

计算机系统设计实验

PA2 - 简单复杂的机器：冯诺依曼计算机系统

安祺 1913630

南开大学计算机学院

日期：2022 年 3 月 30 日

目录

1 概述	3	3.2.8 MOVZX/MOVSX . . .	17
1.1 实验目的	3	3.2.9 TEST	17
1.2 实验内容	3	3.2.10 Jcc	17
2 阶段一	3	3.2.11 POP	17
2.1 EFLAGS 寄存器	3	3.2.12 INC	18
2.2 必要指令	4	3.2.13 ADC	18
2.2.1 CALL	4	3.2.14 XOR	18
2.2.2 RET	6	3.2.15 OR	19
2.2.3 PUSH	7	3.2.16 SAL/SAR/SHL/SHR .	19
2.2.4 XOR	8	3.2.17 DEC	20
2.2.5 SUB	9	3.2.18 LEAVE	20
3 阶段二	11	3.2.19 NOT	20
3.1 可执行文件的堆和栈	11	3.2.20 JMP	20
3.2 必要指令	12	3.2.21 MUL/IMUL/DIV/IDIV	21
3.2.1 LEA	12	3.2.22 CWD/CDQ	21
3.2.2 AND	12	3.2.23 SUB/SBB	21
3.2.3 PUSH	14	4 阶段三	21
3.2.4 NOP	14	4.1 Differential Testing	21
3.2.5 ADD	14	4.2 定位死循环	22
3.2.6 CMP	14	4.3 输入输出	22
3.2.7 SETcc	15	4.3.1 理解 volatile 关键字 .	22
		4.3.2 运行 Hello World . . .	23

4.3.3	实现 IOE	24	6 BUG 汇总	31
4.3.4	看看 NEMU 跑多快 .	25	6.1 直接移位 32 位	31
4.3.5	如何检测多个键同 时被按下	25	6.2 diff-test 中的 eflags 初始化 . .	31
4.3.6	实现 IOE(2): 键盘 .	25	6.3 高版本环境 diff-test 不能正 常运行	31
4.3.7	神奇的调色板	26	6.4 ADC/SBB 指令的 CF 位 . . .	32
4.3.8	添加内存映射 I/O: VGA	26	6.5 QEMU 会出现令人费解的 情况	32
4.3.9	实现 IOE(3): 动画 .	27	6.6 忘记设置 HAS_IOE	33
4.3.10	运行打字小游戏 . . .	27	6.7 打开 HAS_IOE 后, WSL 终 端卡死	33
5 必答题		29	6.8 高版本 linux 环境下 mi- crobench 出现 FAILED	33
5.1	编译与链接	29	7 体会与感悟	33
5.2	了解 Makefile	30		

1 概述

1.1 实验目的

1. 理解体会指令执行和设备输入输出的原理

1.2 实验内容

阶段一 在 NEMU 中运行第一个 C 程序 dummy。

阶段二 实现更多的指令, 在 NEMU 中运行所有 cputests。

阶段三 运行打字小游戏。

2 阶段一

2.1 EFLAGS 寄存器

EFLAGS 寄存器¹的实现需要借助结构体位域特性²。在结构体中使用连续的相同类型字段, 位域会进行压缩存储, 并且小段字节序机器的字节内位序同样是小端排列。因此, 通过在一个 32 位整数中通过位域实现 EFLAGS 寄存器, 如代码段1所示。其中, 各保留位用数字表示。

Listing 1: EFLAGS 的实现

```
union {
    rtlreg_t reg;
    struct { /* Bit-Fields */
        rtlreg_t CF:1, _1:1, PF:1, _3:1, AF:1, _5:1, ZF:1, SF:1, /*1-7*/
                TF:1, IF:1, DF:1, OF:1, IOPL:2,      NT:1, _f:1, /*8-15*/
                RF:1, VM:1, :0;                          /*16-*/
    };
} eflags;
```

