计算机系统设计实验报告

PA2 - 简单复杂的机器: 冯诺依曼计算机系统

姓名：孟启轩
学号：2212452
专业：计算机科学与技术

- 计算机系统设计实验报告
  - PA2 - 简单复杂的机器: 冯诺依曼计算机系统
  - 一、实验目的
  - 二、实验重要内容
    - 任务
    - 立即数背后的故事
    - 堆和栈在哪里
    - 理解volatile关键字
    - 如何检测多个键同时被按下（这个问题我在阶段三中详细说明）
    - 神奇的调色板
  - 三、实验方法与结果
    - 实现指令的一般流程
    - （一）阶段一 运行第一个C程序——dummy.c
      - 1. push
      - 2. pop
      - 3. call
      - 4. ret
      - 5. sub
      - 6. xor
      - 7. 补充
    - （二）阶段二 实现更多的指令，运行更多的程序
      - 实现更多指令
        - 1、and
        - 2、xchg与nop
        - 3、SETcc
        - 4、jcc
        - 5、movsx, movzx
        - 6、sar, shl, shr
        - 7、test
        - 8、lea

- 9、add
- 10、cmp
- 10、jmp
- 11、mul, imul, div, idiv
- 12、sbb
- 13、not
- 14、neg
- 15、adc
- 16、inc，dec
- 17、cltd
- 18、leave
- 19、call_rm
- 运行更多程序
- （三）阶段三 输入输出
  - 串口
  - 时钟
  - 键盘
  - VGA
- 四、遇到的BUG及解决思路
- 五、必答题
- 六、总结与感悟

一、实验目的

（1）在PA1的基础上进一步探究“程序在计算机上运行”的原理， 深入了解指令周期与指令执行过程，同时了解 x86 指令集并实现 (x86) 指令系统。
（2）深入理解冯诺依曼计算机体系结构并尝试在 NEMU 中实现 (TRM、IOE)。
（3）了解基础设施测试、调试的基本框架与思想。

二、实验重要内容

任务

（1）了解指令周期、指令执行的原理以及指令执行过程在NEMU的体现， 尝试编写简单的指令并运行第一个C程序——dummy.c。
（2）了解AM的概念和基本原理， 完善NEMU的指令系统和调试的基础设施 (Differential testing功能)，使NEMU可以运行更多的客户程序和提供更强大的调试基础设施。
（3）了解I/O的基本原理和NEMU中I/O设备的工作方式，并加入I/O扩展 (串口、时钟、键盘、VGA)，使NEMU具备与外设交互的能力。

立即数背后的故事

立即数背后的故事

在 decode_op_I()函数中通过 instr_fetch()函数获得指令中的立即数. 别看这里就这么一行代码, 其实背后隐藏着针对字节序的慎重考虑. 我们知道 x86 是小端机, 当你使用高级语言或者汇编语言写了一个 32 位常数 0x1234 的时候, 在生成的二进制代码中, 这个常数对应的字节序列如下(假设这个常数在内存中的起始地址是 x):

```
x    x+1   x+2   x+3
+----+----+----+----+
| 34 | 12 | 00 | 00 |
+----+----+----+----+
```

而大多数 PC 机都是小端架构(我们相信没有同学会使用 IBM 大型机来做 PA), 当 NEMU 运行的时候, `op_src->imm=instr_fetch(eip,4)`, 这行代码会将 34 12 00 00 这个字节序列原封不动地从内存读入 imm 变量中, 主机的 CPU 会按照小端方式来解释这一字节序列, 于是会得到 0x1234, 符合我们的预期结果。

Motorola 68k 系列的处理器都是大端架构的. 现在问题来了, 考虑以下两种情况:

- 假设我们需要将 NEMU 运行在 Motorola 68k 的机器上(把 NEMU 的源代码编译成 Motorola 68k 的机器码)
- 假设我们需要编写一个新的模拟器 NEMU-Motorola-68k, 模拟器本身运行在 x86 架构中, 但它模拟的是 Motorola 68k 程序的执行

在这两种情况下, 你需要注意些什么问题?为什么会产生这些问题?怎么解决它们?事实上不仅仅是立即数的访问, 长度大于 1 字节的内存访问都需要考虑类似的问题. 我们在这里把问题统一抛出来, 以后就不再单独讨论了.

情况1：将NEMU运行在Motorola 68k机器上

Motorola 68k是大端机，而NEMU的原始设计基于x86的小端字节序。当直接在Motorola 68k上运行时，读取内存中的多字节数据（如32位立即数`0x1234`）会导致字节顺序错误：x86小端机存储为`34 12 00 00`，但大端机Motorola 68k会将其解释为`0x34120000`而非预期的`0x00001234`。为解决此问题，需修改`instr_fetch()`函数，通过检测当前机器的字节序（如使用`__BYTE_ORDER__`宏），在大端机上对读取的内存数据进行字节反转（如通过`ntohl()`或手动反转），确保数据解释与目标架构一致。

情况2：在x86上模拟Motorola 68k

NEMU-Motorola-68k模拟器运行在x86上，但需模拟大端机的行为。由于主机与目标架构的字节序不一致，直接使用x86的字节序处理数据会导致错误（如`0x34120000`被误认为`0x1234`）。解决方法是在模拟器内部实现字节序转换：**读取内存时**将小端主机存储的字节流转换为大端格式（如`0x34120000` → `0x00001234`），**写入内存时**将大端数据转换为小端存储格式。通过自定义函数（如`convert_to_big_endian()`）统一处理所有多字节操作，确保模拟器严格遵循Motorola 68k的大端规则。

堆和栈在哪里

堆和栈在哪里?

我们知道代码和数据都在可执行文件里面, 但却没有提到堆(heap)和栈(stack). 为什么堆和栈的内容没有放入可执行文件里面?那程序运行时刻用到的堆和栈又是怎么来的?AM 的代码是否能给你带来一些启发?

为什么堆和栈的内容没有放入可执行文件里面?

堆和栈的大小及内容在程序运行时才确定（如函数调用深度、malloc的调用次数），编译时无法预知。可执行文件仅包含静态分配的数据（如全局变量、代码指令），而动态内存分配需依赖运行时环境。

那程序运行时刻用到的堆和栈又是怎么来的?

程序运行时的堆和栈是由操作系统在进程启动时动态分配的内存区域：栈由操作系统初始化并自动管理，随函数调用上下文压入和弹出，其大小固定且向下增长；堆则通过`malloc`等函数向操作系统申请，由程序手动管理，大小动态扩展（通过系统调用如`brk/sbrk`或`mmap`），用于存储运行时不确定大小或跨函数共享的数据，两者均不在可执行文件中，而是运行时通过操作系统内存管理机制动态生成。

AM的代码是否能给你带来一些启发?

在PA2的AM中，堆和栈均基于TRM的物理内存（pmem数组）实现，其布局由链接脚本`loader.ld`指定：代码段（.text）、只读数据段（.rodata）、数据段（.data）和未初始化数据段（.bss）依次从`0x100000`开始加载，堆的起始地址`_heap_start`从`0x100000 + .bss结束地址`对齐到4096字节边界，而栈指针初始化为`0x7c00`，堆的上限设为`0x8000000`。AM通过自定义的`malloc`和`free`函数，采用链表结构动态管理堆空间，实现内存的分配与释放，而栈则在固定地址范围内按需扩展，这种设计使程序能够在TRM的内存中灵活管理堆和栈的动态需求。

理解volatile关键字

理解 volatile 关键字

也许你从来都没听说过 C 语言中有 volatile 这个关键字，但它从 C 语言诞生开始就一直存在. volatile 关键字的作用十分特别，它的作用是避免编译器对相应代码进行优化. 你应该动手体会一下 volatile 的作用，在 GNU/Linux 下编写以下代码：

```
void fun() {
    volatile unsigned char *p = (void *)0x8049000;
    *p = 0;
    while(*p != 0xff);
    *p = 0x33;
    *p = 0x34;
    *p = 0x86;
}
```

然后使用-O2 编译代码. 尝试去掉代码中的 volatile 关键字，重新使用-O2 编译，并对比去掉 volatile 前后反汇编结果的不同.

你或许会感到疑惑，代码优化不是一件好事情吗?为什么会有 volatile 这种奇葩的存在?思考一下，如果代码中的地址 0x8049000 最终被映射到一个设备寄存器，去掉 volatile 可能会带来什么问题?

用以下指令编译并进行反汇编：

```
gcc -O2 volatile.c -o volatile
objdump -d ./volatile
```

1. 带volatile的反汇编代码

```
0000000000001170 <fun>:
    1170:       c6 04 25 00 90 04 08    movb   $0x0,0x8049000
    1177:       00
    1178:       0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
    117f:       00
    1180:       0f b6 04 25 00 90 04    movzbl 0x8049000,%eax
    1187:       08
    1188:       3c ff                   cmp    $0xff,%al
    118a:       75 f4                   jne    1180 <fun+0x10>
    118c:       c6 04 25 00 90 04 08    movb   $0x33,0x8049000
    1193:       33
    1194:       c6 04 25 00 90 04 08    movb   $0x34,0x8049000
    119b:       34
    119c:       c6 04 25 00 90 04 08    movb   $0x86,0x8049000
    11a3:       86
    11a4:       c3                      ret
```

- `volatile` 告知编译器不能优化涉及 `*p` 的操作，因此：
  - 每次循环都会强制从内存读取 `*p` 的最新值（`movzbl` 指令）。
  - 后续的写入操作（`0x33`、`0x34`、`0x86`）正常执行。

2. 去掉volatile的反汇编代码

```
0000000000001140 <fun>:
    1140:       c6 04 25 00 90 04 08    movb   $0x0,0x8049000
    1147:       00
    1148:       eb fe                   jmp    1148 <fun+0x8>
```

- **循环被优化为死循环**：编译器认为 `*p` 的值不会被外部修改（如硬件），因此 `while(*p != 0xff)` 的条件始终为真（初始值为0），直接简化为 `jmp` 指令。
- **后续写入操作被删除**：编译器认为循环会无限执行，后续的 `0x33`、`0x34`、`0x86` 写入操作永远不会执行。

3. 为什么需要volatile?

