

# 计算机系统设计 PA2

---

- 姓名：管昀玫
- 学号：2013750
- 专业：计算机科学与技术

## 1 概述

---

### 1.1 实验目的

- 熟悉CPU执行指令的步骤
- 探究程序运行时的环境
- 探究diff-test
- 探究模拟输入输出

### 1.2 实验内容

- 在nemu实现部分指令，运行第一个C程序dummy
- 实现更多的指令，能运行所有的 `cputest`
- 实现输入输出，能运行打字小游戏

### 1.3 环境配置

在PA2第一个程序实现之前，需要先配置好环境。需要使用 `cd` 切换至用户主目录下，执行 `vim .bashrc` 修改该环境初始化文件，在文件末尾加入以下三行：

```
export NEMU_HOME=~/.自己的路径/nemu
export AM_HOME=~/.自己的路径/nexus-am
export NAVY_HOME=~/.自己的路径/navy-apps
```

需要让项目默认编译到 `x86-nemu` 中的AM中：

```
// nexus-am/Makefile.check
ARCH ?= x86-nemu

ARCHS = $(shell ls $(AM_HOME)/am/arch/)
```

然后在 `nexus-am/tests/cputest` 中执行

```
make ALL=xxx run
```

## 2 阶段一 运行第一个客户端程序

---

## 2.0 RTL语言

NEMU 使用 RTL(寄存器传输语言)来描述 x86 指令的行为。这样做的好处是可以提高代码的复用率,使得指令模拟的实现更加规整。同时 RTL 也可以作为一种 IR(中间表示)语言,将来可以很方便地引入即时编译技术对 NEMU进行优化。

在NEMU中, RTL寄存器有:

- x86 的八个通用寄存器(在 nemu/include/cpu/reg.h 中定义)
- id\_src, id\_src2和id\_dest中的访存地址addr和操作数内容val(在nemu/include/cpu/decode.h中定义)。从概念上看,它们分别与 MAR 和 MDR 有异曲同工之妙
- 临时寄存器 t0~t3(在 nemu/src/cpu/decode/decode.c 中定义)
- 0 寄存器 tzero(在 nemu/src/cpu/decode/decode.c 中定义), 它只能读出 0, 不能写入

RTL 基本指令包括:

- 立即数读入 rtl\_li
- 算术运算和逻辑运算,包括寄存器-寄存器类型 rtl\_(add|sub|and|or|xor|shl|shr|sar|slt|sltu)和立即数-寄存器类型 rtl\_(add|sub|and|or|xor|shl|shr|sar|slt|sltu)i
- 内存的访存 rtl\_lm 和 rtl\_sm
- 通用寄存器的访问 rtl\_lr\_(b|w|l)和 rtl\_sr\_(b|w|l)

RTL伪指令:

- 带宽度的通用寄存器访问 rtl\_lr 和 rtl\_sr
- EFLAGS 标志位的读写 rtl\_set\_(CF|OF|ZF|SF|IF)和 rtl\_get\_(CF|OF|ZF|SF|IF)
- 其它常用功能,如数据移动 rtl\_mv,符号扩展 rtl\_sext 等


## 2.1 了解需要填写的指令

PA2的第一个任务就是运行第一个简单的C程序。代码在 nexus-am/tests/cputest/tests/dummy.c, 它什么都不做直接返回。在 nexus-am/tests/cputest 目录下键入 make ARCH=x86-nemu ALL=dummy run, 启动NEMU运行。

其输出如下所示:

```
[src/monitor/monitor.c,65,load_img] The image is /mnt/d/LessonProjects/SystemDesign/PA-pal/PA-pal/nexus-am/tests/cputest/build/dummy-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 11:35:44, Apr 6 2023
For help, type "help"
(nemu) c
invalid opcode(eip = 0x0010000a): e8 01 00 00 00 90 f3 0f ...

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)
```

根据提示, 可以看到:

1. 在eip=0x0010000a这个位置有指令没有实现
2. 操作码e8的指令没有实现

查看反汇编文件，获取详细信息：

在 /nexus-am/tests/cputest/build 目录下，找到 dummy-x86-nemu.txt，打开它查看：

Disassembly of section .text:

```
00100000 <_start>:
 100000:  bd 00 00 00 00      mov     $0x0,%ebp
 100005:  bc 00 7c 00 00      mov     $0x7c00,%esp
 10000a:  e8 01 00 00 00      call    100010 <_trm_init>
 10000f:  90                  nop

00100010 <_trm_init>:
 100010:  f3 0f 1e fb      endbr32
 100014:  55                push    %ebp
 100015:  89 e5             mov     %esp,%ebp
 100017:  83 ec 08          sub     $0x8,%esp
 10001a:  e8 05 00 00 00      call    100024 <main>
 10001f:  d6                (bad)
 100020:  eb fe            jmp     100020 <_trm_init+0x10>
 100022:  66 90             xchg    %ax,%ax

00100024 <main>:
 100024:  f3 0f 1e fb      endbr32
 100028:  31 c0             xor     %eax,%eax
 10002a:  c3                ret
```

查找i386手册的opcode table，寻找e8对应的指令：

XLAT		ESC(Escape to		
OUT	CALL	JNP		
b,AL	Ib,eAX	Av	Jv	Ap
Unary Grp3		CLC	STC	CLI
Eb	Ev			

进一步的，了解A和v的含义：

A Direct address; the instruction has no modR/M byte; the address of the operand is encoded in the instruction; no base register, index register, or scaling factor can be applied; e.g., far JMP (EA).

v Word or double word, depending on operand size attribute.

A的含义是地址，v为单字或双字，取决于操作空间的大小，这就是我们调用函数时使用的call指令。

在PA2中，只需要实现CALL rel32即可。%eip 的跳转可以通过将 decoding.is\_jump 设为 1,并将 decoding.jump\_eip 设为跳转目标地址来实现，这时在 update\_eip()函数中会把跳转目标地址作为新的%eip,而不是顺序意义下的下一条指令的地址。

需要实现更多的指令：

- Data Movement Instructions: mov, push, pop, leave, cld(在 i386 手册中为 cdq), movsx, movzx
- Binary Arithmetic Instructions: add, inc, sub, dec, cmp, neg, adc, sbb, mul, imul, div, idiv
- Logical Instructions: not, and, or, xor, sal(shl), shr, sar, setcc, test
- Control Transfer Instructions: jmp, jcc, call, ret
- Miscellaneous Instructions: lea, nop

## 2.2 Opcode-table

执行辅助函数 统一通过宏 `def_EHelper` (在 `nemu/include/cpu/exec.h` 中定义) 来定义:

```
#define make_EHelper(name) void concat(exec_, name) (vaddr_t *eip)
typedef void (*EHelper) (vaddr_t *);
```

执行辅助函数通过 RTL 指令来描述指令真正的执行功能。

`exec.c` 中间持有一个最关键的表格, 也就是 `opcode_table`。 `opcode_table` 是一个 `opcode_entry` 类型结构体的数组。 `opcode_entry` 中间持有两个函数指针和设置操作数长度的变量, 而函数指针就是处理一个 opcode 所需要执行的解码逻辑和执行逻辑。

`exec.c` 中间 `idex` 函数会调用 `opcode_entry` 中间两个函数指针, 来 `decode` 和 `execute` 指令, 试验框架已经完成了所有 `mov` 指令的处理, 我们剩下需要做的事情就是完成其他的指令的填写。

首先需要在 `nemu/src/cpu/exec/all-instr.h` 里补全指令:

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

然后再 `nemu/src/cpu/exec/exec.h` 里对 `opcode_entry opcode_table [512]` 进行相应的更改:

```
/* 0x30 */    IDEXW(G2E, xor, 1), IDEX(G2E, xor), IDEXW(E2G, xor, 1),
IDEX(E2G, xor),
/* 0x34 */    IDEXW(I2a, xor, 1), IDEX(I2a, xor), EMPTY, 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),
/* 0x58 */    IDEX(r, pop), IDEX(r, pop), IDEX(r, pop), IDEX(r, pop),
/* 0x5c */    IDEX(r, pop), IDEX(r, pop), IDEX(r, pop), IDEX(r, pop),
/* 0xc0 */    IDEXW(gp2_Ib2E, gp2, 1), IDEX(gp2_Ib2E, gp2), EMPTY, EX(ret),
/* 0xe8 */    IDEX(J, call), IDEX(J, jmp), EMPTY, IDEXW(J, jmp, 1),
```

其中, `ID` 和 `EX` 分别表示了译码和执行两个阶段, `W` 表示需要取字。

## 2.3 EFlags

由于在程序执行过程中都是使用 RTL 来实现该过程, 所以我们需要在 `nemu/include/cpu/reg.h` 中先补充 EFlags 标志位, 代码如下所示:

```

struct bs {
    unsigned int CF:1;
    unsigned int one:1;
    unsigned int :4;
    unsigned int ZF:1;
    unsigned int SF:1; // bit 0 ~ 7
    unsigned int :1;
    unsigned int IF:1;
    unsigned int :1;
    unsigned int OF:1;
    unsigned int :20;
} eflags;

```

这些 Eflags 将在 `nemu/src/monitor/monitor.c` 中进行初始化, 并在 `nemu/src/cpu/arith.c` 中进行标志位的设置进行减法计算。

需要补充 `restart()` 函数和 `eflags_modify()` 函数:

`restart` 函数使用 `memcpy` 将 `eflags` 设置为 `0x00000002H`:

```

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
}