此外, 写了一个简易的测试来验证 EFLAGS 实现的正确性, 确保互相不发生重叠, 各标志位位于对应 bit 位上, 如代码段2所示。

在实现 EFLAGS 寄存器后, 实现了对 EFLAGS 寄存器的初始化, 如代码段3所示。其中 EFLAGS 的初始值应为00000002H³。

Listing 2: EFLAGS 的验证

¹各标志位参考《INTEL 80386 PROGRAMMER'S REFERENCE MANUAL》第 34 页图 2-8。

²位域的内存分布规则参考了这篇博客: <https://www.cnblogs.com/axjlx/p/15008070.html>

³EFLAGS 寄存器的初始值参考《INTEL 80386 PROGRAMMER'S REFERENCE MANUAL》第 174 页。

```
cpu.eflags.reg=0b111101111011101101;  
assert(cpu.eflags.CF == 1);  
assert(cpu.eflags._1 == 0);  
assert(cpu.eflags.PF == 1);  
assert(cpu.eflags._3 == 1);  
assert(cpu.eflags.AF == 0);  
assert(cpu.eflags._5 == 1);  
assert(cpu.eflags.ZF == 1);  
assert(cpu.eflags.SF == 1);  
assert(cpu.eflags.TF == 0);  
assert(cpu.eflags.IF == 1);  
assert(cpu.eflags.DF == 1);  
assert(cpu.eflags.OF == 1);  
assert(cpu.eflags.IOPL == 0b01);  
assert(cpu.eflags.NT == 1);  
assert(cpu.eflags._f == 1);  
assert(cpu.eflags.RF == 1);  
assert(cpu.eflags.VM == 1);
```

Listing 3: EFALGS 的初始化

```
static inline void restart() {  
    /* Set the initial instruction pointer. */  
    cpu.eip = ENTRY_START;  
    cpu.eflags.reg=0x00000002;  
}
```

2.2 必要指令

为了使 dummy 正常运行需要对 CALL、RET、PUSH、SUB、XOR 几条指令进行实现。由于绝大部分译码函数和执行函数都定义好了,这里的主要工作就是对opcode_table进行填充。未被实现的译码函数和执行函数,会有TODO()函数进行提示,可以有针对性地进行实现。

最终可以成功运行 dummy 程序,如图1。

2.2.1 CALL

操作码为0xe8的指令未实现,查阅得知是 CALL 指令。查阅⁴到 CALL 指令的定义及实现,相关信息如表1所示。因此,在opcode_table中0xe8处插入IDEX(J,call)。

⁴ 《INTEL 80386 PROGRAMMER'S REFERENCE MANUAL》第 34 页

```
aqni@DESKTOP-Q06TNC3:~/ics2017/nexus-am/tests/cputest> make ALL=dummy run
Building dummy [x86-nemu]
Building am [x86-nemu]
make[2]: *** No targets specified and no makefile found. Stop.
[src/monitor/diff-test/diff-test.c,99,init_difftest] Connect to QEMU successfully
[src/monitor/monitor.c,65,load_img] The image is /home/aqni/ics2017/nexus-am/tests/cputest/build/dummy-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 09:53:49, Mar 30 2022
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x0010001b
```

图 1: dummy 运行成功

CALL – Call Procedure		
Opcode	Instruction	Description
E8 cd	CALL rel32	Call near, displacement relative to next instruction
Operation	Push(EIP); EIP ← EIP + rel32;	
Description	略	
Flags Affected	no flags are affected if a task switch does not occur	

表 1: CALL 指令实现定义 (仅 E8)

其中J是解码函数，已经被定义好了；call为执行函数，需要进行实现，暂时只实现32 位操作数的，参见代码段4。

Listing 4: exec_call的实现

```
make_EHelper(call) {
    // the target address is calculated at the decode stage
    if(decoding.is_operand_size_16){
        TODO();
    }else{
        rtl_push(&decoding.seq_eip);
    }
    decoding.is_jump=true;

    print_asm("call %x", decoding.jump_eip);
}
```