- **地址 `0x8049000` 映射到设备寄存器**

1. **硬件交互需求**：
   假设 `0x8049000` 是某个**硬件设备的控制寄存器**，例如：
   - 写入 `0` 表示“启动设备操作”。
   - 硬件完成操作后，会将寄存器值设置为 `0xff`。
   - 程序需要等待 `0xff` 后继续执行后续操作（如 `0x33`、`0x34`）。
2. **volatile的作用**：
   - **确保循环正确等待硬件信号**：每次循环都强制读取寄存器的最新值，直到硬件设置为 `0xff`。
   - **避免代码逻辑错误**：若去掉 `volatile`，循环被优化为死循环，程序无法继续执行后续写入，导致硬件操作未完成时，程序已卡死，或者后续配置（如 `0x33`、`0x34`）未执行，设备可能无法正常工作。

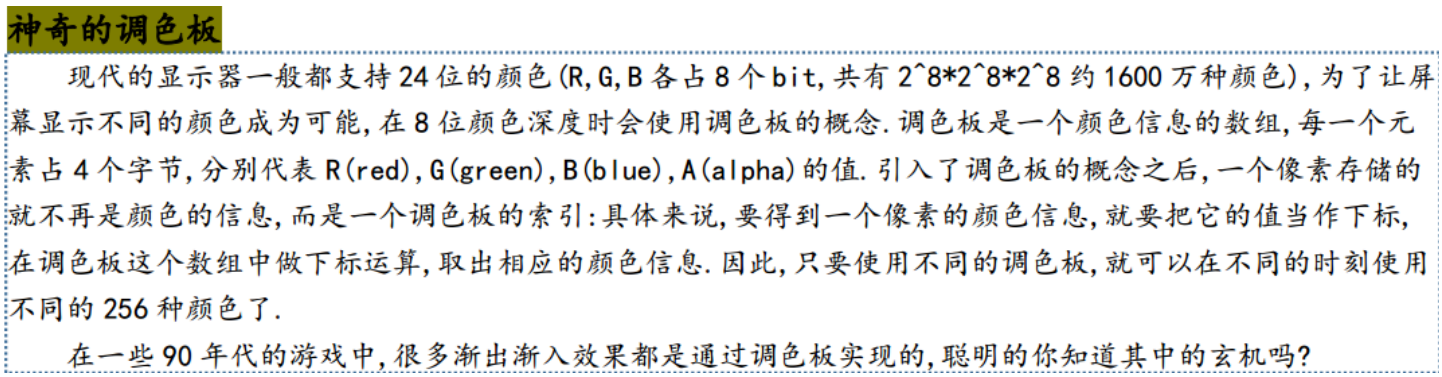
- **volatile存在的必要性**：
  volatile是编译器与硬件交互的桥梁，确保程序能感知到外部（如硬件、中断）对变量的修改，避

免因优化导致逻辑错误。

- **去掉volatile的后果**：
  - 在硬件场景中，程序可能因死循环卡死或未完成必要的配置，导致设备无法正常工作。
  - 在多线程场景中，可能导致数据竞争或不可重现的错误。

如何检测多个键同时被按下（这个问题我在阶段三中详细说明）

神奇的调色板


神奇的调色板

现代的显示器一般都支持24位的颜色(R,G,B各占8个bit,共有2^8*2^8*2^8约1600万种颜色),为了让屏幕显示不同的颜色成为可能,在8位颜色深度时会使用调色板的概念.调色板是一个颜色信息的数组,每一个元素占4个字节,分别代表R(red),G(green),B(blue),A(alpha)的值.引入了调色板的概念之后,一个像素存储的就不再是颜色的信息,而是一个调色板的索引:具体来说,要得到一个像素的颜色信息,就要把它的值当作下标,在调色板这个数组中做下标运算,取出相应的颜色信息.因此,只要使用不同的调色板,就可以在不同的时刻使用不同的256种颜色了.

在一些90年代的游戏中,很多渐出渐入效果都是通过调色板实现的,聪明的你知道其中的玄机吗?

调色板技术通过一个包含256种颜色的数组实现图像的高效渐入效果，每个像素仅存储调色板的索引值而非直接颜色数据，从而节省内存。渐入的核心是动态修改调色板中的颜色值：例如从全黑开始，逐帧增加所有颜色的R、G、B值，最终使画面从黑变白。此过程只需更新调色板数组并同步到屏幕，无需修改像素数据，计算成本极低。该技术能以低资源消耗实现复杂视觉效果，尤其适合早期硬件条件，通过调整调色板还可扩展为单色渐变、非线性变化等多样化效果。

三、实验方法与结果

> Opcode表和Group表我放在报告最后了

实现指令的一般流程

- 通过i386手册的附录A或者运行程序的反汇编文件报错信息查找未被实现的指令以及指令对应的操作码。
- 在i386手册中`Instruction Set Detail`部分查找指令对应的description、opcode、operation等信息。
- 在`ics2017/nemu/src/cpu/exec/all-instr.h`中声明指令的执行函数。
- 在`ics2017/nemu/src/cpu/exec/exec.c`中的opcode_table数组中,找到指令的操作码的对应位置，填写指令的译码函数、执行函数、操作数宽度。有些指令还需要填写make_group指令组。
  - IDEXW()：有译码函数和执行函数，指定操作数宽度。
  - IDEX()：有译码函数和执行函数，使用默认操作数宽度0
  - EXW()：无译码函数，只有执行函数，指定操作数宽度
  - EX()：无译码函数，只有执行函数，使用默认操作数宽度0
- 在`ics2017/nemu/include/cpu/rtl.h`实现这条指令需要用到的 RTL指令函数。
- 在`ics2017/nemu/src/cpu/decode.c`中实现指令的译码函数。

（一）阶段一 运行第一个C程序——dummy.c

在 `nexus-am/tests/cputest` 目录下键入 `make ARCH=x86-nemu ALL=dummy run` 运行dummy.c，输入c继续执行，出现i386 Manual，提示未实现 `eip(0x0010000a)` 处的指令，操作码 `e8` 。

```
[src/monitor/monitor.c,65,load_img] The image is /home/ics/ics2017/nexus-am/tests/cputest/build/dummy-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 10:56:05, Mar 25 2025
For help, type "help"
(nemu) c
invalid opcode(eip = 0x0010000a): e8 01 00 00 00 90 55 89 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x0010000a is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x0010000a) in the disassembling result to distinguish which case it is.

If it is the first case, see
 _ ____   ___    __   __  __                    _
(_)___ \ / _ \  / /   |  \/  |                  | |
 _  __) | (_) |/ /_   | \  / | __ _ _ __  _   _| |
| ||__ < > _ <| '_ \  | |\/| |/ _` | '_ \| | | | |
| |___) | (_) | (_) | | |  | | (_| | | | | |_| |_|
|_|____/ \___/ \___/  |_|  |_|\__,_|_| |_|\__,_(_)

for more details.

If it is the second case, remember:
* The machine is always right!
* Every line of untested code is always wrong!

(nemu)
```

查看 `ics2017/nexus-am/tests/cputest/build/dummy-x86-nemu.txt` 反汇编结果，我们需要实现call、push、pop、sub、xor、ret指令。

在 `ics2017/nemu/src/cpu/exec/all-instr.h` 中添加需要实现的指令。

```
make_EHelper(call);
make_EHelper(sub);
make_EHelper(xor);
make_EHelper(push);
make_EHelper(pop);
make_EHelper(ret);
```

> call指令需要用到push和pop指令，因此先实现push和pop指令。

1. push

push指令核心功能是将操作数存入栈中，并调整栈指针（在32位系统中是ESP）以指向新的栈顶。

- 根据i386手册中的操作码-指令对应关系表和Group表填写opcode_table和gp5。

```
  /* 0xff */
make_group(gp5,
    EX(inc), EX(dec), EX(call_rm), EMPTY,
    EX(jmp_rm), EMPTY, EX(push), EMPTY)
  /* 0x50 */    IDEX(r, push), IDEX(r, push), IDEX(r, push), IDEX(r, push),
  /* 0x54 */    IDEX(r, push), IDEX(r, push), IDEX(r, push), IDEX(r, push),
  /* 0x68 */    IDEX(I, push), IDEX(I_E2G, imul3), IDEXW(push_SI, push, 1), IDEX(I_E2G, imul3),
```

- 实现rtl_push函数，先将栈指针减4，然后将操作数存入栈中。

```
static inline void rtl_push(const rtlreg_t* src1) {
  // esp <- esp - 4
  // M[esp] <- src1
  rtl_subi(&cpu.esp, &cpu.esp, 4);
  rtl_sm(&cpu.esp, 4, src1);
}
```

- 实现make_EHelper(push)函数，直接调用rtl_push函数。

```
make_EHelper(push) {
  rtl_push(&(id_dest->val));//译码函数将源寄存器的内容保存在了id_dest->val中
  print_asm_template1(push);
}
```

2. pop

pop指令的核心功能是将栈顶的数据弹出到指定的目标操作数，并调整栈指针指向新的栈顶。

- 填写opcode_table。

```
  /* 0x58 */    IDEX(r, pop), IDEX(r, pop), IDEX(r, pop), IDEX(r, pop),
  /* 0x5c */    EMPTY, IDEX(r, pop), IDEX(r, pop), IDEX(r, pop),
```

- 实现rtl_pop函数，从ESP读取4字节数据到dest，恢复栈指针。

```
static inline void rtl_pop(rtlreg_t* dest) {
  // dest <- M[esp]
  // esp <- esp + 4
  rtl_lm(dest, &cpu.esp, 4);
  rtl_addi(&cpu.esp, &cpu.esp, 4);
}
```

- 实现make_EHelper(pop)函数，将栈顶数据弹出到临时寄存器t2，然后将弹出的数据写入目标操作数。

```
make_EHelper(pop) {
  rtl_pop(&t2);
  operand_write(id_dest, &t2); //decode.c
  print_asm_template1(pop);
}
```

3. call

```
E8   cd     CALL rel32        7+m             Call near, displacement relative to next instruction
```