```

`eflags_modify` 为计算减法并设置 `eflags` 的值:

```

static inline void eflags_modify() {
    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);
}

```

实现 `DopHelper` 调用取指进行取字:

```

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 = ???
    */
//TODO();
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
}

```

push和pop:

push主要作用是修改栈顶，将指针src1的内容写入栈；pop函数是将rtl\_pop读取的数据写入通用寄存器中。

```

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

```

```

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

```

更新标识符操作:

```

static inline void rtl_eq0(rtlreg_t* dest, const rtlreg_t* src1) {
    // dest <- (src1 == 0 ? 1 : 0)
    // TODO();
    rtl_slui(dest, src1, 1);
}

static inline void rtl_eqi(rtlreg_t* dest, const rtlreg_t* src1, int imm) {
    // dest <- (src1 == imm ? 1 : 0)
    // TODO();
    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)
    // TODO();
    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]
    // TODO();
    rtl_shri(dest, src1, width*8-1);
    rtl_andi(dest, dest, 0x1);
}

```

根据提示更新 ZF 位:

```

static inline void rtl_update_ZF(const rtlreg_t* result, int width) {
    // eflags.ZF <- is_zero(result[width * 8 - 1 .. 0])
    // TODO();
    rtl_andi(&t0, result, (0xffffffffu >> (4-width)*8));
    rtl_eq0(&t0, &t0);
    rtl_set_ZF(&t0);
}

```

更新 SF 位:

```

static inline void rtl_update_SF(const rtlreg_t* result, int width) {
    // eflags.SF <- is_sign(result[width * 8 - 1 .. 0])
    // TODO();
    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);
}

```

EFlags 寄存器标志位的读写函数:

```

#define make_rtl_arith_logic(name) \
    static inline void concat(rtl_, name) (rtlreg_t* dest, const rtlreg_t* src1, \
    const rtlreg_t* src2) { \
        *dest = concat(c_, name) (*src1, *src2); \
    } \
    static inline void concat3(rtl_, name, i) (rtlreg_t* dest, const rtlreg_t* \
    src1, int imm) { \
        *dest = concat(c_, name) (*src1, imm); \
    }

```

## 2.4 指令实现

### 2.4.1 call

call在指令中的位置为E8，详细如下所示：

FF	/3	CALL m16:16	5 + ts	Call to task
E8	cd	CALL rel32	7+m	Call near, displacement relative to next instruction
FF	/2	CALL r/m32	7+m/10+m	Call near, indirect
9A	cp	CALL ptr16:32	17+m, pm=34+m	Call intersegment, to pointer given
9A	cb	CALL ptr16:32	om=52+m	Call gate, same privilege

#### Operation

```
IF rel16 or rel32 type of call
THEN (* near relative call *)
  IF OperandSize = 16
  THEN
    Push(IP);
    EIP ← (EIP + rel16) AND 0000FFFFH;
  ELSE (* OperandSize = 32 *)
    Push(EIP);
    EIP ← EIP + rel32;
  FI;
FI;
```

call指令读取需要压栈的EIP值，用rtl\_push压栈，最后设置跳转。

在之前我们已经修改了 all-instr.h 和 opcode\_table，所以接下来需要在 control.c 文件中具体实现 make\_EHelper(call)：

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

call是一个形指令，需要编写 decode.c 中的译码函数 make\_DHelper(J)，在里面调用 decode\_op\_SI(eip, id\_dest, false); 来实现立即数的读取，并更新 jump\_eip。

```
make_DHelper(J) {
    decode_op_SI(eip, id_dest, false);
    // the target address can be computed in the decode stage
    decoding.jump_eip = id_dest->simm + *eip;
}
```

### 2.4.2 push

同上，修改 all-instr.h 和 opcode\_table。

由于 push 是数据移动指令，需要在 data-mov.c 中实现。调用已实现的 rtl\_push 执行函数写进栈中即可。



```
make_EHelper(push) {
    //TODO();
    rtl_push(&id_dest -> val);
    print_asm_template1(push);
}
```

## 2.4.3 pop

同上，调用 `rtl_pop` 执行函数进入读栈，并把读取的数据写到通用寄存器中。

与 `push` 指令相同，`pop` 同样也是数据移动指令，需要在 `data-mov.c` 中实现：

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

## 2.3.3 sub

### Flags Affected

OF, SF, ZF, AF, PF, and CF as described in Appendix C

在 `sub` 指令中，OF/SF/ZF/AF/PF/CF 标志位受到影响。各标志位的作用如下所示：

- CF(bit 0) [进位标志]: 若算术操作产生的结果在最高有效位(most-significant bit)发生进位或借位则将其置 1，反之清零。这个标志指示无符号整型运算的溢出状态。
- PF(bit 4) [奇偶标志]: PF 标志位位于 EFLAGS 寄存器的第 2 位，用于表示结果的二进制表示中 1 的个数是否为偶数。对于 `sub` 指令来说，当结果的二进制表示中 1 的个数为偶数时，PF 被设置为 1，否则为 0。
- AF(bit 4) [辅助进位标志]: AF 标志位位于 EFLAGS 寄存器的第 4 位，用于表示低 4 位的进位或借位情况。当低 4 位有进位或借位时，AF 被设置为 1，否则为 0。
- ZF(bit 6) [零标志]: 当结果为零时，ZF 被设置为 1，否则为 0。
- SF(bit 7) [符号标志]: 该标志被设置为有符号整型的最高有效位。当结果为负数时，SF 被设置为 1，否则为 0。
- OF(bit 11) [溢出标志]: 如果整型结果是较大的正数或较小的负数，并且无法匹配目的操作数时将该位置 1，反之清零。用于表示有符号数的运算结果是否超出了其数据类型所能表示的范围。

查看 i386 手册，获取 `sub` 指令的详细 opcode 字段解释：

### 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`: 表示8位立即数
- `iw`: 表示16位立即数
- `id`: 表示32位立即数
- `/r`: 表示后面有一个ModR/M字节, 且其中的reg/opcode字段被解释为寄存器编码
- `/digit`: digit 为 0~7 中一个数字 (/0), 表示操作码后跟一个 ModR/M 字节, 并且 reg/opcode 字段被解释为扩展 opcode, 取值为 digit。

为了实现sub指令, 需要修改 `arith.c` 中的函数:

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

然后填写opcode\_table: opcode=0x2D, 译码函数: `make_DHelper(I2a)`, 执行函数:

`make_EHelper(sub)`, 操作数宽度为 2 或 4, 定义为 `IDEX(I2a,sub)`。

```
/* 0x2c */    IDEXW(I2a, sub, 1), IDEX(I2a, sub), EMPTY, EMPTY,
```

sub 指令具有较多的不同形式, 由于其执行阶段相同, 译码函数不同。在实现执行函数后, 只需再根据不同形式设定译码函数和操作数宽度。

- 0x29/r 译码函数: G2E, 定义为 `IDEX(G2E,sub)`
- 0x2B/r 译码函数: E2G, 定义为 `IDEX(E2G,sub)`

```
/* 0x28 */    EMPTY, IDEX(G23, sub), EMPTY, IDEX(E2G, sub),
```

0x80、0x81 与 0x83 的 sub 需要进行 opcode 的扩展。

查看 `make_group` 函数, 该函数用于处理扩展 opcode 的情况。

在 `opcode_table_gp` 中存储各 `opcode_entry`, 并在运行 `make_EHelper(gp)` 时使用 `idex(eip, opcode_table_gp1[decoding.ext_opcode])` 作为进一步的译码-执行函数。

因为 sub 的 `ext_opcode=5`, 所以在 `opcode_table_gp1[5]` 处填写 `EX(sub)` 执行函数。

```
/* 0x80, 0x81, 0x83 */
make_group(gp1,
    EX(add), EX(or), EX adc), EX(sbb),
    EX(and), EX(sub), EX(xor), EX(cmp))
```

### 2.3.4 xor指令

## 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
```

### Description

XOR computes the exclusive OR of the two operands. Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same. The answer replaces the first operand.

### Flags Affected

CF = 0, OF = 0; SF, ZF, and PF as described in Appendix C; AF is undefined

步骤同上。首先修改 `all-instr.h`：

```
make_EHelper(xor);
```

然后修改 `opcode_table[]`

```
/* 0x30 */ IDExW(G2E, xor, 1), IDEx(G2E, xor),
IDExW(E2G, xor, 1), IDEx(E2G, xor),
/* 0x34 */ IDExW(I2a, xor, 1), IDEx(I2a, xor), EMPTY,
EMPTY,
```

由于 `xor` 是逻辑运算指令，需要在 `logic.c` 中实现

```
make_EHelper(xor) {
    // TODO();
    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);
}
```

2.3.5 ret指令

RET — Return from Procedure			
Opcode	Instruction	Clocks	Description
C3	RET	10+m	Return (near) to caller
CB	RET	18+m,pm=32+m	Return (far) to caller, same privilege
CB	RET	pm=68	Return (far), lesser privilege, switch stacks
C2 iw	RET imm16	10+m	Return (near), pop imm16 bytes of parameters
CA iw	RET imm16	18+m,pm=32+m	Return (far), same privilege, pop imm16 bytes
CA iw	RET imm16	pm=68	Return (far), lesser privilege, pop imm16 bytes

步骤同上。首先修改 `all-instr.h`：

```
make_EHelper(ret);
```

然后修改 `opcode_table[]`

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

由于 `xor` 是逻辑运算指令，需要在 `logic.c` 中实现