在exec_call使用了 rtl 伪指令rtl_push，需要进行实现，参见5。

Listing 5: rtl_push的实现

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

```
    rtl_sm(&cpu.esp,4,src1);           // M[esp] <- src1
}
```

在decode_J中使用了解码函数decode_op-SI，其功能是从当前位置读取有符号立即数，实现见代码段6。

Listing 6: decode_op-SI的实现

```
static inline make_DopHelper(SI) {
    assert(op->width == 1 || op->width == 4);
    op->type = OP_TYPE_IMM;

    if(op->width == 4) { /* operand_size ==32 */
        op->simm=instr_fetch(eip,4);
    }else { /* operand_size ==8 */
        op->simm=(int8_t)(uint8_t)instr_fetch(eip,1);
    }

    rtl_li(&op->val, op->simm);
}
```

2.2.2 RET

操作码为0xc3的指令未实现，查阅得知是 RET 指令。查阅⁵到 RET 指令的定义及实现，相关信息如表2所示。因此，在opcode_table中0xc3处插入EX(ret)。其中ret为执行函数，需要进行实现，暂时只实现 32 位操作数的，参见代码段7。其中，decoding.jump_eip为跳转地址，解码前通过decoding.is_jump判断是否需要跳转。

Ret – Return from Procedure		
C3	Instruction	Description
E8 cd	RET	Call near, displacement relative to next instruction
Operation	EIP ← Pop();	
Description	略	
Flags Affected	None	

表 2: RET 指令实现定义 (仅 C3)

Listing 7: exec_ret的实现

```
make_EHelper(ret) {
    rtl_pop(&decoding.jump_eip);
}
```

⁵ 《INTEL 80386 PROGRAMMER’S REFERENCE MANUAL》 第 378 页

```
decoding.is_jump=true;

print_asm("ret");
}
```

在ret-exec中使用了伪指令rtl_pop，需要进行实现，参见代码段8。

Listing 8: rtl-pop的实现

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

2.2.3 PUSH

操作码为0x55的指令未实现，查阅得知是SUB指令。查阅⁶到PUSH指令的定义及实现，相关信息如表3所示。因此，在opcode_table中0x50到0x57处插入IDEX(r,push)。该指令的操作数保存在操作码中，因此这几个操作码对应的都是该指令。其中push为执行函数，需要进行实现，暂时只实现32位操作数的，参见代码段9；decode_r为解码指令，从操作码中获得寄存器操作数，该函数已经被实现了。

PUSH – Push Operand onto the Stack		
C3	Instruction	Description
50 + /r	PUSH r32	Push register dword
Operation	ESP ← ESP - 4; (SS:ESP) ← (SOURCE); (* dword assignment *)	
Description	PUSH decrements the stack pointer by 2 if the operand-size attribute of the instruction is 16 bits; otherwise, it decrements the stack pointer by 4. PUSH then places the operand on the new top of stack, which is pointed to by the stack pointer.	
Flags Affected	None	

表 3: PUSH 指令实现定义 (仅 50)

Listing 9: exec_push的实现

```
make_EHelper(push) {
    rtl_push(&id_dest->val);

    print_asm_template1(push);
}
```

⁶ 《INTEL 80386 PROGRAMMER’S REFERENCE MANUAL》第 367 页

```
}

```

2.2.4 XOR

操作码为0x31的指令未实现，查阅得知是 XOR 指令。查阅⁷到 XOR 指令的定义及实现，相关信息如表4所示，其中‘r’为扩展位，表示寄存器操作数。因此，在opcode_table中0x31处插入IDEX(G2E,xor)。该指令的操作数保存在操作码中，因此这几个操作码对应的都是该指令。其中xor是异或指令的执行函数，需要进行实现，实现方法见代码段10；G2E为解码函数，解码数据流为从 the modR/M 字节中获得寄存器/内存操作数（来自 r/m 域）并保存到某个寄存器操作数（来自 reg/opcode 域）的指令，该函数已经被实现了。

XOR – Logical Exclusive OR		
C3	Instruction	Description
31 /r	XOR r/m32,r32	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	

表 4: XOR 指令实现定义 (仅 31)