E8 cd表示0xE8后面有一个4字节的操作数，表示要跳转的地址与当前地址的偏移量。

- 填写opcode_table和gp5。

```
  /* 0xff */
make_group(gp5,
    EX(inc), EX(dec), EX(call_rm), EMPTY,
    EX(jmp_rm), EMPTY, EX(push), EMPTY)
  /* 0xe8 */    IDEX(J, call), IDEX(J, jmp), EMPTY, IDEXW(J, jmp, 1),
```

- call指令的译码函数为decode_J，需要实现对有符号立即数的解析函数decode_op_SI，而decode_op_SI函数就是make_DopHelper(SI)。对width为1的立即数进行符号扩展：先将instr_fetch()取到的值强制转换成int8_t进行截断，再赋值给类型为int32_t的op->simm ，int8_t到int32_t会自动进行符号扩展。

```
static inline make_DopHelper(SI){
    assert(op->width == 1 || op->width == 4);
    op->type = OP_TYPE_IMM;
    /* TODO: Use instr_fetch() to read `op->width' bytes of memory
     * pointed by `eip'. Interpret the result as a signed immediate,
     * and assign it to op->simm.
     *
     op->simm = ???
     */
    op->simm = instr_fetch(eip, op->width);
    if (op->width == 1){
        op->simm = (int8_t)op->simm;
    }
    rtl_li(&op->val, op->simm);
#ifdef DEBUG
    snprintf(op->str, OP_STR_SIZE, "$0x%x", op->simm);
#endif
}
```

- 实现make_EHelper(call)函数，将eip压栈，然后进行跳转。

```
make_EHelper(call) {
  // the target address is calculated at the decode stage
  rtl_li(&t2, decoding.seq_eip);
  rtl_push(&t2);
  decoding.is_jmp = 1;
  print_asm("call %x", decoding.jmp_eip);
}
```

4. ret

```
C3          RET             10+m            Return (near) to caller
```

ret指令只需要实现上图格式即可，没有操作数，所以不需要译码函数。

- 填写opcode_table。

```
  /* 0xc0 */    IDEXW(gp2_Ib2E, gp2, 1), IDEX(gp2_Ib2E, gp2), EMPTY, EX(ret),
```

- 实现make_EHelper(ret)函数，弹出栈顶数据到临时寄存器t2，将 t2 的值设为新的指令指针EIP，然后标记为跳转指令。

```
make_EHelper(ret) {
  rtl_pop(&t2);
  decoding.jmp_eip = t2;
  decoding.is_jmp = 1;
  print_asm("ret");
}
```

5. sub

SUB — Integer Subtraction

Opcode	Instruction	Clocks	Description
2C ib	SUB AL,imm8	2	Subtract immediate byte from AL
2D iw	SUB AX,imm16	2	Subtract immediate word from AX
2D id	SUB EAX,imm32	2	Subtract immediate dword from EAX
80 /5 ib	SUB r/m8,imm8	2/7	Subtract immediate byte from r/m byte
81 /5 iw	SUB r/m16,imm16	2/7	Subtract immediate word from r/m word
81 /5 id	SUB r/m32,imm32	2/7	Subtract immediate dword from r/m dword
83 /5 ib	SUB r/m16,imm8	2/7	Subtract sign-extended immediate byte from r/m word
83 /5 ib	SUB r/m32,imm8	2/7	Subtract sign-extended immediate byte from r/m dword
28 /r	SUB r/m8,r8	2/6	Subtract byte register from r/m byte
29 /r	SUB r/m16,r16	2/6	Subtract word register from r/m word
29 /r	SUB r/m32,r32	2/6	Subtract dword register from r/m dword
2A /r	SUB r8,r/m8	2/7	Subtract byte register from r/m byte
2B /r	SUB r16,r/m16	2/7	Subtract word register from r/m word
2B /r	SUB r32,r/m32	2/7	Subtract dword register from r/m dword

ib/iw/id：表示立即数的长度为1/2/4字节。

/5：表示通过ModR/M字节的Reg字段选择操作（5对应SUB指令）。
/r：表示指令需要解析ModR/M字节来确定操作数的寻址方式。
sub操作还会影响CF、OF、SF、ZF标志位。

> 由于sub指令涉及标志位的更新，所以我们们要先实现一些标志位相关的函数，这部分我在后面“7.补充”中说明。

- 填写opcode_table和gp1。

```
/* 0x80, 0x81, 0x83 */
make_group(gp1,
    EX(add), EX(or), EX(adc), EX(sbb),
    EX(and), EX(sub), EX(xor), EX(cmp))
  /* 0x28 */    EMPTY, IDEX(G2E, sub), EMPTY, IDEX(E2G, sub),
  /* 0x2c */    EMPTY, IDEX(I2a, sub), EMPTY, EMPTY,
  /* 0x80 */    IDEXW(I2E, gp1, 1), IDEX(I2E, gp1), EMPTY, IDEX(SI2E, gp1),
```

- 实现make_EHelper(sub)函数。

```
make_EHelper(sub) {
  rtl_sub(&t2, &id_dest->val, &id_src->val);
  rtl_update_ZFSF(&t2, id_dest->width);//更新零标志ZF和符号标志SF

  //设置进位标志CF
  rtl_sltu(&t0, &id_dest->val, &id_src->val);
  rtl_set_CF(&t0);

  //计算溢出标志OF ////OF 触发条件：当两个同号的操作数相减，结果的符号与原操作数符号不一致 时，产生溢
  rtl_xor(&t0, &id_dest->val, &id_src->val);//检测操作数的符号位是否相同。若相同，结果符号位为0
  rtl_xor(&t1, &id_dest->val, &t2);//检测结果的符号位是否与原dest相同
  rtl_and(&t0, &t0, &t1);//提取关键位，符号位的变化
  rtl_msb(&t0, &t0, id_dest->width);//提取最高位，若为1则OF=1
  rtl_set_OF(&t0);

  operand_write(id_dest, &t2);
  print_asm_template2(sub);
}
```

6. xor

仍然是先查看i386手册获取提示。xor指令也会影响CF、OF、SF、ZF标志位，其中CF和OF恒为0。

XOR — Logical Exclusive OR

Opcode	Instruction	Clocks	Description
34 ib	XOR AL,imm8	2	Exclusive-OR immediate byte to AL
35 iw	XOR AX,imm16	2	Exclusive-OR immediate word to AX
35 id	XOR EAX,imm32	2	Exclusive-OR immediate dword to EAX
80 /6 ib	XOR r/m8,imm8	2/7	Exclusive-OR immediate byte to r/m byte
81 /6 iw	XOR r/m16,imm16	2/7	Exclusive-OR immediate word to r/m word
81 /6 id	XOR r/m32,imm32	2/7	Exclusive-OR immediate dword to r/m dword
83 /6 ib	XOR r/m16,imm8	2/7	XOR sign-extended immediate byte with r/m word
83 /6 ib	XOR r/m32,imm8	2/7	XOR sign-extended immediate byte with r/m dword
30 /r	XOR r/m8,r8	2/6	Exclusive-OR byte register to r/m byte
31 /r	XOR r/m16,r16	2/6	Exclusive-OR word register to r/m word
31 /r	XOR r/m32,r32	2/6	Exclusive-OR dword register to r/m dword
32 /r	XOR r8,r/m8	2/7	Exclusive-OR byte register to r/m byte
33 /r	XOR r16,r/m16	2/7	Exclusive-OR word register to r/m word
33 /r	XOR r32,r/m32	2/7	Exclusive-OR dword register to r/m dword

Operation

```
DEST ← LeftSRC XOR RightSRC
CF ← 0
OF ← 0
```

- 填写opcode_table和gp1。

```
/* 0x80, 0x81, 0x83 */
make_group(gp1,
    EX(add), EX(or), EX(adc), EX(sbb),
    EX(and), EX(sub), EX(xor), EX(cmp))
  /* 0x30 */    IDEXW(G2E, xor, 1), IDEX(G2E, xor), IDEXW(E2G, xor, 1), IDEX(E2G, xor),
  /* 0x34 */    EMPTY, IDEX(I2a, xor), EMPTY, EMPTY,
```

- 实现make_EHelper(xor)函数。首先计算目标操作数与源操作数的按位异或结果，并将结果暂存到临时寄存器t2；然后通过operand_write将结果写入目标操作数；接着根据结果更新零标志ZF和符号标志SF；CF和OF恒为0。

```
make_EHelper(xor) {
  rtl_xor(&t2, &id_dest->val, &id_src->val);
  operand_write(id_dest, &t2);
  rtl_update_ZFSF(&t2, id_dest->width);
  rtl_set_CF(&tzero);
  rtl_set_OF(&tzero);
  print_asm_template2(xor);
}
```

or指令类似，把rtl_xor替换成rtl_or，阶段二不再赘述。

7. 补充

- 在 `ics2017/nemu/include/cpu/reg.h` 中，在CPU_ state结构体中增加eflags共用体。

```
struct {
    unsigned int CF:1;
    unsigned int :5;
    unsigned int ZF:1;
    unsigned int SF:1;
    unsigned int :1;
    unsigned int IF:1;
    unsigned int :1;
    unsigned int OF:1;
    unsigned int :20;
  }eflags;
```

- 在 `ics2017/nemu/src/monitor/monitor.c` 的restart函数中初始化eflags，eflags的初值为0x00000002H。

EFLAGS =00000002H

```
static inline void restart() {
  /* Set the initial instruction pointer. */
  cpu.eip = ENTRY_START;
  unsigned int origin=2;
  memcpy(&cpu.eflags,&origin,sizeof(cpu.eflags));
#ifdef DIFF_TEST
  init_qemu_reg();
#endif
}
```

- 实现EFLAGS寄存器标志位读写函数。

```
#define make_rtl_setget_eflags(f) \
  static inline void concat(rtl_set_, f) (const rtlreg_t* src) { \
    /* TODO(); */ \
    cpu.eflags.f = *src;  \
  } \
  static inline void concat(rtl_get_, f) (rtlreg_t* dest) { \
    /* TODO(); */ \
    *dest = cpu.eflags.f; \
  }
```