```
make_EHelper(ret) {
    // TODO();
    rtl_pop(&t2);
    decoding.jump_eip = t2;
    decoding.is_jump = 1;
    print_asm("ret");
}
```

2.3.6 endbr32指令

ENDBR32 — Terminate an Indirect Branch in 32-bit and Compatibility Mode				
Opcode/Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 1E FB ENDBR32	ZO	V/V	CET_IBT	Terminate indirect branch in 32 bit and compatibility mode.

Instruction Operand Encoding ¶

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA	NA

Description ¶

Terminate an indirect branch in 32 bit and compatibility mode.

该指令由三条指令组成，所以PC = PC + 3

在 `all-instr.h` 中添加：

```
make_EHelper(endbr32)
```

修改 `opcode_table[]`

```
/* 0xf0 */ EMPTY, EMPTY, EMPTY, EXW(endbr, 3),
```

在 `special.c` 中实现：

```
make_EHelper(endbr32)
{
    decinfo.seq_pc += 3;
    print_asm("endbr32");
}
```

## 2.5 实验结果

实现所有的指令之后，再次执行`make ARCH=x86-nemu ALL=dummy run`：

```
myrrolinz@Myrrolinz:/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nexus-am/tests/cputest$ make ARCH=x86-nemu ALL=dummy run
Building dummy [x86-nemu]
make[1]: Warning: File 'Makefile.dummy' has modification time 878 s in the future
Building am [x86-nemu]
make[3]: Warning: File '/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nexus-am/am/build/x86-nemu/./arch/x86-nemu/src/ioe.d' has modification time 306 s in the future
make[3]: warning: Clock skew detected. Your build may be incomplete.
make[2]: *** No targets specified and no makefile found. Stop.
make[2]: Warning: File 'build/obj/monitor/monitor.d' has modification time 732 s in the future
[src/monitor/diff-test/diff-test.c,96,init_difftest] Connect to QEMU successfully
[src/monitor/monitor.c,65,load_img] The image is /mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nexus-am/tests/cputest/build/dummy-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 18:22:48, Apr 12 2023
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x0010001f
(nemu) |
```

### 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?

答：堆和栈都是程序运行时的动态内存分配，它们不会在可执行文件中存储，因为它们的大小和内容在程序运行时才能确定。程序运行时需要用到堆和栈，所以它们被分配在程序的运行时内存中。

栈是一种自动分配和释放的内存，用于存储程序的局部变量、函数参数和函数返回值等。当程序运行到一个函数时，会在栈中为该函数分配一块内存空间，当函数返回时，这块内存会被自动释放。栈的大小由系统自动管理，并根据程序的需要进行动态分配和释放。

堆是另一种动态内存分配的机制，程序可以通过调用库函数（如 `malloc` 和 `free`）来在堆中动态地分配和释放内存。堆的大小由程序员控制，但是由于堆中的内存需要手动释放，所以需要特别注意内存泄漏等问题。

当程序运行时，操作系统会为程序分配一块虚拟内存，其中包括可执行文件中的代码和数据，以及堆和栈。当程序需要访问堆和栈时，操作系统会根据程序的需求在虚拟内存中分配相应的空间，并将其映射到物理内存中。这样，程序就可以访问堆和栈了。

## 3 阶段二 运行更多的程序

### 3.1 了解需要填写的指令

```
make_EHelper(mov);
make_EHelper(inv);
make_EHelper(call);
make_EHelper(call_rm);
make_EHelper(push);
make_EHelper(pop);
make_EHelper(sub);
make_EHelper(xor);
make_EHelper(ret);
```

```
make_EHelper(operand_size);  
make_EHelper(nemu_trap);
```

```
make_EHelper(endbr);
```

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

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

```
make_EHelper(leave);  
make_EHelper(cld);  
make_EHelper(cwtl);  
make_EHelper(movsx);  
make_EHelper(movzx);
```

```
make_EHelper(jmp);  
make_EHelper(jmp_rm);  
make_EHelper(jcc);
```

```
make_EHelper(lea);  
make_EHelper(nop);
```

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

```
make_EHelper(lidt);  
make_EHelper(int);
```

```
make_EHelper(pusha);  
make_EHelper(popa);  
make_EHelper(iret);
```

```
make_EHelper(mov_store_cr);
```

## 3.2 指令实现

### 3.2.1 and

在i386手册中为CBW和CWD:

1. CBW: AL符号位扩展至AH
2. CWD: AX的符号位扩展至DX

```
// logic.c
make_EHelper(and) {
    // TODO();
    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);
}

// exec.c
/* 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,
```

### 3.2.2 push & pushl

pushl的opcode如下:

FF	/6	PUSH m16	5	Push memory word
FF	/6	PUSH m32	5	Push memory dword
50	+ /r	PUSH r16	2	Push register word
50	+ /r	PUSH r32	2	Push register dword
6A		PUSH imm8	2	Push immediate byte
68		PUSH imm16	2	Push immediate word
68		PUSH imm32	2	Push immediate dword
0E		PUSH CS	2	Push CS
16		PUSH SS	2	Push SS
1E		PUSH DS	2	Push DS
06		PUSH ES	2	Push ES
0F	A0	PUSH FS	2	Push FS
0F	A8	PUSH GS	2	Push GS

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

push:

```
/* 0x68 */ IDEX(I, push), IDEX(I_E2G, imul3), IDEXW(push_SI, push, 1),
IDEX(SI_E2G, imul3),
```

### 3.2.3 lea

根据opcode修改 exec.c :

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

### 3.2.4 xchg (NOP)

nop实际上什么都不做。其opcode为90:

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

### 3.2.5 SETcc

SETcc指令是一种根据条件设置一个字节数据值的指令，其中cc表示条件代码，SETcc指令会根据特定的条件设置一个字节的数值为1或0，具体的条件代码和作用如下：

指令	条件代码	描述
SETA/SETNBE	CF=0, ZF=0	如果无符号操作数大于，则设置为1，否则设置为0。
SETAE/SETNB/SETNC	CF=0	如果无符号操作数大于等于，则设置为1，否则设置为0。
SETB/SETNAE	CF=1	如果无符号操作数小于，则设置为1，否则设置为0。
SETBE/SETNA	CF=1, ZF=1	如果无符号操作数小于等于，则设置为1，否则设置为0。
SETE/SETZ	ZF=1	如果操作数相等，则设置为1，否则设置为0。
SETG/SETNLE	ZF=0, SF=OF	如果有符号操作数大于，则设置为1，否则设置为0。
SETGE/SETNL	SF=OF	如果有符号操作数大于等于，则设置为1，否则设置为0。
SETL/SETNGE	SF≠OF	如果有符号操作数小于，则设置为1，否则设置为0。
SETLE/SETNG	ZF=1, SF≠OF	如果有符号操作数小于等于，则设置为1，否则设置为0。
SETNE/SETNZ	ZF=0	如果操作数不相等，则设置为1，否则设置为0。
SETO	OF=1	如果发生溢出，则设置为1，否则设置为0。
SETNO	OF=0	如果没有发生溢出，则设置为1，否则设置为0。
SETP/SETPE	PF=1	如果结果的二进制表示中1的个数为偶数，则设置为1，否则设置为0。
SETNP/SETPO	PF=0	如果结果的二进制表示中1的个数为奇数，则设置为1，否则设置为0。



SETcc指令的操作数是一个寄存器或内存位置，指令会将1或0写入该位置，其中cc表示条件代码，每个条件代码都有特定的作用，可以用于执行基于条件的操作或控制流。SETcc指令通常与CMP或TEST等比较或测试指令配合使用，用于根据比较结果设置一个字节的数值。

### SETcc — Byte Set on Condition

Opcode	Instruction	Clocks	Description
0F 97	SETA r/m8	4/5	Set byte if above (CF=0 and ZF=0)
0F 93	SETAE r/m8	4/5	Set byte if above or equal (CF=0)
0F 92	SETB r/m8	4/5	Set byte if below (CF=1)
0F 96	SETBE r/m8	4/5	Set byte if below or equal (CF=1 or (ZF=1)
0F 92	SETC r/m8	4/5	Set if carry (CF=1)
0F 94	SETE r/m8	4/5	Set byte if equal (ZF=1)
0F 9F	SETG r/m8	4/5	Set byte if greater (ZF=0 or SF=OF)
0F 9D	SETGE r/m8	4/5	Set byte if greater or equal (SF=OF)
0F 9C	SETL r/m8	4/5	Set byte if less (SF=OF)
0F 9E	SETLE r/m8	4/5	Set byte if less or equal (ZF=1 and SF=OF)
0F 96	SETNA r/m8	4/5	Set byte if not above (CF=1)
0F 92	SETNAE r/m8	4/5	Set byte if not above or equal (CF=1)
0F 93	SETNB r/m8	4/5	Set byte if not below (CF=0)
0F 97	SETNBE r/m8	4/5	Set byte if not below or equal (CF=0 and ZF=0)
0F 93	SETNC r/m8	4/5	Set byte if not carry (CF=0)
0F 95	SETNE r/m8	4/5	Set byte if not equal (ZF=0)
0F 9E	SETNG r/m8	4/5	Set byte if not greater (ZF=1 or SF=OF)
0F 9C	SETNGE r/m8	4/5	Set if not greater or equal (SF=OF)
0F 9D	SETNL r/m8	4/5	Set byte if not less (SF=OF)
0F 9F	SETNLE r/m8	4/5	Set byte if not less or equal (ZF=1 and SF=OF)
0F 91	SETNO r/m8	4/5	Set byte if not overflow (OF=0)
0F 9B	SETNP r/m8	4/5	Set byte if not parity (PF=0)
0F 99	SETNS r/m8	4/5	Set byte if not sign (SF=0)
0F 95	SETNZ r/m8	4/5	Set byte if not zero (ZF=0)
0F 90	SETO r/m8	4/5	Set byte if overflow (OF=1)
0F 9A	SETP r/m8	4/5	Set byte if parity (PF=1)
0F 9A	SETPE r/m8	4/5	Set byte if parity even (PF=1)
0F 9B	SETPO r/m8	4/5	Set byte if parity odd (PF=0)
0F 98	SETS r/m8	4/5	Set byte if sign (SF=1)
0F 94	SETZ r/m8	4/5	Set byte if zero (ZF=1)

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

OF时涉及到两字节opcode。程序先在 0x0F 处执行 `make_EHelper(2byte_esc)`，在该函数中再确定其正确的两字节opcode编码，并调用 `idex(eip, &opcode_table[opcode])` 进行指令的进一步译码与执行。

