB 中获取并分析一个信息.
 - 假设你需要获取并分析 20 个信息才能排除一个 bug.
1. 那么这个学期下来, 你将会在调试上花费多少时间?
 2. 由于简易调试器可以直接观测客户程序, 假设通过简易调试器只需要花费 10 秒的时间从中获取并分析相同的信息. 那么这个学期下来, 简易调试器可以帮助你节省多少调试的时间?

答：

1. 根据以上假设, 经过以下计算公式可以得到答案:

总调试时间 = 编译次数 × 调试比例 × 每次调试需要获得的信息 × 每个信息获取时间

即：

$$\text{总调试时间} = 500 \times 90\% \times 20 \times 30 = 270000s = 75h$$

2. 根据该假设获取每个信息的时间从 30s 变为了 10s, 则总的时间变成了原来的三分之一, 于是有:

$$\text{总调试时间} = \frac{75}{3}h = 25h$$

问：

查阅 i386 手册。理解了科学查阅手册的方法之后, 请你尝试在 i386 手册中查阅以下问题所在的位置, 把需要阅读的范围写到你的实验报告里面:

1. EFLAGS 寄存器中的 CF 位是什么意思?
2. ModR/M 字节是什么?
3. mov 指令的具体格式是怎么样的?

答：

1. CF 位包含了无符号数加减法的进位信息, 可在 i386 手册的 3.2 节和 2.3.4 节找到.
2. ModR/M 字节包含了指令的寻址等信息, 具体的内容可见报告的 2.2.6 一节. 在 i386 手册中可在 17.2.1 一节中找到详细的描述.

3. mov 指令有很多种变形, 若操作数为寄存器则是一个字节的 opcode 跟上一个字节的 modr/m; 若有一个操作数是立即数则是一个字节的 opcode 跟上若干字节的立即数, 立即数的位数由 opcode 决定。

问:

shell 命令。完成 PA1 的内容之后, nemu/目录下的所有.c 和.h 文件总共有多少行代码? 你是使用什么命令得到这个结果的? 和框架代码相比, 你在 PA1 中编写了多少行代码? (Hint: 目前 pa1 分支中记录的正好是做 PA1 之前的状态, 思考一下应该如何回到“过去”?) 你可以把这条命令写入 Makefile 中, 随着实验进度的推进, 你可以很方便地统计工程的代码行数, 例如敲入 make count 就会自动运行统计代码行数的命令。再来个难一点的, 除去空行之外, nemu/目录下的所有.c 和.h 文件总共有多少行代码?

答:

1. 当不考虑空行时, 可以使用以下命令来统计:

```
1 find . -not -regex '.*riscv32.*\|.*mips32.*\|.*build.*\|.*tools.*' | grep  
   ↪ '\.c$\|\.h$\|\.y$\|\.l$' | xargs wc -l
```

这条命令首先使用 find 命令来查找 nemu/下的所有文件, 并且使用正则表达式去掉了所有包含 riscv32、mips32、build 或 tools 的文件, 之后使用 grep 命令筛选出后缀为.c、.h、.l、.y 的文件, 最后使用 wc 命令统计。统计结果为 5187 行。**注意:** 由于之前编写时没有按阶段保存, 这里统计的代码是完成 PA4 之后的。

2. 当考虑空行时, 需要在获取到文件之后使用 cat 命令读出所有文件的内容, 之后使用 sed 流编辑器去除空行, 最后再用 wc 来统计。该命令如下所示:

```
1 find . -not -regex '.*riscv32.*\|.*mips32.*\|.*build.*\|.*tools.*' | grep  
   ↪ '\.c$\|\.h$\|\.y$\|\.l$' | xargs cat | sed '/^\s*$/d' | wc -l
```

统计结果为 4378 行。**注意:** 由于之前编写时没有按阶段保存, 这里统计的代码是完成 PA4 之后的。

问:

使用 man。打开工程目录下的 Makefile 文件, 你会在 CFLAGS 变量中看到 gcc 的一些编译选项。请解释 gcc 中的-Wall 和-Werror 有什么作用? 为什么要使用-Wall 和-Werror?