- 实现EFLAGS寄存器的标志位更新函数

```
static inline void rtl_eq0(rtlreg_t* dest, const rtlreg_t* src1) {
  // dest <- (src1 == 0 ? 1 : 0)
  rtl_sltui(dest, src1, 1);
}
static inline void rtl_eqi(rtlreg_t* dest, const rtlreg_t* src1, int imm) {
  // dest <- (src1 == imm ? 1 : 0)
  rtl_xori(dest, src1, imm);
  rtl_eq0(dest, dest);
}
static inline void rtl_neq0(rtlreg_t* dest, const rtlreg_t* src1) {
  // dest <- (src1 != 0 ? 1 : 0)
  rtl_eq0(dest, src1);
  rtl_eq0(dest, dest);
}
static inline void rtl_msb(rtlreg_t* dest, const rtlreg_t* src1, int width) {
  // dest <- src1[width * 8 - 1]
  rtl_shri(dest, src1, width*8-1);
  rtl_andi(dest, dest, 0x1);
}
static inline void rtl_update_ZF(const rtlreg_t* result, int width) {
  // eflags.ZF <- is_zero(result[width * 8 - 1 .. 0])
  assert(result != &t0);
  rtl_andi(&t0, result, (0xffffffffu) >> (4-width)*8);
  rtl_eq0(&t0, &t0);
  rtl_set_ZF(&t0);
}
static inline void rtl_update_SF(const rtlreg_t* result, int width) {
  // eflags.SF <- is_sign(result[width * 8 - 1 .. 0])
  assert(result != &t0);
  rtl_msb(&t0, result, width);
  rtl_set_SF(&t0);
}
static inline void rtl_update_ZFSF(const rtlreg_t* result, int width) {
  rtl_update_ZF(result, width);
  rtl_update_SF(result, width);
}
```

阶段一测试结果

```
[src/monitor/monitor.c,65,load_img] The image is /home/ics/ics2017/nexus-am/tests/cputest/build/dummy-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 14:22:49, Mar 29 2025
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x0010001b

(nemu) q
 dummy
```

（二）阶段二 实现更多的指令，运行更多的程序

> 阶段二是要实现更多指令，运行更多的程序，通过 `bash runall.sh` 一键回归测试，即通过 `ics2017/nexus-am/tests/cputest/` 下的所有测试。这一阶段需要实现的指令很多，但是基本流程是一样的。我们尽量一个一个程序进行测试，为此，我修改了Makefile.check中的ARCH为 `ARCH ?= x86-nemu` ，这样，我们就可以用 `make ALL=add run` （add位置为想要测试的程序）指令来一个一个测试程序。如果报错，我们就去查看对应程序的反汇编文件，定位到报错点看是哪个指令没有正确实现，然后我们根据i386手册去实现相应的指令。

在`ics2017/nemu/src/cpu/exec/all-instr.h`中添加需要实现的指令。

```
#include "cpu/exec.h"

make_EHelper(mov);

make_EHelper(operand_size);

make_EHelper(inv);
make_EHelper(nemu_trap);

make_EHelper(push);
make_EHelper(pop);
make_EHelper(movzx);
make_EHelper(movsx);
make_EHelper(cwtl);
make_EHelper(cltd);
make_EHelper(lea);
make_EHelper(leave);
```

```
make_EHelper(call);
make_EHelper(call_rm);
make_EHelper(ret);
make_EHelper(jmp);
make_EHelper(jmp_rm);
make_EHelper(jcc);

make_EHelper(add);
make_EHelper(sub);
make_EHelper(sbb);
make_EHelper(neg);
make_EHelper(inc);
make_EHelper(dec);
make_EHelper(cmp);
make_EHelper(adc);
make_EHelper(mul);
make_EHelper(imul1);
make_EHelper(imul2);
make_EHelper(imul3);
make_EHelper(div);
make_EHelper(idiv);
```

```
make_EHelper(or);
make_EHelper(and);
make_EHelper(xor);
make_EHelper(not);
make_EHelper(setcc);
make_EHelper(rol);
make_EHelper(sar);
make_EHelper(sal);
make_EHelper(shl);
make_EHelper(shr);
make_EHelper(test);

make_EHelper(nop);

make_EHelper(in);
make_EHelper(out);
```

实现更多指令

1、and

and指令我们根据i386手册的信息进行实现即可。

```
/* 0x80, 0x81, 0x83 */
make_group(gp1,
    EX(add), EX(or), EX(adc), EX(sbb),
    EX(and), EX(sub), EX(xor), EX(cmp))
  /* 0x20 */    IDEXW(G2E, and, 1), IDEX(G2E, and), IDEXW(E2G, and, 1), IDEX(E2G, and),
  /* 0x24 */    IDEXW(I2a, and, 1), IDEX(I2a, and), EMPTY, EMPTY,
make_EHelper(and) {
  rtl_and(&t2, &id_dest->val, &id_src->val);
  operand_write(id_dest, &t2);
  rtl_update_ZFSF(&t2, id_dest->width);
  rtl_set_CF(&tzero);
  rtl_set_OF(&tzero);
  print_asm_template2(and); }
```

2、xchg与nop

运行add，在0x0010006e停止，查看反汇编文件，`10006e: 66 90 xchg   %ax,%ax`，这里没有实际效果，并且90对应nop，所以我们可以用nop替代xchg，nop已经实现，我们直接添加opcode_table即可。

```
  /* 0x90 */    EX(nop), EMPTY, EMPTY, EMPTY,
```

3、SETcc

SETcc指令一类条件设置指令，根据CPU的标志寄存器的当前状态，将目标操作数设置为1或0。

填写2 byte_opcode_table。

```
  /* 0x90 */    IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E, setcc, 1),
  /* 0x94 */    IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E, setcc, 1),
  /* 0x98 */    IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E, setcc, 1),
  /* 0x9c */    IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E, setcc, 1),
```

与前面的指令不同，SETcc指令的操作码是两字节，其中第一个字节固定为0x0f。程序在遇到0x0f时，会执行make_EHelper(2byte_esc)函数，以确定完整的两字节操作码。随后，程序会调用idex(eip, &opcode_table[opcode])函数，对指令进行进一步的解码和执行。执行函数需要调用rtl_setcc()，我们进行实现。

```
// TODO: Query EFLAGS to determine whether the condition code is satisfied.
// dest <- ( cc is satisfied ? 1 : 0)
switch (subcode & 0xe) {
  case CC_O:rtl_get_OF(dest);break;
  case CC_B:rtl_get_CF(dest);break;
  case CC_E:rtl_get_ZF(dest);break;
  case CC_BE:
    assert(dest != &t0);
    rtl_get_CF(dest);
    rtl_get_ZF(&t0);
    rtl_or(dest, dest, &t0);
    break;
  case CC_S:rtl_get_SF(dest);break;
  case CC_L:
    assert(dest != &t0);
    rtl_get_SF(dest);
    rtl_get_OF(&t0);
    rtl_xor(dest, dest, &t0);
    break;
  case CC_LE:
    assert(dest != &t0);
    rtl_get_SF(dest);
    rtl_get_OF(&t0);
    rtl_xor(dest, dest, &t0);
    rtl_get_ZF(&t0);
    rtl_or(dest, dest, &t0);
    break;
  default: panic("should not reach here");
  case CC_P: panic("n86 does not have PF");
}
```

4、jcc

jcc指令的作用是根据eflags寄存器中的状态标志位来判断是否跳转到目标操作数指定的地址执行后续的指令，即根据这些状态标志位来将decoding.is_jmp置1或0。执行函数已经实现，我们来补充opcode_table。

```
/* 0x70 */    IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1),
/* 0x74 */    IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1),
/* 0x78 */    IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1),
/* 0x7c */    IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1),
//2 byte_opcode_table
/* 0x80 */    IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc),
/* 0x84 */    IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc),
/* 0x88 */    IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc),
/* 0x8c */    IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc),
```

5、movsx, movzx

movsx和movzx指令是mov指令的变体，movsx作用是将源操作数的值符号扩展并传送给目的操作数，movzx作用是将源操作数的值零扩展并传送给目的操作数。执行函数已经实现，我们来填写2 byte_opcode_table。

```
/* 0xb4 */    EMPTY, EMPTY, IDEXW(mov_E2G, movzx, 1), IDEXW(mov_E2G, movzx, 2),
/* 0xb8 */    EMPTY, EMPTY, EMPTY, EMPTY,
/* 0xbc */    EMPTY, EMPTY, IDEXW(mov_E2G, movsx, 1), IDEXW(mov_E2G, movsx, 2),
```

6、sar, shl, shr

shl: 逻辑左移, 低位补0，高位进CF
shr: 逻辑右移, 低位进CF，高位补0
sar: 算数右移, 低位进CF，高位补符号位
根据i386手册填写gp2，三个指令会对标志位有影响但在NEMU中不是必须实现的，我们完善相应的执行函数。

```
  /* 0xc0, 0xc1, 0xd0, 0xd1, 0xd2, 0xd3 */
make_group(gp2,
    EMPTY, EMPTY, EMPTY, EMPTY,
    EX(shl), EX(shr), EMPTY, EX(sar))

make_EHelper(sar) {
  // unnecessary to update CF and OF in NEMU
  rtl_sext(&t2, &id_dest->val, id_dest->width);
  rtl_sar(&t2, &t2, &id_src->val);
  operand_write(id_dest, &t2);
  rtl_update_ZFSF(&t2, id_dest->width);
  print_asm_template2(sar); }
make_EHelper(shl) {
  // unnecessary to update CF and OF in NEMU
  rtl_shl(&t2, &id_dest->val, &id_src->val);
  operand_write(id_dest, &t2);
  rtl_update_ZFSF(&t2, id_dest->width);
  print_asm_template2(shl); }
make_EHelper(shr) {
  // unnecessary to update CF and OF in NEMU
  rtl_shr(&t2, &id_dest->val, &id_src->val);
  operand_write(id_dest, &t2);
  rtl_update_ZFSF(&t2, id_dest->width);
  print_asm_template2(shr); }
```