```
/* 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),
```

还需实现 `cc.c` 中的 `rtl_setcc` 函数：

```
// TODO: Query EFLAGS to determine whether the condition code is satisfied.
// dest <- ( cc is satisfied ? 1 : 0)
switch (subcode & 0xe) {
    case CC_0:
        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");
}

```

### 3.2.6 MOVSX & MOVZX

需要注意的是，`id_src` 和 `id_dest` 两者宽度不同，在进入函数的时候会重新调整 `id_dest` 的宽度。在填写译码函数的时候按照 `id_src` 的宽度填写即可。

```

/* 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),

//data-mov.c
make_EHelper(movsx) {
    id_dest->width = decoding.is_operand_size_16 ? 2 : 4;
    rtl_sext(&t2, &id_src->val, id_src->width);
    operand_write(id_dest, &t2);
    print_asm_template2(movsx);
}

```

```
make_EHelper(movzx) {
    id_dest->width = decoding.is_operand_size_16 ? 2 : 4;
    operand_write(id_dest, &id_src->val);
    print_asm_template2(movzx);
}
```

### 3.2.7 JCC

即有条件跳转指令。Jcc 指令会检查相应的标志位，如果标志位符合指定的条件，则跳转到目标地址；否则，继续执行下一条指令。Jcc 指令的执行跳转范围为有符号的 8 位偏移量，可以用于短跳转和条件循环等操作。

1 byte 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 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),
```

### 3.2.8 SAR/SAL/SHL/SHR

1. SAR (算术右移)：SAR 指令是带符号右移指令，用于将一个有符号数向右移动指定的位数，移位时保留符号位。SAR 指令的语法为 `SAR destination, count`，其中，`destination` 表示要移位的目标操作数，`count` 表示要右移的位数。SAR 指令将 `destination` 操作数的每一位都向右移 `count` 位，并用符号位填充空位。
2. SAL (逻辑左移)：SAL 指令是逻辑左移指令，用于将一个无符号数向左移动指定的位数，移位时低位补零。SAL 指令的语法与 SHL 指令相同，可以使用 SHL 代替。
3. SHL (逻辑左移)：SHL 指令是逻辑左移指令，用于将一个无符号数向左移动指定的位数，移位时低位补零。SHL 指令的语法为 `SHL destination, count`。其中，`destination` 表示要移位的目标操作数，`count` 表示要左移的位数。SHL 指令将 `destination` 操作数的每一位都向左移 `count` 位，并用零填充空位。
4. SHR (逻辑右移)：SHR 指令是逻辑右移指令，用于将一个无符号数向右移动指定的位数，移位时高位补零。SHR 指令的语法为 `SHR destination, count`，其中，`destination` 表示要移位的目标操作数，`count` 表示要右移的位数。SHR 指令将 `destination` 操作数的每一位都向右移 `count` 位，并用零填充空位。

```
// logic.c
make_EHelper(shl) {
    // TODO();
    // 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) {
    //TODO();
    // 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);
}

make_EHelper(sar) {
    // TODO();
    // 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) {
    // TODO();
    // 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);
}

// rtl.h
static inline void rtl_mv(rtlreg_t* dest, const rtlreg_t *src1) {
    // dest <- src1
    // TODO();
    rtl_addi(dest, src1, 0);
}

static inline void rtl_not(rtlreg_t* dest) {
    // dest <- ~dest
    // TODO();
    rtl_xori(dest, dest, 0xffffffff);
}

static inline void rtl_sext(rtlreg_t* dest, const rtlreg_t* src1, int width) {
    // dest <- signext(src1[(width * 8 - 1) .. 0])
    // TODO();
    if(width == 0) {
        rtl_mv(dest, src1);
    }
}

```

```

else {
    // assert(width == 1 || width == 2);
    rtl_shli(dest, src1, (4 - width) * 8);
    rtl_sari(dest, dest, (4 - width) * 8);
}
}

//exec.c
/* 0xc0, 0xc1, 0xd0, 0xd1, 0xd2, 0xd3 */
make_group(gp2,
    EX(rol), EMPTY, EMPTY, EMPTY,
    EX(shl), EX(shr), EMPTY, EX(sar))

```

### 3.2.9 TEST

TEST 指令是 x86 汇编语言中的一种逻辑操作指令，用于将两个操作数进行按位逻辑与运算，并设置标志位以反映结果。

TEST 指令会设置 CF 和 OF 标志位为 0，SF、ZF 和 PF 标志位根据运算结果而定。如果结果为 0，则 ZF 被设置为 1，否则 ZF 被清零；如果结果中 1 的个数为偶数，则 PF 被设置为 1，否则 PF 被清零；如果结果的最高位为 1，则 SF 被设置为 1，否则 SF 被清零。

```

// logic.c
make_EHHelper(test) {
    // TODO();
    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);
}

```

扩展 opcode 时，F6、F7 处使用了 `IDEXW(E, gp3, 1)` 与 `IDEX(E, gp3)`。这里的 E 只从 ModR/M 中读取了 r/m 的信息（即 LeftSRC），另一 RightSRC 为立即数，需要在 gp3[0] 的译码中进行读取。故定义为 `IDEX(test_I, test)`。

```

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

```

### 3.2.10 add

和 sub 相同，只需要修改 CF 和 OF 的判别即可。

```

// arith.c
make_EHHelper(add) {
    // TODO();
    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);
}
// exec.c
/* 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,
/* 0xc0, 0xc1, 0xd0, 0xd1, 0xd2, 0xd3 */
make_group(gp2,
    EX(rol), EMPTY, EMPTY, EMPTY,
    EX(shl), EX(shr), EMPTY, EX(sar))

```

### 3.2.11 cmp

Operation

```

LeftSRC - SignExtend(RightSRC);
(* CMP does not store a result; its purpose is to set the flags *)

```

CMP的操作与Sub类似，但不存储结果，只修改EFLAGS的标志。

```

//arith.c
make_EHelper(cmp) {
    // TODO();
    eflags_modify();
    print_asm_template2(cmp);
}

// exec.c
/* 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,
/* 0x80, 0x81, 0x83 */
make_group(gp1,
    EX(add), EX(or), EX(adc), EX(sbb),
    EX(and), EX(sub), EX(xor), EX(cmp))

```

### 3.2.12 jmp

`make_EHelper(jmp)` 与 `make_EHelper(jmp_rm)` 的区别在于 `jmp_eip` 的计算方法，前者根据偏移量计算 eip，后者根据寄存器取值。

框架代码已实现，只需要填写opcode表：

```

/* 0x80 */ IDEXW(I2E, gp1, 1), IDEX(I2E, gp1), EMPTY, IDEX(SI2E, gp1),

```

### 3.2.13 mul

```

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

```

### 3.2.14 div & idiv

DIV 用于无符号整数除法，而 IDIV 用于带符号整数除法。

```

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

```

### 3.2.15 adc

带进位加法指令。

#### ADC — Add with Carry

Opcode	Instruction	Clocks	Description
14 ib	ADC AL,imm8	2	Add with carry immediate byte to AL
15 iw	ADC AX,imm16	2	Add with carry immediate word to AX
15 id	ADC EAX,imm32	2	Add with carry immediate dword to EAX
80 /2 ib	ADC r/m8,imm8	2/7	Add with carry immediate byte to r/m byte
81 /2 iw	ADC r/m16,imm16	2/7	Add with carry immediate word to r/m word
81 /2 id	ADC r/m32,imm32	2/7	Add with CF immediate dword to r/m dword
83 /2 ib	ADC r/m16,imm8	2/7	Add with CF sign-extended immediate byte to r/m word
83 /2 ib	ADC r/m32,imm8	2/7	Add with CF sign-extended immediate byte into r/m dword
10 /r	ADC r/m8,r8	2/7	Add with carry byte register to r/m byte
11 /r	ADC r/m16,r16	2/7	Add with carry word register to r/m word
11 /r	ADC r/m32,r32	2/7	Add with CF dword register to r/m dword
12 /r	ADC r8,r/m8	2/6	Add with carry r/m byte to byte register
13 /r	ADC r16,r/m16	2/6	Add with carry r/m word to word register
13 /r	ADC r32,r/m32	2/6	Add with CF r/m dword to dword register