Listing 10: exec_xor的实现

```
make_EHelper(xor) {
    rtl_xor(&t2, &id_dest->val, &id_src->val);
    operand_write(id_dest, &t2);

    rtl_set_CF(&tzero);
    rtl_set_OF(&tzero);
    rtl_update_ZFSF(&t2, id_dest->width);

    print_asm_template2(xor);
}
```

⁷ 《INTEL 80386 PROGRAMMER’S REFERENCE MANUAL》第 367 页

2.2.5 SUB

操作码为0x83的指令未实现，查阅得知是 PUSH 指令。查阅⁸到 SUB 指令的定义及实现，相关信息如表5所示，其中’/5’ 为扩展 opcode，表示应进行减法操作。因此，在opcode_table中0x83处插入IDEX(SI2E, gp1)。该指令的操作数保存在操作码中，因此这几个操作码对应的都是该指令。其中gp1是一组算数指令的执行函数，sub的操作码是 5 即该组中的第 6 个，需要进行实现。sub 的执行函数仿照 sbb 执行函数进行实现，参见代码段11；SI2E为解码指令，从操作码中获得寄存器操作数，该函数已经被实现了。

SUB – Integer Subtraction		
C3	Instruction	Description
83 /5 ib	SUB r/m32,imm8	Subtract sign-extended immediate byte from r/m dword
Operation	DEST = DEST - SignExtend(SRC);	
Description	SUB subtracts the second operand (SRC) from the first operand (DEST). The first operand is assigned the result of the subtraction, and the flags are set accordingly.	
	When an immediate byte value is subtracted from a word operand, the immediate value is first sign-extended to the size of the destination operand	
Flags Affected	OF, SF, ZF, AF, PF, and CF as described in Appendix C	

表 5: SUB 指令实现定义 (仅 83)

Listing 11: exec_sub的实现

```
make_EHelper(sub) {
    rtl_sub(&t2, &id_dest->val, &id_src->val);
    operand_write(id_dest, &t2);

    rtl_update_ZFSF(&t2, id_dest->width);

    rtl_sltu(&t0, &id_dest->val, &t2);
    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);
}
```

⁸ 《INTEL 80386 PROGRAMMER’S REFERENCE MANUAL》第 367 页

```
print_asm_template2(sub);
}
```

sub 指令需要对状态寄存器进行更新。Carry Flag⁹在发生高位借位或进位时设置为1，对于 sub 指令来说，使用补码进行减法运算，若发生借位，则结果的补码会比被减数的补码更大。Overflow Flag表示溢出，在正数过大或者负数过小时被设置，对于 sub 指令来说，若发生溢出，只有两种可能：

1. 负数减去正数得到一个正数。
2. 正数减去一个负数得到一个负数。

对于 EFLAGS 寄存器，当结果为 0 时，ZF 被设置，而 SF 的值与结果的最高比特位保持一致。在exec_sub中使用了rtl_update_ZFZS函数，rtl_update_ZFSF中依次执行rtl_update_ZF（更新 ZF 标志位）和rtl_update_SF（更新 SF 标志位），其实现见代码段13和代码段12。

Listing 12: rtl_update_SF的实现

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

Listing 13: rtl_update_ZF的实现

```
static inline void rtl_update_ZF(const rtlreg_t* result, int width) {
    // eflags.ZF <- is_zero(result[width * 8 - 1 .. 0])
    rtlreg_t result_width;
    switch (width) {
        case 4: rtl_andi(&result_width, result, 0xFFFFFFFF); break;
        case 1: rtl_andi(&result_width, result, 0xFF); break;
        case 2: rtl_andi(&result_width, result, 0xFFFF); break;
        default: assert(0);
    }
    rtlreg_t zf;
    rtl_eq0(&zf, &result_width);
    rtl_set_ZF(&zf);
}
```