7、test

test指令很关键，除了dummy所有的测试样例都需要正确实现test，否则测试会FAIL。test对两个参数执行and操作，并根据结果设置标志寄存器。

```
  /* 0xf6, 0xf7 */
make_group(gp3,
    IDEX(test_I, test), EMPTY, EX(not), EX(neg),
    EX(mul), EX(imul1), EX(div), EX(idiv))
  /* 0xa8 */    IDEXW(I2a, test, 1), IDEX(I2a, test), EMPTY, EMPTY,
  /* 0x84 */    IDEXW(G2E, test, 1), IDEX(G2E, test), EMPTY, EMPTY,

make_EHelper(test) {
  rtl_and(&t2, &id_dest->val, &id_src->val);
  rtl_update_ZFSF(&t2, id_dest->width);
  rtl_set_CF(&tzero);
  rtl_set_OF(&tzero);
  print_asm_template2(test);
}
```

一般来说，一个指令组的译码函数通常是一样的，但test这里是先执行指令组的译码函数decode_E()，再执行decode_test_I()进行立即数的译码，所以我们这里定义为IDEX(test_I,test)。

8、lea

lea指令计算内存操作数的有效地址（即内存位置的偏移地址），并将结果存入指定寄存器，不访问该内存的实际数据。我们将其添加到opcode_table。

```
  /* 0x8c */    EMPTY, IDEX(lea_M2G, lea), EMPTY, EMPTY,
```

9、add

核心逻辑与sub相似，只需要修改CF和OF的判别逻辑即可。

```
/* 0x80, 0x81, 0x83 */
make_group(gp1,
    EX(add), EX(or), EX(adc), EX(sbb),
    EX(and), EX(sub), EX(xor), EX(cmp))
  /* 0x00 */    IDEXW(G2E, add, 1), IDEX(G2E, add), IDEXW(E2G, add, 1), IDEX(E2G, add),
  /* 0x04 */    IDEXW(I2a, add, 1), IDEX(I2a, add), EMPTY, EMPTY,

make_EHelper(add) {
  rtl_add(&t2, &id_dest->val, &id_src->val);
  operand_write(id_dest, &t2);
  rtl_update_ZFSF(&t2, id_dest->width);
  rtl_sltu(&t0,&t2, &id_dest->val);
  rtl_set_CF(&t0);
  rtl_xor(&t0, &id_src->val, &t2);
  rtl_xor(&t1, &id_dest->val, &t2);
  rtl_and(&t0, &t0,&t1);
  rtl_msb(&t0, &t0, id_dest->width);
  rtl_set_OF(&t0);
  print_asm_template2(add);
}
```

10、cmp

cmp指令和sub指令唯一的区别是：cmp指令只影响标志位，不写回操作数。

```
/* 0x80, 0x81, 0x83 */
make_group(gp1,
    EX(add), EX(or), EX(adc), EX(sbb),
    EX(and), EX(sub), EX(xor), EX(cmp))
  /* 0x38 */    IDEXW(G2E, cmp, 1), IDEX(G2E, cmp), IDEXW(E2G, cmp,1), IDEX(E2G, cmp),
  /* 0x3c */    IDEXW(I2a, cmp, 1), IDEX(I2a, cmp), EMPTY, EMPTY,

make_EHelper(cmp) {
  rtl_sub(&t2, &id_dest->val, &id_src->val);
  rtl_update_ZFSF(&t2, id_dest->width);
  rtl_sltu(&t0, &id_dest->val, &id_src->val);
  rtl_set_CF(&t0);
  rtl_xor(&t0, &id_dest->val, &id_src->val);
  rtl_xor(&t1, &id_dest->val, &t2);
  rtl_and(&t0, &t0, &t1);
  rtl_msb(&t0, &t0, id_dest->width);
  rtl_set_OF(&t0);
  print_asm_template2(cmp);
}
```

10、jmp

执行函数已经实现，我们来填写opcode_table。

```
  /* 0xe8 */    IDEX(J, call), IDEX(J, jmp), EMPTY, IDEXW(J, jmp, 1),
```

11、mul, imul, div, idiv

mul指令是无符号乘法，imul指令是有符号乘法。div指是无符号除法，idiv指令是有符号除法。对应的执行函数都已经实现，我们只需要填写gp3和opcode_table。

```
  /* 0xf6, 0xf7 */
make_group(gp3,
    IDEX(test_I, test), EMPTY, EX(not), EX(neg),
    EX(mul), EX(imul1), EX(div), EX(idiv))
  /* 0x68 */    IDEX(I, push), IDEX(I_E2G, imul3), IDEXW(push_SI, push, 1), IDEX(I_E2G, imul3),
  //2 byte_opcode_table
  /* 0xac */    EMPTY, EMPTY, EMPTY, IDEX(E2G, imul2),
```

12、sbb

sbb指令和sub指令的区别是：sbb指令会考虑CF的进位，而sub指令不会，其执行函数已经实现，我们来填写gp1和opcode_table。

```
/* 0x80, 0x81, 0x83 */
make_group(gp1,
    EX(add), EX(or), EX(adc), EX(sbb),
    EX(and), EX(sub), EX(xor), EX(cmp))
  /* 0x18 */    IDEXW(G2E, sbb, 1), IDEX(G2E, sbb), IDEXW(E2G, sbb, 1), IDEX(E2G, sbb),
  /* 0x1c */    IDEXW(I2a, sbb, 1), IDEX(I2a, sbb), EMPTY, EMPTY,
```

13、not

根据i386手册提示信息进行实现即可。

```
  /* 0xf6, 0xf7 */
make_group(gp3,
    IDEX(test_I, test), EMPTY, EX(not), EX(neg),
    EX(mul), EX(imul1), EX(div), EX(idiv))

make_EHelper(not) {
  rtl_not(&id_dest->val);
  operand_write(id_dest, &id_dest->val);
  print_asm_template1(not);
}
```

14、neg

neg指令是求操作数的补码 (按位取反再加1)，i386手册说neg会影响OF、SF、ZF、CF标志位，我们实现执行函数并填写gp3（上面not已经给出）。

```
make_EHelper(neg) {
  rtl_sub(&t2, &tzero, &id_dest->val);//0-src
  rtl_update_ZFSF(&t2, id_dest->width);
  rtl_neq0(&t0, &id_dest->val);
  rtl_set_CF(&t0);
  rtl_eqi(&t0, &id_dest->val, 0x80000000);
  rtl_set_OF(&t0);
  operand_write(id_dest, &t2);
  print_asm_template1(neg);
}
```

15、adc

add不会考虑CF的进位，而adc指令会，其执行函数已经实现，我们来填写gp1和opcode_table。

```
/* 0x80, 0x81, 0x83 */
make_group(gp1,
    EX(add), EX(or), EX(adc), EX(sbb),
    EX(and), EX(sub), EX(xor), EX(cmp))
  /* 0x10 */    IDEXW(G2E, adc, 1), IDEX(G2E, adc), IDEXW(E2G, adc, 1), IDEX(E2G, adc),
  /* 0x14 */    IDEXW(I2a, adc, 1), IDEX(I2a, adc), EMPTY, EMPTY,
```

16、inc，dec

inc指令用于将目标操作数的值递增1，溢出检测基于结果是否为最小有符号数。
dec指令用于将目标操作数的值递减1，溢出检测基于结果是否为最大有符号数。

```
  /* 0xfe */
make_group(gp4,
    EX(inc), EX(dec), EMPTY, EMPTY,
    EMPTY, EMPTY, EMPTY, EMPTY)
  /* 0xff */
make_group(gp5,
    EX(inc), EX(dec), EX(call_rm), EMPTY,
    EX(jmp_rm), EMPTY, EX(push), EMPTY)
  /* 0x40 */    IDEX(r, inc), IDEX(r, inc), IDEX(r, inc), IDEX(r, inc),
  /* 0x44 */    IDEX(r, inc), IDEX(r, inc), IDEX(r, inc), IDEX(r, inc),
  /* 0x48 */    IDEX(r, dec), IDEX(r, dec), IDEX(r, dec), IDEX(r, dec),
  /* 0x4c */    IDEX(r, dec), IDEX(r, dec), IDEX(r, dec), IDEX(r, dec),

make_EHelper(inc) {
  rtl_addi(&t2, &id_dest->val, 1);
  operand_write(id_dest, &t2);
  rtl_update_ZFSF(&t2, id_dest->width);
  rtl_eqi(&t0, &t2, 0x80000000);
  rtl_set_OF(&t0);
  print_asm_template1(inc);
}
make_EHelper(dec) {
  rtl_subi(&t2, &id_dest->val,1);
  operand_write(id_dest, &t2);
  rtl_update_ZFSF(&t2, id_dest->width);
  rtl_eqi(&t0, &t2, 0x7fffffff);
  rtl_set_OF(&t0);
  print_asm_template1(dec);
}
```

17、cltd

cltd指令（在i386手册中对应cdq指令）的作用是将eax（或ax）寄存器中的值进行符号扩展，将其扩展为64位（或32位）有符号数，其中eax/ax的值作为低32位（或低16位），而高32位（或高16位）则通过复制eax的最高位填入edx/dx寄存器，这一操作通常在执行除法指令div或idiv前使用，以确保被除数的符号正确扩展，从而支持80386 CPU上对有符号数除法的处理；值得注意的是，cltd指令无需操作数，其功能直接由指令本身定义，不需要译码函数。

```
  /* 0x98 */    EX(cwtl), EX(cltd), EMPTY, EMPTY,

make_EHelper(cltd) {
  if (decoding.is_operand_size_16) {
    rtl_lr_w(&t0, R_AX);
    rtl_sext(&t0, &t0, 2);
    rtl_sari(&t0, &t0, 31);
    rtl_sr_w(R_DX, &t0);}
  else {  rtl_sari(&cpu.edx, &cpu.eax, 31); }
  print_asm(decoding.is_operand_size_16 ? "cwtl" : "cltd");
}
```

18、leave

leave指令用于子过程（通常为函数）的退出，其作用是销毁当前函数的栈帧，并恢复到调用该函数的上一个函数的栈帧。leave指令不带操作数，不需要译码函数。

```
  /* 0xc8 */    EMPTY, EX(leave), EMPTY, EMPTY,

make_EHelper(leave) {
  rtl_mv(&cpu.esp, &cpu.ebp);
  rtl_pop(&cpu.ebp);
  print_asm("leave");
}
```