答:

-Wall 会在编译时显示所有的 Warning, -Werror 会把所有的 Warning 变成 error。加入这两个编译选项是尽量在编译时显示出代码中的错误, 这样会避免一些简单的错误。

A.2 PA2

问：

编译与链接。在 `nemu/include/rtl/rtl.h` 中, 你会看到由 `static inline` 开头定义的各种 RTL 指令函数. 选择其中一个函数, 分别尝试去掉 `static`, 去掉 `inline` 或去掉两者, 然后重新进行编译, 你可能会看到发生错误. 请分别解释为什么这些错误会发生/不发生? 你有办法证明你的想法吗?

答：

对于这些函数, 当同时去掉 `static` 和 `inline` 时会发生链接错误, 这是因为这些函数定义在头文件中, 若两个修饰符都没有的话每次包含该头文件时都会将该函数定义一边, 这样就造成了重定义。`static` 代表该函数只能在该文件中使用, 所以带上该修饰符时不会造成重定义。而 `inline` 修饰符有时候编译器会将函数变为类似于宏来处理, 也不会发生重定义。

问：

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?

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.

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：

1. 有一个, 因为这里只在该文件中使用 `static` 修饰符定义了一次。

2. 有一个, 因为这里虽然在两个文件中定义了, 但这两个都使用了 `static` 修饰符, 而且 `debug.h` 包含了 `common.h`, 这样就相当于在 `debug.h` 中定义了两个包含 `static` 修饰符的 `dummy`, 而在同一个文件中使用 `static` 修饰符同一个变量可以写多次定义, 但只定义了一次。

3. 会出现错误, 因为在 `debug.h` 文件中使用 `static` 定义了两次 `dummy` 且对这两个都进行了赋值。

为了进行检验, 在 `common.h` 中加入以下代码:

```

1 static int a;
2 static int dummy;
3 static int b;
4
5 static inline dummy_common() {
6     printf("dummy_common\n");
7     printf("  &a: %p\n", &a);
8     printf("  &b: %p\n", &b);
9     printf("  &dummy: %p\n", &dummy);
10 }

```

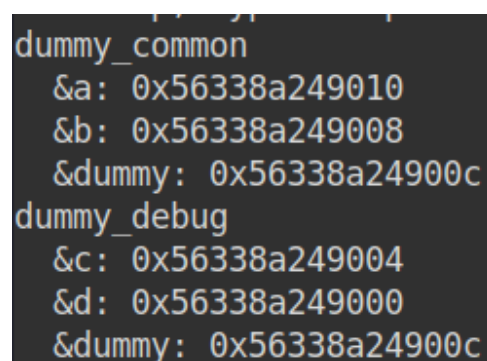
在 debug.h 中加入以下代码：

```

1 static int c;
2 static int dummy;
3 static int d;
4
5 static inline dummy_common() {
6     printf("dummy_common\n");
7     printf("  &c: %p\n", &c);
8     printf("  &d: %p\n", &d);
9     printf("  &dummy: %p\n", &dummy);
10 }

```

并在 main 函数中调用 dummy_common() 和 dummy_debug(), 运行的输入如图 A.1 所示。



```

dummy_common
&a: 0x56338a249010
&b: 0x56338a249008
&dummy: 0x56338a24900c
dummy_debug
&c: 0x56338a249004
&d: 0x56338a249000
&dummy: 0x56338a24900c

```

图 A.1: dummy 变量有关输出

从图中能够看出，两个文件中的 dummy 具有相同的内存地址。所以能够知道只有一个 dummy 实体。

问：

了解 Makefile。请描述你在 nemu/ 目录下敲入 make 后, make 程序如何组织.c 和.h 文

件, 最终生成可执行文件 `nemu/build/$ISA-nemu`. (这个问题包括两个方面: Makefile 的工作方式和编译链接的过程.) 关于 Makefile 工作方式的提示:

- Makefile 中使用了变量, 包含文件等特性
- Makefile 运用并重写了一些 implicit rules
- 在 `man make` 中搜索 -n 选项, 也许会对你有帮助
- RTFM