#### Operation

$DEST \leftarrow DEST + SRC + CF;$

#### Description

ADC performs an integer addition of the two operands DEST and SRC and the carry flag, CF. The result of the addition is assigned to the first operand (DEST), and the flags are set accordingly. ADC is usually executed as part of a multi-byte or multi-word addition operation. When an immediate byte value is added to a word or doubleword operand, the immediate value is first sign-extended to the size of the word or doubleword operand.

```

//exec.c
/* 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,

```



## 3.2.16 sbb

与adc类似，sbb用于带借位的减法运算。在使用SBB指令时，需要确保 destination 和 source 的大小相同。

### SBB — Integer Subtraction with Borrow

Opcode	Instruction	Clocks	Description
1C ib	SBB AL,imm8	2	Subtract with borrow immediate byte from AL
1D iw	SBB AX,imm16	2	Subtract with borrow immediate word from AX
1D id	SBB EAX,imm32	2	Subtract with borrow immediate dword from EAX
80 /3 ib	SBB r/m8,imm8	2/7	Subtract with borrow immediate byte from r/m byte
81 /3 iw	SBB r/m16,imm16	2/7	Subtract with borrow immediate from r/m word
81 /3 id	SBB r/m32,imm32	2/7	Subtract with borrow immediate dword from r/m dword
83 /3 ib	SBB r/m16,imm8	2/7	Subtract with borrow sign-extended immediate byte from r/m word
83 /3 ib	SBB r/m32,imm8	2/7	Subtract with borrow sign-extended immediate byte from r/m dword
18 /r	SBB r/m8,r8	2/6	Subtract with borrow byte register from r/m byte
19 /r	SBB r/m16,r16	2/6	Subtract with borrow word register from r/m word
19 /r	SBB r/m32,r32	2/6	Subtract with borrow dword from r/m dword
1A /r	SBB r8,r/m8	2/7	Subtract with borrow byte register from r/m byte
1B /r	SBB r16,r/m16	2/7	Subtract with borrow word register from r/m word
1B /r	SBB r32,r/m32	2/7	Subtract with borrow dword register from r/m dword

#### Operation

```
IF SRC is a byte and DEST is a word or dword
THEN DEST = DEST - (SignExtend(SRC) + CF)
ELSE DEST ← DEST - (SRC + CF);
```

```
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,
```

## 3.2.17 neg

NEG 用于求一个操作数的负值。NEG 指令会将操作数取反并加 1，得到的结果作为新的值存入操作数中。

### NEG — Two's Complement Negation

Opcode	Instruction	Clocks	Description
F6 /3	NEG r/m8	2/6	Two's complement negate r/m byte
F7 /3	NEG r/m16	2/6	Two's complement negate r/m word
F7 /3	NEG r/m32	2/6	Two's complement negate r/m dword

#### Operation

```
IF r/m = 0 THEN CF ← 0 ELSE CF ← 1; FI;
r/m ← - r/m;
```

#### Description

NEG replaces the value of a register or memory operand with its two's complement. The operand is subtracted from zero, and the result is placed in the operand.

The carry flag is set to 1, unless the operand is zero, in which case the carry flag is cleared to 0.



其影响如下：

- CF（进位标志）：如果操作数为零，则 CF 被清零，否则 CF 被设置为 1。
- OF（溢出标志）：如果操作数为  $-2^{31}$ ，则 OF 被设置为 1，否则 OF 被清零。
- PF（奇偶标志）：如果结果的二进制表示中 1 的个数为偶数，则 PF 被设置为 1，否则 PF 被清零。
- SF（符号标志）：如果结果为负，则 SF 被设置为 1，否则 SF 被清零。
- ZF（零标志）：如果结果为零，则 ZF 被设置为 1，否则 ZF 被清零。

```
//arith.c
make_EHelper(neg) {
    // TODO();
    rtl_sub(&t2, &tzero, &id_dest->val);
    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);
}

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

### 3.2.18 or

```
//logic.c
make_EHelper(or) {
    // TODO();

    rtl_or(&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(or);
    print_asm_template2(or);
}

//exec.c
/* 0x08 */    IDEXW(G2E, or, 1), IDEX(G2E, or), IDEXW(E2G, or, 1), IDEX(E2G,
or),
/* 0x0c */    IDEXW(I2a, or, 1), IDEX(I2a, or), EMPTY, EX(2byte_esc),
/* 0x80, 0x81, 0x83 */
make_group(gp1,
    EX(add), EX(or), EX adc), EX(sbb),
    EX(and), EX(sub), EX(xor), EX(cmp))
```

### 3.2.19 not

```
//logic.c
make_EHelper(not) {
    // TODO();
    rtl_not(&id_dest->val);
    operand_write(id_dest, &id_dest->val);
    print_asm_template1(not);
}

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

### 3.2.20 dec

dec指令：用于将操作数减1，并把新值存储到操作数中。dec指令只影响OF, SF, ZF, AF, and PF标志位。

```
//exec.c
/* 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)

//arith.c
make_EHelper(dec) {
    // TODO();
    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);
}

//exec.c
/* 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),
```

### 3.2.21 inc

$DEST \leftarrow DEST + 1;$

与dec相似，不影响CF标志位。

```
//arith.c
make_EHelper(inc) {
    //TODO();
```

```

    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);
}
//exec.c
/* 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),

```

### 3.2.22 cltd

cltd: 将带符号的长整数转换为带符号的双长整数。

作用是把 eax 的 32 位整数扩展为 64 位，高 32 位用 eax 的符号位填充保存到 edx，或 ax 的 16 位整数扩展为 32 位，高 16 位用 ax 的符号位填充保存到 dx。

```

//data-mov.c
make_EHelper(cltd) {
    if (decoding.is_operand_size_16) {
        // TODO();
        rtl_lr_w(&t0, R_AX);
        rtl_sext(&t0, &t0, 2);
        rtl_sari(&t0, &t0, 31);
        rtl_sr_w(R_DX, &t0);
    }
    else {
        // TODO();
        rtl_sari(&cpu.edx, &cpu.eax, 31);
    }
    print_asm(decoding.is_operand_size_16 ? "cwtl" : "cltd");
}
//exec.c
/* 0x98 */    EX(cwtl), EX(cltd), EMPTY, EMPTY,

```

### 3.2.23 leave

LEAVE 指令用于执行栈帧的恢复操作，其作用相当于执行以下两个操作：

1. 将 ESP 寄存器的值设置为 EBP 寄存器中保存的值，以恢复栈指针的位置。
2. 将 EBP 寄存器的值设置为栈顶中保存的值，以恢复调用者的栈帧。

它等价于：

```
mov esp, ebp
pop ebp
```

```
//data-mov.c
make_EHhelper(leave) {
    // TODO();
    rtl_mv(&cpu.esp, &cpu.ebp);
    rtl_pop(&cpu.ebp);
    print_asm("leave");
}
// exec.c
/* 0xc8 */ EMPTY, EX(leave), EMPTY, EMPTY,
```

### 3.2.24 call\_rm

`CALL rm` 指令的目标地址是通过寄存器或内存中的值来确定的，其寻址方式类似于其他的寄存器间接寻址指令。与第一阶段实现的call的区别在于 `jmp_eip` 的计算方法。

Opcode	Instruction	Clocks	Description
E8 cw	CALL rel16	7+m	Call near, displacement relative to next instruction
FF /2	CALL r/m16	7+m/10+m	Call near, register indirect/memory indirect
9A cd	CALL ptr16:16	17+m, pm=34+m	Call intersegment, to full pointer given
9A cd	CALL ptr16:16	pm=52+m	Call gate, same privilege
9A cd	CALL ptr16:16	pm=86+m	Call gate, more privilege, no parameters
9A cd	CALL ptr16:16	pm=94+4x+m	Call gate, more privilege, x parameters
9A cd	CALL ptr16:16	ts	Call to task
FF /3	CALL m16:16	22+m, pm=38+m	Call intersegment, address at r/m dword
FF /3	CALL m16:16	pm=56+m	Call gate, same privilege
FF /3	CALL m16:16	pm=90+m	Call gate, more privilege, no parameters
FF /3	CALL m16:16	pm=98+4x+m	Call gate, more privilege, x parameters
FF /3	CALL m16:16	5 + ts	Call to task
E8 cd	CALL rel32	7+m	Call near, displacement relative to next instruction
FF /2	CALL r/m32	7+m/10+m	Call near, indirect
9A cp	CALL ptr16:32	17+m, pm=34+m	Call intersegment, to pointer given
9A cp	CALL ptr16:32	pm=52+m	Call gate, same privilege
9A cp	CALL ptr16:32	pm=86+m	Call gate, more privilege, no parameters
9A cp	CALL ptr32:32	pm=94+4x+m	Call gate, more privilege, x parameters
9A cp	CALL ptr16:32	ts	Call to task
FF /3	CALL m16:32	22+m, pm=38+m	Call intersegment, address at r/m dword
FF /3	CALL m16:32	pm=56+m	Call gate, same privilege
FF /3	CALL m16:32	pm=90+m	Call gate, more privilege, no parameters
FF /3	CALL m16:32	pm=98+4x+m	Call gate, more privilege, x parameters
FF /3	CALL m16:32	5 + ts	Call to task

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

## 3.3 Differential Testing

由于有些指令实现得不一样，并不需要比较eflags是否相同。

```
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);
}
if (diff) {
    nemu_state = NEMU_END;
}

```

并且打开 `nemu/include/common.h` 中的宏 `DIFF-TEST`

```

4  #define DEBUG
5  #define DIFF_TEST