19、call_rm

call_rm指令通过压栈保存返回地址并跳转到操作数指定的地址，实现子程序调用的控制转移。

```
  /* 0xff */
make_group(gp5,
    EX(inc), EX(dec), EX(call_rm), EMPTY,
    EX(jmp_rm), EMPTY, EX(push), EMPTY)

make_EHelper(call_rm) {
  rtl_li(&t2, decoding.seq_eip);
  rtl_push(&t2);
  decoding.jmp_eip = id_dest->val;
  decoding.is_jmp = 1;
  print_asm("call *%s", id_dest->str);
}
```

运行更多程序

differential testing

differential testing通过对比QEMU和NEMU执行状态来验证指令实现正确性，其核心是让两者在相同初始寄存器状态下逐条执行同一客户程序指令，每执行一条指令后仅对比寄存器状态（因内存对比开销过大），若发现状态差异则立即报错终止；该功能通过`different_step()`函数实现，
在`exec_wrapper()`中调用时，会触发`difftest_step()`让后台QEMU执行相同指令，同时利用`is_skip_nemu`和`is_skip_qemu`变量跳过少数难以对比的指令，但由于需通过进程间通信保持同步导致性能显著下降，因此这种测试并非完全精确，需在`nemu/include/common.h`中开启`DIFF_TEST`宏才能启用。
我们来完善输出提示信息以帮助我们更好的调试，帮助我们更快定位问题并进行实验。

```
// TODO: Check the registers st1ate with QEMU.
// Set `diff` as `true` if they are not the same.
if(r.eax!=cpu.eax) {
  printf("expect: %d true: %d at: %x\n", r.eax, cpu.eax, cpu.eip);
  diff=true;
}
if(r.ecx!=cpu.ecx) {
  printf("expect: %d true: %d at: %x \n", r.ecx, cpu.ecx, cpu.eip);
  diff=true;
}
if(r.edx!=cpu.edx) {
  printf("expect: %d true: %d at: %x\n", r.edx, cpu.edx, cpu.eip);
  diff=true;
}
if(r.ebx!=cpu.ebx) {
  printf("expect: %d true: %d at: %x\n", r.ebx, cpu.ebx, cpu.eip);
  diff=true;
}
if(r.esp!=cpu.esp) {
  printf("expect: %d true: %d at: %x\n", r.esp, cpu.esp, cpu.eip);
  diff=true;
}
if(r.ebp!=cpu.ebp) {
  printf("expect: %d true: %d at: %x\n", r.ebp, cpu.ebp, cpu.eip);
  diff=true;
}
if(r.esi!=cpu.esi) {
  printf("expect: %d true: %d at: %x\n", r.esi, cpu.esi, cpu.eip);
  diff=true;
}
if(r.edi!=cpu.edi) {
  printf("expect: %d true: %d at: %x\n", r.edi, cpu.edi, cpu.eip);
  diff=true;
}
if(r.eip!=cpu.eip) {
  diff=true;
  Log("different:qemu.eip=0x%x,nemu.eip=0x%x",r.eip,cpu.eip);
}
```

然后在`ics2017/nemu/`下键入`bash runall.sh`进行一件回归测试。

> 测试结果

```
ics@e3b48c25a345:~/ics2017/nemu$ bash runall.sh
NEMU compile OK
compiling testcases...
testcases compile OK
[  add-longlong] PASS!
[           add] PASS!
[           bit] PASS!
[   bubble-sort] PASS!
[         dummy] PASS!
[          fact] PASS!
[           fib] PASS!
[      goldbach] PASS!
[     hello-str] PASS!
[       if-else] PASS!
[     leap-year] PASS!
[    load-store] PASS!
[    matrix-mul] PASS!
[           max] PASS!
[          min3] PASS!
[         mov-c] PASS!
[         movsx] PASS!
[  mul-longlong] PASS!
[        pascal] PASS!
[         prime] PASS!
[    quick-sort] PASS!
[     recursion] PASS!
[   select-sort] PASS!
[         shift] PASS!
[   shuixianhua] PASS!
[        string] PASS!
[  sub-longlong] PASS!
[           sum] PASS!
[        switch] PASS!
[ to-lower-case] PASS!
[       unalign] PASS!
[        wanshu] PASS!
ics@e3b48c25a345:~/ics2017/nemu$
```

（三）阶段三 输入输出

虽然完成了前两个阶段，但是只能进行纯粹的计算，不能进行输入输出，接下来，我们实现CPU与I/O设备的交互。

串口

根据i386手册实现in和out指令。

```
  /* 0xe4 */    IDEXW(in_I2a, in, 1), IDEXW(in_I2a, in, 1), IDEXW(out_a2I, out, 1), IDEXW(out_a
  /* 0xe8 */    IDEX(J, call), IDEX(J, jmp), EMPTY, IDEXW(J, jmp, 1),
  /* 0xec */    IDEXW(in_dx2a, in, 1), IDEXW(in_dx2a, in, 1), IDEXW(out_a2dx, out, 1), IDEXW(ou

make_EHelper(in) {
  rtl_li(&t0, pio_read(id_src->val, id_dest->width));
  operand_write(id_dest, &t0);
  print_asm_template2(in);
#ifdef DIFF_TEST
  diff_test_skip_qemu();
#endif  }
make_EHelper(out) {
  pio_write(id_dest->val, id_src->width, id_src->val);
  print_asm_template2(out);
#ifdef DIFF_TEST
  diff_test_skip_qemu();
#endif  }
```

测试结果

```
make: [/home/ics/ics2017/nexus-am/Makefile.compile:86: klib] Error 2 (ignored)
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

时钟

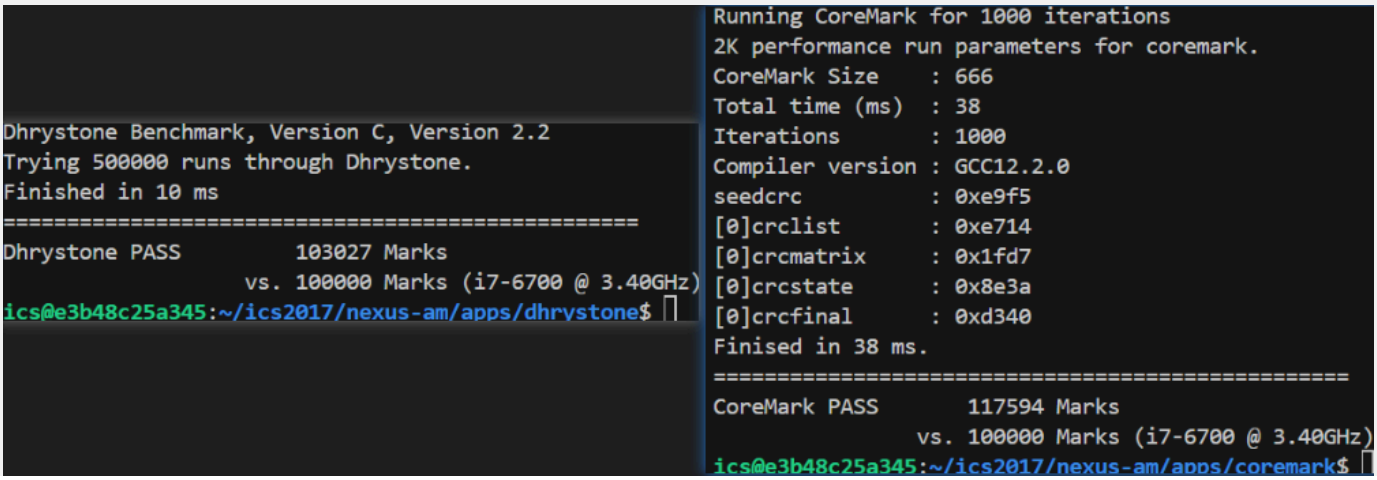
系统启动后经过的毫秒数是当前时间与系统启动时间的差值，实现_uptime()函数。