此外，rtl_msb、rtl_eq0、rtl_set_XX函数被使用，rtl_set_XX的实现见代码段14，其余两者相对简单。

⁹ 《INTEL 80386 PROGRAMMER'S REFERENCE MANUAL》第 419 页

Listing 14: rtl_set_XX的实现

```
#define make_rtl_setget_eflags(f) \
    static inline void concat(rtl_set_, f) (const rtlreg_t* src) { \
        cpu.eflags.f=(*src!=0)?1:0; \
    } \
    static inline void concat(rtl_get_, f) (rtlreg_t* dest) { \
        *dest=cpu.eflags.f; \
    }

make_rtl_setget_eflags(CF)
make_rtl_setget_eflags(OF)
make_rtl_setget_eflags(ZF)
make_rtl_setget_eflags(SF)
```

3 阶段二

这一阶段指令的实现个数较多，因此报告中对指令实现的介绍相对简略，仅对重要内容进行说明。

3.1 可执行文件的堆和栈

实验指导书 [1] 中提到了这个问题：

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？

堆和栈都是动态扩展的，只保存程序运行时生成的数据。操作系统负责对栈和堆进行初始化，程序只需要进行使用即可。所以，可执行文件无需保存堆和栈的内容。

在 AM 中，堆区是一段有限的内存空间，在链接时被定义，并作为 `_heap` 变量的初始值；栈顶地址在链接时被定义，在 AM 启动时（即应用程序启动前）被初始化给 ESP 寄存器。相关代码见代码段 15。

Listing 15: AM堆区初始化

```
// loader.ld
SECTIONS {
    此处省略无关内容
    _heap_start = ALIGN(4096);
    _stack_pointer = 0x7c00;
    _heap_end = 0x8000000;
```

```

}

// trm.c
_Area _heap = {
    .start = &_heap_start,
    .end = &_heap_end,
};

// start.S
_start:
    mov $0, %ebp
    mov $_stack_pointer, %esp
    call _trm_init                # never return

```

3.2 必要指令

为了通过 cputest 中的所有测试样例，实验过程中实现了一系列指令。将这些指令实现后，成功通过了 cputest 中的所有样例，如图2。

3.2.1 LEA

操作码为 0x8D，借助已经被实现好的exec_lea和decode_lea_M2G即可实现。在opcode 表中插入IDEX(lea_M2G, lea)即可。其中复杂的解码操作在read_ModR_M中被实现，无需再次实现。

3.2.2 AND

操作码为 0x83，扩展操作码为 4，因此在 gp1 中的第五个位置插入EX(and)即可。而exec_and函数尚未实现，其实现如代码段16所示。

Listing 16: exec_and的实现

```

make_EHelper(and) {
    rtl_and(&t2, &id_dest->val, &id_src->val); // DEST <- DEST AND SRC
    operand_write(id_dest, &t2);

    rtl_set_CF(&tzero);                // CF <- 0
    rtl_set_OF(&tzero);                // OF <- 0
    rtl_update_ZFSF(&t2, id_dest->width);

    print_asm_template2(and);
}

```

```
(pa2)aqni@DESKTOP-Q06TNC3:~/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!
```

图 2: 所有测试运行成功

3.2.3 PUSH

根据操作码和指令格式, 可得 PUSH 指令及其opcode_table插入项内容, 如表6所示。注意到, 整个 gp5 的解码函数均使用decode_E, 即解码 Mod_R/M 中的 rm 域作为操作数; 这里使用push_SI作为解码函数, 是为了进行符号扩展。

Opcode	Instruction	Table Item
FF /6	PUSH m16	IDEX(E, gp5)
	PUSH m32	
	PUSH r16	
50 + /r		IDEX(r, push)
	PUSH r32	
6A	PUSH imm8	IDEXW(push_SI, push, 1)
68	PUSH imm16	IDEX(I, push)
	PUSH imm32	

表 6: PUSH 指令实现定义 (其中 gp5[6] 为 EX(push))

3.2.4 NOP

操作码为 0x66, 发现时 Operand_size 指令, 但是 Operand_size 已经实现了, 这里曾疑惑不解。后来发现, Operand_size 是指令前缀, 用于指示当前指令需要改变操作数的宽度。所以, 实际执行的指令的操作码为 0x90, 为 nop 指令, 插入EX(nop)即可。