```

### 3.4 一键回归测试

执行 `bash runall.sh`:

```

myrrolinz@Myrrolinz:/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/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!
myrrolinz@Myrrolinz:/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nemu$

```

所有测试样例都能正确通过。

#### 捕捉死循环。如何实现？

对程序设置最大运行时间，如果超过该时间则判断程序进入死循环。

对于一个操作系统来说，操作系统可以通过检测进程是否超过了分配给它的时间片来判断是否出现了死循环。当一个进程的时间片用完时，操作系统会强制将其挂起，并将控制权转移到其他进程。如果一个进程在多次运行时都超过了它的时间片，那么很可能出现了死循环，操作系统可以将其杀死或报告错误。

## 4 阶段三 输入输出

理解volatile关键字。

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？

答：变量如果加了 `volatile` 修饰，编译器可能会对代码进行优化，则会从内存重新装载内容，而不是直接从寄存器拷贝内容，去掉 `volatile` 会导致错误发生。会检测不到设备寄存器的变化。

### 4.1 串口

实现串口要实现 `in` 和 `out` 指令。

#### 4.1.1 代码实现

查阅i386手册，查找关于 `in` 和 `out` 指令相关的内容：

## IN — Input from Port

Opcode	Instruction	Clocks	Description
E4 ib	IN AL,imm8	12,pm=6*/26**	Input byte from immediate port into AL
E5 ib	IN AX,imm8	12,pm=6*/26**	Input word from immediate port into AX
E5 ib	IN EAX,imm8	12,pm=6*/26**	Input dword from immediate port into EAX
EC	IN AL,DX	13,pm=7*/27**	Input byte from port DX into AL
ED	IN AX,DX	13,pm=7*/27**	Input word from port DX into AX
ED	IN EAX,DX	13,pm=7*/27**	Input dword from port DX into EAX

## OUT — Output to Port

Opcode	Instruction	Clocks	Description
E6 ib	OUT imm8,AL	10,pm=4*/24**	Output byte AL to immediate port number
E7 ib	OUT imm8,AX	10,pm=4*/24**	Output word AL to immediate port number
E7 ib	OUT imm8,EAX	10,pm=4*/24**	Output dword AL to immediate port number
EE	OUT DX,AL	11,pm=5*/25**	Output byte AL to port number in DX
EF	OUT DX,AX	11,pm=5*/25**	Output word AL to port number in DX
EF	OUT DX,EAX	11,pm=5*/25**	Output dword AL to port number in DX

修改 `system.c`, `in` 指令用于将设备寄存器中的数据传输到 CPU 寄存器中, `out` 指令正好相反:

```
// system.c
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
}

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

`in` 和 `out` 指令在函数中分别调用 `pio_read()` 和 `pio_write()` 函数。

之后需要在 `nexus-am/am/arch/x86-nemu/src/trm.c` 中定义宏 `HAS_SERIAL`。

## 4.1.2 测试结果

运行HELLO WORLD，结果如下所示：

```
[src/monitor/diff-test/diff-test.c,96,init_diff_test] Connect to QEMU successfully
[src/monitor/monitor.c,65,load_img] The image is /mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nexus-am/apps/hello/build/hello-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 11:34:38, Apr 13 2023
For help, type "help"
(nemu) c
[]Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
nemu: HIT GOOD TRAP at eip = 0x00100076
(nemu)
```

## 4.2 时钟

需要完善 `nexus-am/am/arch/x86-nemu/src/ioe.c` 中的代码。

### 4.2.1 代码实现

`uptime()` 返回的是系统 (`x86-nemu`) 启动后经过的毫秒数。`ioe_init()` 中的 `boot_time` 计算的是系统启动到 IOE 启动时已经经过的毫秒数。故当前 `timer` 需要减去初始时间 `boot_time` 即可。

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

### 4.2.2 测试结果

`time-test`运行：

```
[src/monitor/diff-test/diff-test.c,96,init_diff_test] Connect to QEMU successfully
[src/monitor/monitor.c,65,load_img] The image is /mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nexus-am/tests/timetest/build/timetest-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 11:34:38, Apr 13 2023
For help, type "help"
(nemu) c
01 seconds.
2 seconds.
3 seconds.
4 seconds.
5 seconds.
6 seconds.
7 seconds.
8 seconds.
9 seconds.
10 seconds.
11 seconds.
12 seconds.
13 seconds.
14 seconds.
15 seconds.
16 seconds.
17 seconds.
18 seconds.
19 seconds.
20 seconds.
21 seconds.
```

测试成功。

然后进行跑分测试：

1. Dhrystone



```
Dhrystone Benchmark, Version C, Version 2.2
Trying 500000 runs through Dhrystone.
Finished in 20 ms

=====
Dhrystone PASS          51513 Marks
                        vs. 100000 Marks (i7-6700 @ 3.40GHz)
```

## 2. Coremark

```
make: [klib] Error 2 (ignored)
Running CoreMark for 1000 iterations
2K performance run parameters for coremark.
CoreMark Size      : 666
Total time (ms)    : 112
Iterations         : 1000
Compiler version   : GCC5.4.0 20160609
seedcrc            : 0xe9f5
[0]crclist         : 0xe714
[0]crcmatrix       : 0x1fd7
[0]crcstate        : 0x8e3a
[0]crcfinal        : 0xd340
Finised in 112 ms.

=====
CoreMark PASS       39898 Marks
                    vs. 100000 Marks (i7-6700 @ 3.40GHz)
```

## 3. microbench

```
[sieve] Eratosthenes sieve: * Passed.
      min time: 70 ms [60580]
[15pz] A* 15-puzzle search: * Passed.
      min time: 14 ms [41371]
[dinic] Dinic's maxflow algorithm: * Passed.
      min time: 16 ms [84600]
[lzip] Lzip compression: * Passed.
      min time: 33 ms [80209]
[ssort] Suffix sort: * Passed.
      min time: 9 ms [65722]
[md5] MD5 digest: * Passed.
      min time: 24 ms [81637]

=====
MicroBench PASS     57265 Marks
                    vs. 100000 Marks (i7-6700 @ 3.40GHz)
```

## 4.3 键盘I/O

### 4.3.1 代码实现

当按下下一个键的时候,键盘将会发送该键的通码(make code);当释放一个键的时候,键盘将会发送该键的断码(break code)。每当用户敲下/释放按键时,将会把相应的键盘码放入数据寄存器,同时把状态寄存器的标志设置为 1,表示有按键事件发生。CPU 可以通过端口 I/O 访问这些寄存器,获得键盘码。在 AM 中,约定通码的值为断码+0x8000。

0x60: `I8042_DATA_PORT`, 数据寄存器, 4 字节。

0x64: `I8042_STATUS_PORT`, 状态寄存器, 1 字节。

`i8042_io_handler()`: IO 读写时的设备回调函数。该函数只支持读操作: 读取数据时, 将状态寄存器置为 0; 读取状态时, 若当前状态为 0 且存在按下/释放按键事件 (`key_queue` 的 `key_f!=key_r`), 则将事件记录在数据寄存器中并将状态寄存器置 1。

代码如下所示:

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

## 4.3.2 测试结果

可以看到成功实现了键盘功能。

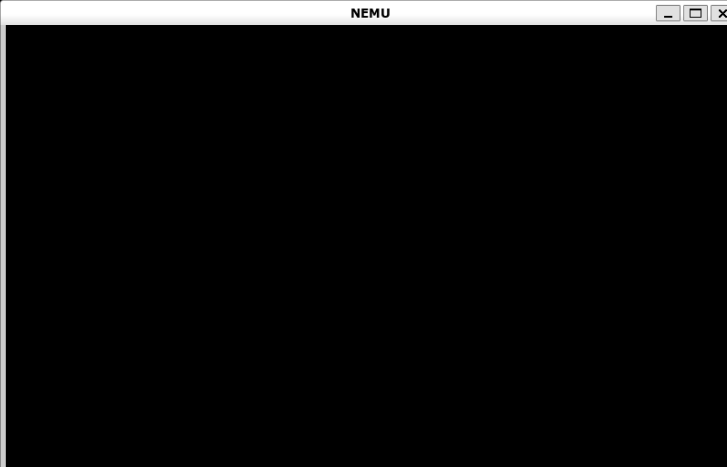
### 如何检测多个键被同时按下？

当发送的数字为键盘码 +0x8000 时，意味着键盘被按下，而当单纯发送键盘码时，意味着键盘被抬起。每个按键的通码断码都不同，所以可以识别出不同的按键。

当检测到一个键被按下的时候，去检测此时其他是否有按键被按下。

从这里开始会使用图形化界面，但是WSL2现在已支持运行Linux GUI（需要Windows 10版本 19044+或Windows 11才能访问此功能），因此不再需要特地另外配置图形化界面了。