```
unsigned long _uptime() { return (inl(RTC_PORT)-boot_time); }
```

测试结果

```
make: [/home/ics/ics2017/nexus-am/Makefile.compile:86: klib] Error 2 (ignored)
error: XDG_RUNTIME_DIR is invalid or not set in the environment.
1 second.
2 seconds.
3 seconds.
4 seconds.
5 seconds.
6 seconds.
7 seconds.
8 seconds.
9 seconds.
10 seconds.
11 seconds.
```

我们进行跑分，发现此系统在理论计算性能上超越i7-6700。



键盘

键盘功能的核心在keyboard.c中:

数据端口（0x60）：用于读取或写入键盘的扫描码（8位）。
状态端口（0x64）：通过状态位判断是否有按键事件待处理。
按下（Make Code）：当按键按下时，键盘发送一个8位通码（如`0x2A`表示“A”键按下）。
释放（Break Code）：当按键释放时，发送一个高位置1的断码（如`0xAA`表示“A”键释放）。
队列管理：通过`key_queue`队列缓存按键事件，确保事件不丢失。
I/O端口模拟：通过`i8042_io_handler`处理端口读写，模拟8042芯片的行为，读取状态端口时，检查队列是否有数据，设置状态位`I8042_STATUS_HASKEY_MASK`。读取数据端口时，从队列中取出扫描码并更新队列指针。

通过按键状态数组和队列事件处理来检测多个键同时按下：

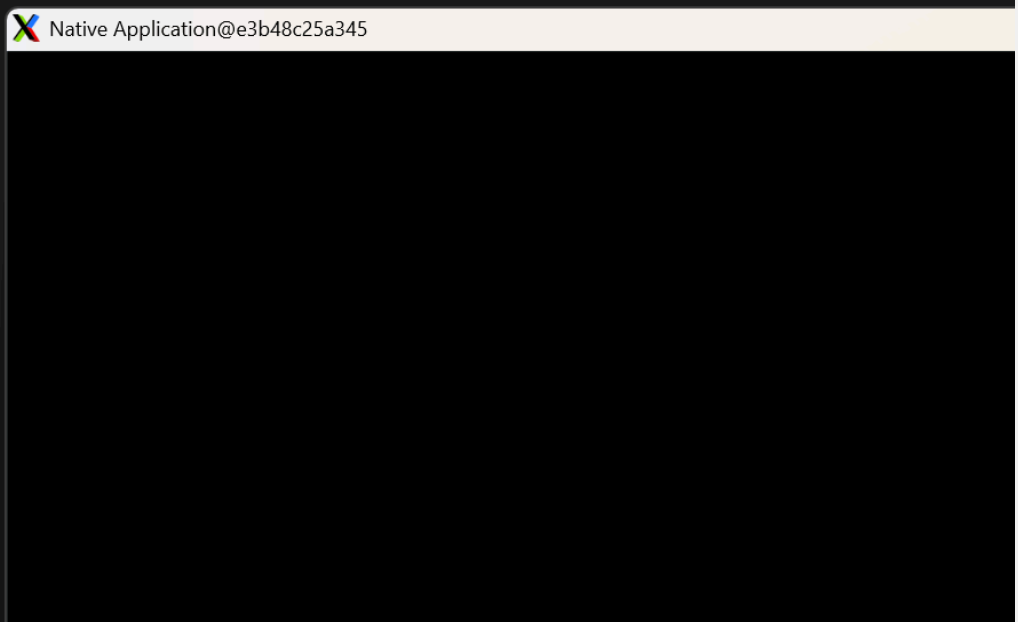
使用数组key_state[256]记录每个按键的实时状态（0=释放，1=按下）。
每次按键`按下/释放`时，更新对应扫描码的数组值。
遍历key_state数组，统计已按下的键数量。
若统计结果≥2，则判定为多键同时按下。
按键事件存入队列，确保事件按顺序处理。
队列长度足够大，通过循环指针实现先进先出。
在读取状态端口时，确保队列中的事件被及时消费，避免遗漏。

通过I/O端口读取PS/2键盘的按键扫描码，返回当前按键的值。

```
int _read_key() {
  if(inb(0x64)){return inb(0x60);}
  else{return _KEY_NONE;}
}
```

测试结果

```
make: [/home/ics/ics2017/nexus-am/Makefile.compile:86: klib] Error 2 (ignored)
Get key: 73 UP down
Get key: 73 UP up
Get key: 75 LEFT down
Get key: 75 LEFT up
Get key: 76 RIGHT down
Get key: 76 RIGHT up
Get key: 74 DOWN down
Get key: 74 DOWN up
Get key: 55 LSHIFT down
Get key: 30 W down
Get key: 30 W up
Get key: 43 A down
Get key: 43 A up
Get key: 44 S down
Get key: 44 S up
Get key: 45 D down
Get key: 45 D up
Get key: 55 LSHIFT up
Get key: 30 W down
Get key: 43 A down
Get key: 43 A up
Get key: 30 W up
```



VGA

添加内存映射I/O。

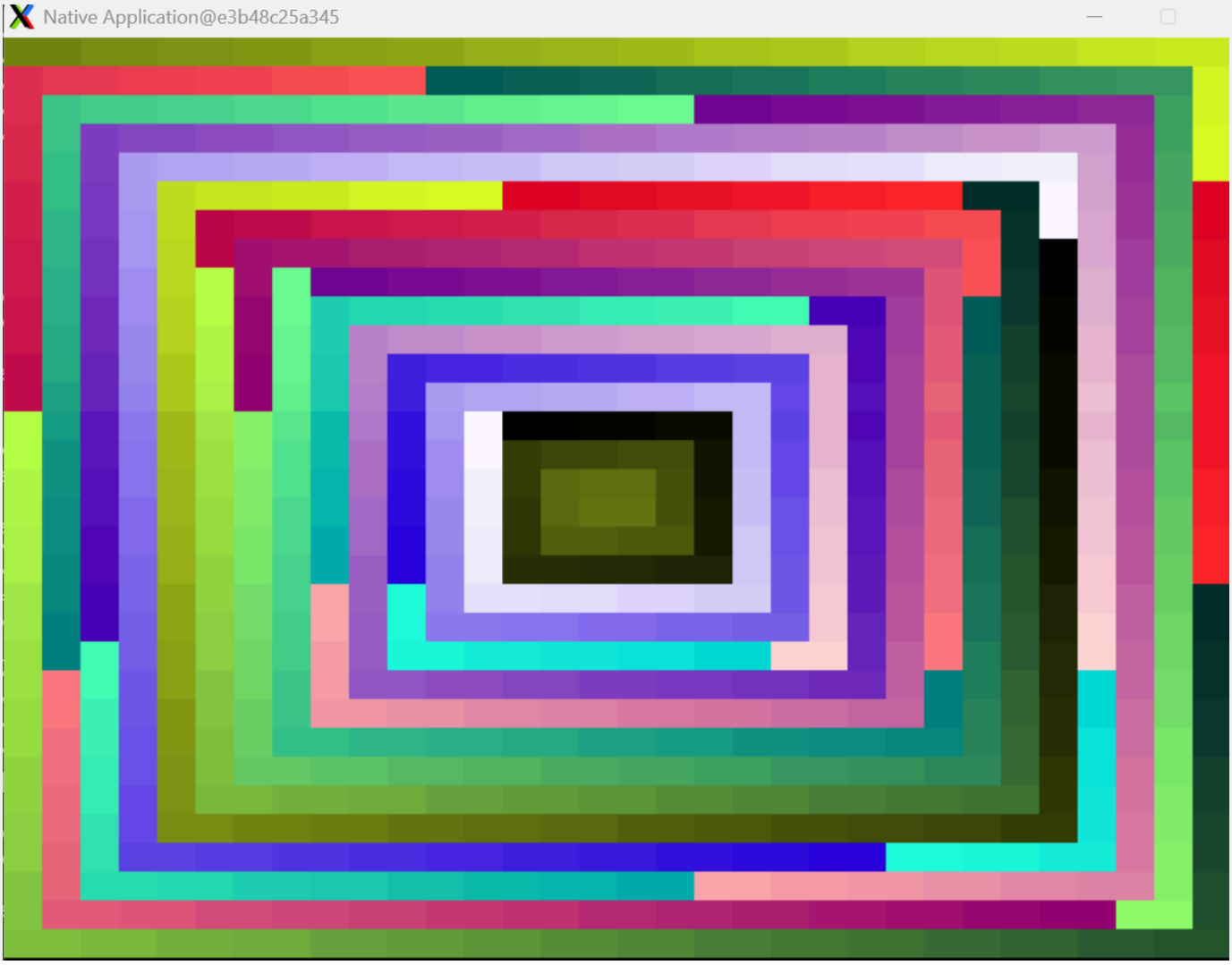
```
uint32_t paddr_read(paddr_t addr, int len){
  // return pmem_rw(addr, uint32_t) & (~0u >> ((4 - len) << 3));
  int r = is_mmio(addr);
  if (r == -1){ return pmem_rw(addr, uint32_t) & (~0u >> ((4 - len) << 3)); }
  else{ return mmio_read(addr, len, r); }
}
void paddr_write(paddr_t addr, int len, uint32_t data){
  // memcpy(guest_to_host(addr), &data, len);
  int r = is_mmio(addr);
  if (r == -1){ memcpy(guest_to_host(addr), &data, len);  }
  else{ mmio_write(addr, len, data, r); }
}
```

实现_draw_rect()函数，进行绘制。

```
void _draw_rect(const uint32_t *pixels, int x, int y, int w, int h) {
  int temp = (w > _screen.width - x) ? _screen.width - x : w;
  int cp_bytes = sizeof(uint32_t) * temp;
  for (int j = 0; j < h && y + j < _screen.height; j++){
    memcpy(&fb[(y + j) * _screen.width + x], pixels, cp_bytes);
    pixels += w;
  }
}
```

另外，要在memory.c中加入`#include "device/mmio.h"`头文件。

测试结果



功能已经实现，我们来运行一下打字小游戏和马里奥。

四、遇到的BUG及解决思路

- 这个错误是因为docker是64位的，系统缺少32位兼容的C库开发文件。通过以下方式解决：

```
ics@e3b48c25a345:~/ics2017/nexus-am/tests/cputest$ make ARCH=x86-nemu ALL=dummy run
Building dummy [x86-nemu]
Building am [x86-nemu]
+ CC arch/x86-nemu/src/asye.c
In file included from /usr/lib/gcc/x86_64-linux-gnu/12/include/stdint.h:9,
                 from /home/ics/ics2017/nexus-am/am/am.h:10,
                 from arch/x86-nemu/src/asye.c:1:
/usr/include/stdint.h:26:10: fatal error: bits/libc-header-start.h: No such file or directory
   26 | #include <bits/libc-header-start.h>
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~~
compilation terminated.
make[3]: *** [/home/ics/ics2017/nexus-am/Makefile.compile:71: /home/ics/ics2017/nexus-am/am/build/x86-nemu//./arch/x86-nemu/src/asye.o] Error 1
make[2]: *** [Makefile:6: all] Error 2
make[1]: *** [/home/ics/ics2017/nexus-am/Makefile.compile:84: am] Error 2
make: [Makefile:13: Makefile.dummy] Error 2 (ignored)
 dummy
```

```
sudo apt update
sudo apt install gcc-multilib
sudo apt install build-essential
```

- 在进行跑分时出现以下错误：`error: XDG_RUNTIME_DIR is invalid or not set in the environment.`，查阅了许多资料，发现是未设置XDG_RUNTIME_DIR环境变量，我们在终端键入`echo $XDG_RUNTIME_DIR`发现输出为空。
  然后键入`id-u`查看用户id，然后去到对应要执行的app目录下键入`export XDG_RUNTIME_DIR=/run/user/1000`，问题解决。