3.2.5 ADD

根据操作码和指令格式, 可得 ADD 指令及其opcode_table插入项内容, 如表7所示。其中exec_add参考exec_adc实现, 方法与上文exec_sub11一致, 不再赘述。

其中使用了rtl_not指令, 该 rtl 指令需要实现, 见代码段17。

Listing 17: rtl_not的实现

```
static inline void rtl_not(rtlreg_t* dest) {
    *dest = ~*dest;    // dest <- ~dest
}
```

3.2.6 CMP

根据操作码和指令格式, 可得 CMP 指令及其opcode_table插入项内容, 如表8所示。其中exec_cmp的实现见代码段18, 这里实现与 sub 指令一致, 但不保存相减结果; decode_I2a表示该指令的源操作数为立即数, 目的操作数为 eax/ax/al 寄存器。

Opcode	Instruction	Table Item
04 ib	ADD AL,imm8	IDEXW(I2a, add, 1)
05 iw	ADD AX,imm16	IDEX(I2a, add)
05 id	ADD EAX,imm32	
80 /0 ib	ADD r/m8,imm8	IDEXW(I2E, gp1, 1)
81 /0 iw	ADD r/m16,imm16	IDEX(I2E, gp1)
83 /0 ib	ADD r/m16,imm8	IDEX(SI2E, gp1)
00 /r	ADD r/m8,r8	IDEXW(G2E, add ,1)
01 /r	ADD r/m16,r16 ADD r/m32,r32	IDEX(G2E, add)
02 /r	ADD r8,r/m8	IDEXW(E2G, add, 1)
03 /r	ADD r16,r/m16 ADD r32,r/m32	IDEX(E2G, add)

表 7: ADD 指令实现定义 (其中 gp1[0] 为 EX(add))

Listing 18: exec_cmp

```

make_EHelper(cmp) {
    // LeftSRC - SignExtend(RightSRC)
    rtl_sub(&t2, &id_dest->val, &id_src->val);

    rtl_update_ZFSF(&t2, id_dest->width);

    rtl_sltu(&t0, &id_dest->val, &t2);
    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);
}

```

3.2.7 SETcc

操作码为 0x0f90 到 0x0f97, 根据具体的条件各不相同。由于目的寄存器为 r/m(E), 操作数宽度为 1 个字节, 因此在对应位置填入 IDEXW(E, setcc, 1) 即可。其中 exec_setcc 借助 rtl_setcc 实现, 见代码段 19。

Opcode	Instruction	Table Item
3C ib	CMP AL,imm8	IDEXW(I2a, cmp, 1)
3D iw	CMP AX,imm16	IDEX(I2a, cmp)
3D id	CMP EAX,imm32	
80 /7 ib	CMP r/m8,imm8	IDEXW(I2E, gp1, 1)
81 /7 iw	CMP r/m16,imm16	IDEX(I2E, gp1)
81 /7 id	CMP r/m32,imm32	
83 /7 ib	CMP r/m16,imm8 CMP r/m32,imm8	IDEX(SI2E, gp1)
38 /r	CMP r/m8,r8	IDEXW(G2E, cmp, 1)
39 /r	CMP r/m16,r16 CMP r/m32,r32	IDEX(G2E, cmp)
3A /r	CMP r8,r/m8	IDEXW(E2G, cmp, 1)
3B /r	CMP r16,r/m16 CMP r32,r/m32	IDEX(E2G, cmp)

表 8: CMP 指令实现定义 (其中 gp1[7] 为 EX(cmp))