```
[src/monitor/monitor.c,65,load_img] The image is /mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nex
keytest/build/keytest-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 11:51:16, Apr 13 2023
For help, type "help"
(nemu) c
[]Get key: 31 E down
Get key: 31 E up
Get key: 31 E down
Get key: 31 E up
Get key: 45 D down
Get key: 45 D up
Get key: 45 D down
Get key: 45 D up
Get key: 46 F down
Get key: 46 F up
Get key: 44 S down
Get key: 44 S up
Get key: 45 D down
Get key: 45 D up
Get key: 45 D down
Get key: 45 D up
Get key: 45 D down
Get key: 45 D up
Get key: 43 A down
Get key: 43 A up
```



## 4.4 VGA

### 4.4.1 代码实现

#### 神奇的调色板

在一些 90 年代的游戏里,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?

答: 当游戏场景需要进行渐出渐入效果时, 游戏开发者可以通过修改调色板中的颜色序列, 使原本的颜色逐渐变暗或变亮, 从而实现渐出渐入的效果。例如, 当一个游戏场景需要进行淡出效果时, 游戏开发者可以将调色板中的颜色序列从明亮的颜色逐渐变为黑色, 从而实现淡出的效果。

在 `paddr_read()` 和 `paddr_write()` 中加入对内存映射 I/O 的判断。通过 `is_mmio()` 函数判断一个物理地址是否被映射到 I/O 空间, 如果是, `is_mmio()` 会返回映射号, 否则返回-1。内存映射 I/O 的访问需要调用 `mmio_read()` 或 `mmio_write()`, 调用时需要提供映射号。如果不是内存映射 I/O 的访问, 就访问 `pmem`。

- `nemu/src/device/io/mmio.c` 中: `int is_mmio(paddr_t addr)`: 查看该内存地址 `addr`, 若为内存映射 IO, 则返回映射号, 否则返回-1
- `uint32_t mmio_read(paddr_t addr, int len, int map_NO)` 与 `void mmio_write(paddr_t addr, int len, uint32_t data, int map_NO)` 为内存映射 IO 的读写访问函数。

修改 `nemu/src/memory/memory.c`:

需要引用头文件 `device/mmio.h`, 相当于把io放入内存中。先用 `is_mmio` 函数判断物理地址是否被映射到 I/O 空间, 对于 `paddr_read` 若返回 -1 则访问 `pmem`, 否则使用 `mmio_read` 函数读取 `port` 位置的内存, 对于 `paddr_write`, 若不返回 -1 则调用 `mmio_write` 将数据写入 `port` 位置内存。

```

#include "device/mmio.h"
/* Memory accessing interfaces */

uint32_t paddr_read(paddr_t addr, int len) {
    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) {
    int r = is_mmio(addr);
    if(r == -1){
        memcpy(guest_to_host(addr), &data, len);
    }
    else {
        mmio_write(addr, len, data, r);
    }
}

```

需要在 `nexus-am/am/arch/x86-nemu/src/ioe.c` , 实现 `_draw_rect()` 函数, 将一个矩阵中像素赋值, 需要注意赋值的顺序。

```

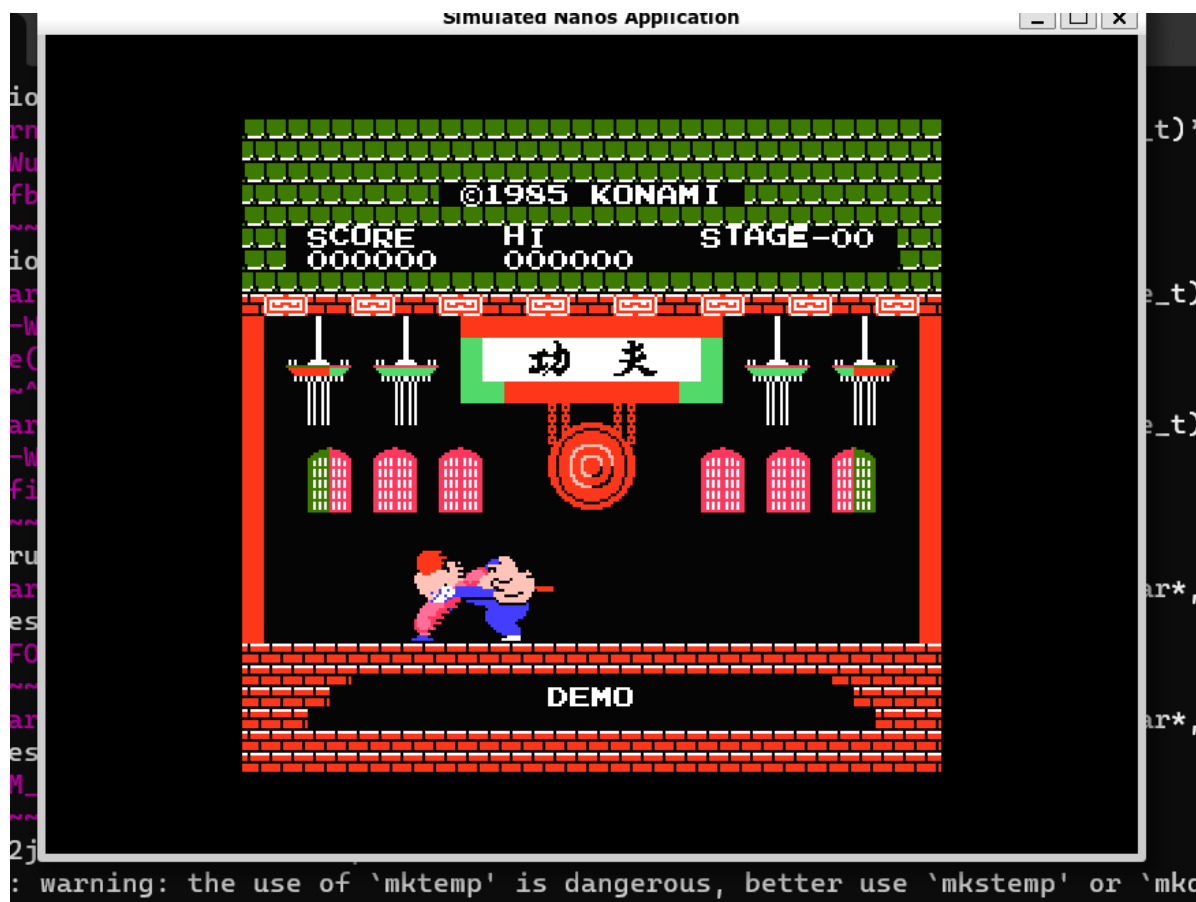
void _draw_rect(const uint32_t *pixels, int x, int y, int w, int h) {

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

```

#### 4.4.2 测试结果





## 5 必答题

### 5.1 问题一

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?

#### 5.1.1 去掉static

这里尝试去掉 `rtl_push` 的 `static`。

```
from src/cpu/exec/special.c:1:
./include/cpu/rtl.h:159:3: warning: 'rtl_sm' is static but used in inline function 'rtl_push' which is not static
159 | rtl_sm(&cpu.esp, 4, src1);
    | ~~~~~^
./include/cpu/rtl.h:158:3: warning: 'rtl_subi' is static but used in inline function 'rtl_push' which is not static
158 | rtl_subi(&cpu.esp, &cpu.esp, 4);
    | ~~~~~^
+ CC src/cpu/exec/system.c
In file included from ./include/cpu/decode.h:6,
                 from ./include/cpu/exec.h:9,
                 from src/cpu/exec/system.c:1:
./include/cpu/rtl.h:159:3: warning: 'rtl_sm' is static but used in inline function 'rtl_push' which is not static
159 | rtl_sm(&cpu.esp, 4, src1);
    | ~~~~~^
./include/cpu/rtl.h:158:3: warning: 'rtl_subi' is static but used in inline function 'rtl_push' which is not static
158 | rtl_subi(&cpu.esp, &cpu.esp, 4);
    | ~~~~~^
```

只有警告, 但可以正常运行。inline实际上表示建议内联, 但并非强制内联, 编译器可以忽略。若想要确保内联, 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, 若多个c文件都包含时就会产生歧义。如果去掉static, 编译器没有强制内联, 该函数就相当于一个普通函数。

## 5.1.2 去掉inline

这里尝试去掉 `rtl_push` 的 `inline`。

```
./include/cpu/rtl.h:154:14: warning: 'rtl_push' defined but not used [-Wunused-function]
154 | static void rtl_push(const rtlreg_t* src1) {
    |             ^~~~~~
+ CC src/cpu/exec/special.c
In file included from ./include/cpu/decode.h:6,
                  from ./include/cpu/exec.h:9,
                  from src/cpu/exec/special.c:1:
./include/cpu/rtl.h:154:14: warning: 'rtl_push' defined but not used [-Wunused-function]
154 | static void rtl_push(const rtlreg_t* src1) {
    |             ^~~~~~
+ CC src/cpu/exec/system.c
In file included from ./include/cpu/decode.h:6,
                  from ./include/cpu/exec.h:9,
                  from src/cpu/exec/system.c:1:
./include/cpu/rtl.h:154:14: warning: 'rtl_push' defined but not used [-Wunused-function]
154 | static void rtl_push(const rtlreg_t* src1) {
    |             ^~~~~~
+ CC src/cpu/intr.c
In file included from ./include/cpu/decode.h:6,
                  from ./include/cpu/exec.h:9,
                  from src/cpu/intr.c:1:
./include/cpu/rtl.h:154:14: warning: 'rtl_push' defined but not used [-Wunused-function]
154 | static void rtl_push(const rtlreg_t* src1) {
    |             ^~~~~~
```

会出现一堆定义但未使用的问题。

当去掉 `inline` 关键字时，编译器会将该函数生成作为一个独立的代码块，并将其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，则编译器会认为该函数是未使用的，并给出相应的警告。因此，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。

## 5.1.3 去掉static和inline

这里尝试去掉 `rtl_push` 的 `static inline`。