```
ics@e3b48c25a345:~$ echo $XDG_RUNTIME_DIR

ics@e3b48c25a345:~$ id -u
1000
```

  不过我觉得这个好像没有什么影响，因为我在vscode(ssh连接ics-vm)运行时有这个问题，在MobaXterm_Personal_25.0.exe中运行时却没有这个问题，而且，即使我这次在vscode中设置了环境变量，等我退出后下一次打开时，还是有这个问题，这个好像对程序没什么影响。
- 在实现键盘输入检测时，使用inl(0x60)读取数据端口0x60，但PS/2键盘的扫描码是8位数据，应使用inb（读取8位）而非inl（读取32位）。inl会读取4字节数据（32位），可能导致读取到无效数据或触发硬件异常。

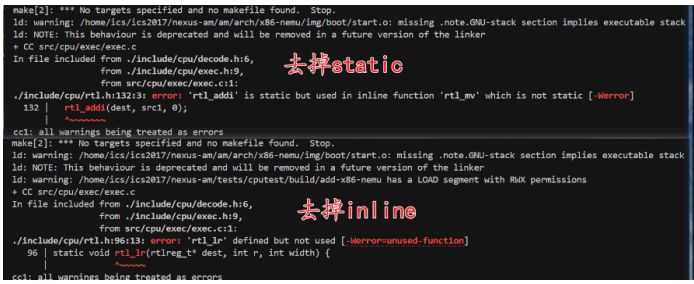
五、必答题

必答题

你需要在实验报告中用自己的语言,尽可能详细地回答下列问题.

- 在 nemu/include/cpu/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,你会看到发生错误.请分别解释为什么会发生这些错误?你有办法证明你的想法吗?
- 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/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

(1) 问题1:在 nemu/include/cpu/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,你会看到发生错误.请分别解释为什么会发生这些错误?



- 去掉static

  static用来控制作用域。inline实际上表示建议内联，并非强制内联，编译器可以忽略。如果想要确保内联，在使用inline的时候要加入static，否则inline不内联的时候就和普通函数在头文件中的定义是一样的，相当于一个普通函数，如果多个c文件都包含rtl.h文件时就会产生歧义。
- 去掉inline

  当去掉inline时，编译器会将该函数生成为一个独立的代码块，并把他放入对应的目标文件中。如果该函数在其他文件中没有被调用，那么编译器会认为该函数是未使用的，并给出相应的警告。所以如果只去掉inline关键字，可能会导致编译器给出 "unused function"的警告信息。
- 去掉static和inline

  当同时去掉static和inline时，该函数会变为一个非静态的非内联函数，会被编译器视为一个普通的函数，并且可以在其他文件中被调用。如果在其他文件中也定义了同名的函数，则会出现重复定义的问题，导致编译错误。

(2) 问题2: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/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.)

Makefile 的工作方式

- **变量管理源文件与编译选项**
  `Makefile` 中通过变量定义关键路径和编译参数：

```
CC = gcc                      # 定义编译器（如 GCC）
CFLAGS = -Wall -g -O0         # 编译选项（警告、调试信息、禁用优化）
OBJ_DIR ?= $(BUILD_DIR)/obj   # 指定编译输出目录
OBJS = $(SRCS:src/%.c=$(OBJ_DIR)/%.o)  # 定义源文件和目标文件之间的映射关系
```

- **可执行文件目标定义**
  最终目标 `nemu` 的依赖关系通过变量表示：

```
NEMU = $(OBJ_DIR)/nemu
$(NEMU): $(OBJS)              # 可执行文件依赖所有 .o 文件
    $(CC) $(OBJS) -o $@       # 链接命令
```

- **分模块管理编译规则**
  `Makefile` 可能通过 `include` 指令引入子目录的规则文件，例如：

```
include Makefile.git     # 引入 git 管理的编译规则
-include $(OBJS:.o=.d)    # 自动包含所有 .d 文件
```

  子模块的Makefile可以定义自己的源文件和依赖关系，主Makefile通过整合这些变量生成全局依赖。
- **隐式规则的重写**
  - **覆盖默认编译规则**
    GNU Make 默认提供隐式规则（如 `.c → .o`），但项目可能通过显式规则添加自定义选项：

```
$(OBJDIR)/%.o: src/%.c  # 重写 .c → .o 的规则，添加头文件依赖和编译选项
    $(CC) $(CFLAGS) -c $< -o $@  # 编译单个 .c 文件为 .o
```

  - **自动依赖生成**
    通过 `gcc -MMD` 生成 `.d` 依赖文件，自动追踪头文件的修改：

```
$(CC) $(CFLAGS) -MMD -c $< -o $@  # 在编译命令中添加依赖生成
-include $(OBJS:.o=.d)  # 包含生成的依赖文件
```

    这样，当 `.h` 文件被修改时，相关 `.o` 会进行重新编译。

编译与链接过程

- **预处理、编译与汇编**
  - **预处理**
    编译器读取 `.c` 文件，处理 `#include` 和宏定义，生成中间文件（`.i` 或 `.ii`）。
  - **编译**
    将预处理后的代码编译为汇编代码（`.s`），再汇编为二进制目标文件（`.o`）：

```
gcc -c src/cpu.c -o build/cpu.o  # 示例：编译 src/cpu.c → build/cpu.o
```

每个 `.c` 文件独立编译为对应的 `.o`，并存储在 `build/` 目录中。

- **链接**
  - **链接器整合目标文件**
    所有 `.o` 文件被链接成最终的可执行文件 `nemu`：

    ```
    gcc build/cpu.o build/device.o ... -o build/nemu
    ```

    链接器解决符号引用（如函数调用），生成可执行代码。
- **依赖关系判断**
  - **时间戳比较**
    `make` 通过文件时间戳判断是否需要重新编译：
    - 若 `.o` 不存在则编译对应的 `.c`。
    - 若 `.c` 或其依赖的 `.h` 文件比 `.o` 新则重新编译 `.c`。
    - 否则跳过，直接使用已有的 `.o`。
  - **使用 `make -n` 查看规则**
    执行 `make -n` 可显示 `make` 的执行计划，但不实际编译：

    ```
    make -n  # 输出所有将要执行的命令

    echo + CC src/cpu/exec/exec.c
    mkdir -p build/obj/cpu/exec/
    gcc -O2 -MMD -Wall -Werror -ggdb -I./include -c -o build/obj/cpu/exec/exec.o src/cpu/e
    ......
    ```

六、总结与感悟

本次实验工作量很大，需要实现的指令非常多，而且涉及到很多的文件，刚开始对于我来说比较困难。通过本次实验，我更加深入地了解了冯诺依曼计算机系统，加强了我查找信息与调试错误的能力，收获很多。

> Opcode表和Group表

One-Byte Opcode Map

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ADD						PUSH	POP	OR						PUSH	2-byte
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	ES	ES	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	CS	escape
1	ADC						PUSH	POP	SBB						PUSH	POP
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	SS	SS	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	DS	DS
2	AND						SEG	DAA	SUB						SEG	DAS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	=ES		Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	=CS	
3	XOR						SEG	AAA	CMP						SEG	AAS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	=SS		Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	=CS	
4	INC general register								DEC general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
5	PUSH general register								POP into general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
6	PUSHA	POPA	BOUND Gv,Ma	ARPL Ew,Rw	SEG =FS	SEG =GS	Operand Size	Address Size	PUSH Ib	IMUL GvEvIv	PUSH Ib	IMUL GvEvIv	INSB Yb,DX	INSW/D Yb,DX	OUTSB Dx,Xb	OUTSW/D DX,Xv
7	Short displacement jump of condition (Jb)								Short-displacement jump on condition(Jb)							
	JO	JNO	JB	JNB	JZ	JNZ	JBE	JNBE	JS	JNS	JP	JNP	JL	JNL	JLE	JNLE
8	Immediate Grpl			Grpl	TEST		XCNG		MOV				MOV	LEA	MOV	POP
	Eb,Ib	Ev,Iv		Ev,Iv	Eb,Gb	Ev,Gv	Eb,Gb	Ev,Gv	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	Ew,Sw	Gv,M	Sw,Ew	Ev
9	NOP	XCHG word or double-word register with eAX							CBW	CWD	CALL Ap	WAIT	PUSHF Fv	POPF Fv	SAHF	LAHF
		eCX	eDX	eBX	eSP	eBP	eSI	eDI								
A	MOV				MOVSB	MOVSW/D	CMPSB	CMPSW/D	TEST		STOSB	STOSW/D	LODSB	LODSW/D	SCASB	SCASW/D
	AL,Ob	eAX,Ov	Ob,AL	Ov,eAX	Xb,Yb	Xv,Yv	Xb,Yb	Xv,Yv	AL,Ib	eAX,Iv	Yb,AL	Yv,eAX	AL,Xb	eAX,Xv	AL,Xb	eAX,Xv
B	MOV immediate byte into byte register								MOV immediate word or double into word or double register							
	AL	CL	DL	BL	AH	CH	DH	BH	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
C	Shift Grp2		RET near		LES	LDS	MOV		ENTER Iw,Ib	LEAVE	RET far		INT 3	INT Ib	INTO	IRET
	Eb,Ib	Ev,Iv	Iw		Gv,Mp	Gv,Mp	Eb,Ib	Ev,Iv			Iw					
D	Shift Grp2				AAM	AAD		XLAT	ESC(Escape to coprocessor instruction set)							
	Eb,1	Ev,1	Eb,CL	Ev,CL												
E	LOOPNE Jb	LOOPE Jb	LOOP Jb	JCXZ Jb	IN		OUT		CALL Av	JMP			IN		OUT	
					AL,Ib	eAX,Ib	Ib,AL	Ib,eAX		Jv	Ap	Jb	AL,DX	eAX,DX	DX,AL	DX,eAX
F	LOCK		REPNE	REP REPE	HLT	CMC	Unary Grp3		CLC	STC	CLI	STI	CLD	STD	INC/DEC Grp4	Indirct Grp5
							Eb	Ev								

Group

mod	nnn	R/M

	000	001	010	011	100	101	110	111
1	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
2	ROL	ROR	RCL	RCR	SHL	SHR		SAR
3	TEST Ib/Iv		NOT	NEG	MUL AL/eAX	IMUL AL/eAX	DIV AL/eAX	IDIV AL/eAX
4	INC Eb	DEC Eb						
5	INC Ev	DEC Ev	CALL Ev	CALL eP	JMP Ev	JMP Ep	PUSH Ev	