答:

Makefile 文件主要由一个个规则构成, 这些规则包含了目标文件、需要的源文件以及生成目标文件所需要的命令。在敲入 `make` 命令之后, 由于在 Makefile 中设置了 `DEFAULT_GOAL` 为 `app`, 所以会去运行 `app` 规则, 而 `app` 需要的文件为 `BINARY`, 这时又会去运行 `BINARY` 所对应的规则, 就这样一直进行下去直到可以运行一条规则, 之后再递归上来最终运行 `app` 规则。

在 Makefile 中, `.c` 和 `.h` 文件主要是由以下三行组织:

```
1 SRCS = $(shell find src/ -name "*.c" | grep -v "isa")
2 SRCS += $(shell find src/isa/${ISA} -name "*.c")
3 INC_DIR += ./include ./src/isa/${ISA}/include
```

其中前两行是通过 `shell` 命令来寻找该文件夹下包含的所有 `.c` 文件。

A.3 PA3

问: 文件读写的具体过程。仙剑奇侠传中有以下行为:

- 在 `navy-apps/apps/pal/src/global/global.c` 的 `PAL_LoadGame()` 中通过 `fread()` 读取游戏存档
- 在 `navy-apps/apps/pal/src/hal/hal.c` 的 `redraw()` 中通过 `NDL_DrawRect()` 更新屏幕

请结合代码解释仙剑奇侠传, 库函数, `libos`, `Nanos-lite`, `AM`, `NEMU` 是如何相互协助, 来分别完成游戏存档的读取和屏幕的更新。

答:

1. 使用 `fread()` 进行读档的过程为: 首先调用 `libc` 中的 `fread()` 函数, 之后 `libc` 会进行一系列的系统调用, 主要的系统调用为 `read()` 系统调用。当调用了 `read()` 之后, 会调用 `libos` 的 `_read()` 函数, 之后 `libos` 会调用 `_syscall_()` 函数, 在调用了 `_syscall_()` 函数之后, 会运行汇编指令 `int`, 这时才会进行真正的系统调用。`int` 在运行时, 会调用 `raise_intr()` 函数, 之后在 `raise_intr()` 里面, 会在中断向量表中进行查找, 找到入口之后执行。系统调用的入口为 `__am_vecsys`, 在 `nexus-am/am/src/x86/nemu/trap.S` 中

定义。紧接着会调用 `__am_asm_trap`，然后调用 `__am_irq_handle`，在 `__am_irq_handle` 中，会将其封装成 `_EVENT_SYSCALL` 事件，并转发给 `nanos-lite` 处理。`nanos-lite` 在 `do_syscall()` 函数中接收到该事件，根据系统调用号知道是 `read` 系统调用，然后会执行 `sys_read()` 帮助函数，在该函数里会调用真正的文件系统操作函数 `fs_read()`，之后文件系统才会进行真正的文件读取操作。

2. `NDL_DrawRect()` 函数的执行与前者有些相似。首先调用了 `libndl` 库，在该库中会打开设备文件 `/dev/fb` 和 `/dev/fbsync`，在接收到该函数调用后会向 `/dev/fb` 设备文件中写入，写入的过程和前者一样，最终到达了文件系统的 `fs_write()` 函数。在该函数中，判断出要写的文件是 `/dev/fb` 设备文件之后，会调用 `fb_write()` 帮助函数，之后会调用 `draw_rect()` 函数，该函数位于 `nexus-am/libs/klib/src/io.c` 中，在函数内会调用 `_io_write()` 函数，该函数是 IOE 的一部分。在 `_io_write()` 函数中，会转发给 `__am_video_write()` 函数，位于 `nexus-am/am/src/nemu-common/nemu-video.c` 文件中。在该函数中会执行 `out` 汇编指令，将数据传送给 `vga` 设备中。`vga` 设备在 `nemu/src/device/io/vga.c` 中定义，接收到数据后会保存在定义的显存中，当之后 `NDL` 库向 `/dev/fbsync` 设备文件中写入时，`vga` 设备最终会调用 `SDL` 库来更新画面。

完