```
/usr/bin/ld: build/obj/cpu/exec/exec.o: in function 'rtl_push':
/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nemu/./include/cpu/rtl.h:154: multiple definition of 'rtl_push';
build/obj/cpu/decode/decode.o:/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nemu/./include/cpu/rtl.h:154: first
defined here
/usr/bin/ld: build/obj/cpu/exec/logic.o: in function 'rtl_push':
/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nemu/./include/cpu/rtl.h:154: multiple definition of 'rtl_push';
build/obj/cpu/decode/decode.o:/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nemu/./include/cpu/rtl.h:154: first
defined here
/usr/bin/ld: build/obj/cpu/exec/prefix.o: in function 'rtl_push':
/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nemu/./include/cpu/rtl.h:154: multiple definition of 'rtl_push';
build/obj/cpu/decode/decode.o:/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nemu/./include/cpu/rtl.h:154: first
defined here
/usr/bin/ld: build/obj/cpu/exec/special.o: in function 'rtl_push':
/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nemu/./include/cpu/rtl.h:154: multiple definition of 'rtl_push';
build/obj/cpu/decode/decode.o:/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nemu/./include/cpu/rtl.h:154: first
defined here
/usr/bin/ld: build/obj/cpu/exec/system.o: in function 'rtl_push':
/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nemu/./include/cpu/rtl.h:154: multiple definition of 'rtl_push';
build/obj/cpu/decode/decode.o:/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nemu/./include/cpu/rtl.h:154: first
defined here
/usr/bin/ld: build/obj/cpu/intr.o: in function 'rtl_push':
/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nemu/./include/cpu/rtl.h:154: multiple definition of 'rtl_push';
build/obj/cpu/decode/decode.o:/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nemu/./include/cpu/rtl.h:154: first
defined here
collect2: error: ld returned 1 exit status
make[2]: *** [Makefile:43: build/nemu] Error 1
make[1]: *** [/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nexus-am/Makefile.app:35: run] Error 2
make: [Makefile:13: Makefile.max] Error 2 (ignored)
max
myrrrolinz@Myrrrolinz:/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nexus-am/tests/cputest$
```

直接报错，无法运行。会出现重复定义的问题。

当同时去掉 `static` 和 `inline` 关键字时，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。

因为 `exe` 文件夹下的 `.c` 文件中都引用了 `rtl.h` 文件，编译的时候这些文件和 `rtl.h` 文件中都有了 `rtl_mv` 函数的定义，导致了多次定义。

因此,同时去掉 `static` 和 `inline` 关键字可能会导致重复定义的问题。

## 5.2 问题二

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

### 5.2.1 make 执行过程

`make` 程序会进行以下步骤:

1. 读取 Makefile 文件, 并解析其中的变量、规则和指令。
2. 根据 Makefile 中的规则和指令, 确定需要生成的目标文件, 以及生成目标文件所需的依赖文件和编译命令。
3. 检查目标文件和依赖文件的时间戳, 确定哪些文件需要重新生成。
  - 如果文件不存在, 或是文件所依赖的后面的 `.o` 文件的文件修改时间要比这个文件新, 那么, 他就会执行后面所定义的命令来生成 `h` 这个文件, 这个也就是重编译
  - 如果文件所依赖的 `.o` 文件也存在, 那么 `make` 会在当前文件中找目标为 `.o` 文件的依赖性, 如果找到则再根据那一个规则生成 `.o` 文件。
  - `.c` 文件和 `.h` 文件存在, 于是 `make` 会生成 `.o` 文件
4. 根据需要重新生成的文件, 执行相应的编译命令, 生成目标文件。
5. 重复步骤 3 和步骤 4, 直到所有的目标文件都已经生成。

具体地说, 在编译链接的过程中, Makefile 会执行以下操作:

1. 根据源文件的后缀名, 使用隐式规则来生成相应的目标文件。例如, 对于 `.c` 文件, Makefile 中定义了如下的规则:

```
%o: %.c $(CC) $(CFLAGS) $(CPPFLAGS) -c -o @<
```

这个规则表示, 对于每个 `.c` 文件, 使用 `$(CC)` 变量所指定的编译器, 加上 `$(CFLAGS)` 和 `$(CPPFLAGS)` 变量所指定的编译选项, 将其编译成一个 `.o` 目标文件, 并保存在 `$(@)` 变量所表示的位置。

2. 根据目标文件之间的依赖关系, 生成可执行文件。例如, 在 `nemu` 的 Makefile 中, 定义了如下的规则:

```
nemu: $(OBJS) $(CC) $(LDFLAGS) -o $@ $(LDLIBS)
```

这个规则表示, 对于 `nemu` 目标文件, 它依赖于 `$(OBJS)` 变量所表示的所有 `.o` 目标文件, 使用 `$(CC)` 变量所指定的编译器, 加上 `$(LDFLAGS)` 变量所指定的链接选项, 将所有 `.o` 目标文件链接成一个可执行文件, 并保存在 `$(@)` 变量所表示的位置。

在生成目标文件和可执行文件的过程中, Makefile 还会使用一些特殊的变量和函数, 用于简化编译链接的操作。例如, `$(wildcard pattern)` 函数可以用来匹配指定模式的文件, `$(patsubst pattern, replacement, text)` 函数可以用来替换字符串中的某个模式。Makefile 中还可以使用 `ifeq`、`ifneq`、`ifdef`、`ifndef` 等条件判断语句, 根据不同的条件选择不同的规则和指令。



## 5.2.2 Makefile中使用的变量和包含文件等特性

基本变量：

- 实现各文件的路径/名称声明
- 定义编译的源文件与中间目标文件

高级变量

- `$<`：规则型变量，表示第一个依赖文件
- `$@`：表示目标文件

从而将所有.c 文件编译生成对应的.o 文件。

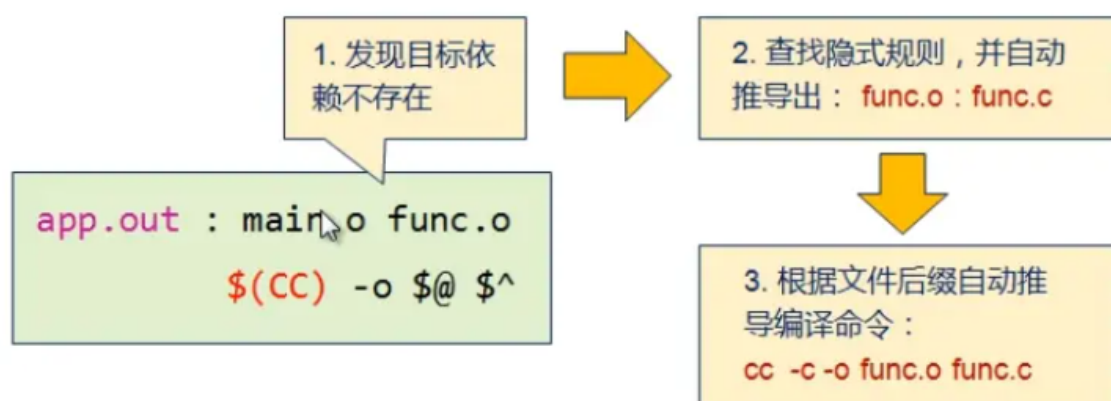
包含文件特性：

可以使用 `include` 命令包含其他的 Makefile 文件，例如：`include common.mk`，这里将会包含名为 `common.mk` 的文件中的所有规则。

## 5.2.3 implicit rules

当make发现目标的依赖不存在时：

- 尝试通过依赖名逐一查找隐式规则；
- 并且通过依赖名推导出可能需要的源文件。



举个例子：如果有一个名为 `foo.c` 的文件，而且没有指定如何编译它，Makefile 将会使用默认的规则来编译它，例如：

```
.c.o:
    $(CC) $(CFLAGS) -c $<
```

这条规则告诉 Makefile，如果有一个 `.c` 文件需要编译成 `.o` 文件，那么应该使用 `$(CC)` 编译器和 `$(CFLAGS)` 编译选项来编译它。这是一个隐式规则，它会自动应用于所有需要编译成 `.o` 文件的 `.c` 文件。

Makefile 还可以重写隐式规则，例如：

```
%.o: %.cpp
    $(CXX) $(CXXFLAGS) -c $< -o $@
```

这条规则告诉 Makefile，如果有一个 `.cpp` 文件需要编译成 `.o` 文件，那么应该使用 `$(CXX)` 编译器和 `$(CXXFLAGS)` 编译选项来编译它。这个规则会替代原来的 `.c.o` 规则。



## 5.2.4 man make -n

在 man make 中搜索 `-n` 选项，可以找到如下描述：

```
arduinoCopy code-n, --just-print, --dry-run, --recon  
    Print the commands that would be executed, but do not execute them.
```

这个选项告诉 Makefile 只输出将会执行的命令，但是不真正执行它们。使用这个选项可以帮助程序员在不真正执行命令的情况下检查 Makefile 的正确性，以及检查将会执行哪些命令，以及它们的顺序。这对于大型项目的编译和构建过程非常有用，可以节省时间和资源。

## 6 实验感想

本次实验过程还算顺利，没有特别卡bug的地方。原本以为会在图形化界面卡一会儿，但是后来发现 wsl2 直接可以使用 GUI 了，从未如此感谢 Windows 11 🙏。

这次要实现的指令实在太多了，光是查手册就花了好多时间，总计用时远远不止40小时...但是最后能实现打字小游戏还是很有成就的！

通过这次实验，我加深了对NEMU的理解，阅读源码的能力进一步提升。希望下一次实验也能顺利。