Listing 19: rtl_setcc

```

void rtl_setcc(rtlreg_t* dest, uint8_t subcode) {
    // other code
    switch (subcode & 0xe) {
        case CC_0: // Set byte if overflow (OF=1)
            rtl_get_OF(&t3); break;
        case CC_B: // Set byte if below (CF=1)
            rtl_get_CF(&t3); break;
        case CC_E: // Set byte if equal (ZF=1)
            rtl_get_ZF(&t3); break;
        case CC_BE: // Set byte if below or equal (CF=1) or (ZF=1)
            rtl_get_CF(&t2); rtl_get_ZF(&t3); rtl_or(&t3,&t2,&t3); break;
        case CC_S: // Set byte if sign (SF=1)
            rtl_get_SF(&t3); break;
        case CC_L: // Set byte if less (SF OF)
            rtl_get_SF(&t2); rtl_get_OF(&t3); rtl_xor(&t3,&t2,&t3); break;
        case CC_LE: // Set byte if less or equal (ZF=1 and SF OF)
            rtl_get_SF(&t2); rtl_get_OF(&t3); rtl_xor(&t3,&t2,&t3);
            rtl_get_ZF(&t2); rtl_or(&t3,&t2,&t3); break;
        default: panic("should not reach here");
        case CC_P: panic("n86 does not have PF");
    }
}

```



```
rtl_lr(dest,&t3,1);
// other code
}
```

3.2.8 MOVZX/MOVSX

MOVZX 的操作码为 0x0FB6(MOVZX r32,r/m8) 和 0x0FB7(MOVZX r32,r/m16), 二者数据流均为 E2G, 但操作数宽度不同。因此在对应位置分别插入IDEXW(mov_E2G, movzx, 1), IDEXW(mov_E2G, movzx, 2)即可。

MOVSX 的实现类似 MOVZX, 在对应位置插入IDEXW(mov_E2G, movsx, 1), IDEXW(mov_E2G, movsx, 2)即可。

3.2.9 TEST

操作码为 0x85, 源操作数 1 为 r32(G), 源操作数 2 为 r/m32(E), 因此在对应位置插入IDEX(G2E, test)即可。操作码为 0x84 的类似, 但是操作数宽度为 8 比特, 因此插入IDEXW(G2E, test,1)。其中exec_test的实现类似exec_and, 只是不将结果保存在目的操作数中。

3.2.10 Jcc

实现的操作码从 0x70 到 0x7F(操作数长度指定为 8 比特, 插入IDEXW(J, jcc,1)), 0x0F80 到 0x0F8F(插入IDEX(J, jcc))。其中关于条件的判断借助rtl_setcc实现, 解码借助decode_J实现, 这二者此前已被实现。

3.2.11 POP

实现的操作码从 0x58 到 0x5F, 实现类似 PUSH 指令, 在opcode_table对应位置插入IDEX(r, pop)即可。其中exec_pop的实现见代码段20。其余操作码类似。

Listing 20: exec_pop的实现

```
make_EHelper(pop) {
    rtl_pop(&t3);
    operand_write(id_dest, &t3);

    print_asm_template1(pop);
}
```

3.2.12 INC

根据操作码和指令格式，可得 INC 指令及其opcode_table插入项内容，如表9所示。其中exec_inc的实现见代码段21。

Opcode	Instruction	Table Item
FE /0	INC r/m8	IDEXW(E, gp4, 1)
FF /0	INC r/m16	IDEX(E, gp5) gp5[0]: EXW(inc, 2)
FF /6	INC r/m32	IDEX(E, gp5) gp5[6]: EXW(inc, 4)
40 + rw	INC r16	IDEX(r,inc)
40 + rd	INC r32	IDEX(r,inc)

表 9: INC 指令实现定义

Listing 21: exec_inc的实现

```
make_EHelper(inc) {
    rtl_addi(&t2, &id_dest->val, 1);
    operand_write(id_dest, &t2);

    rtl_update_ZFSF(&t2, id_dest->width);

    rtl_eqi(&t1, &id_dest->val, -1);
    rtl_set_CF(&t1);

    rtl_slt(&t1, &t2, &id_dest->val);
    rtl_set_OF(&t1);

    print_asm_template1(inc);
}
```

3.2.13 ADC

根据操作码和指令格式，可得 INC 指令及其opcode_table插入项内容，如表10所示。其中exec_adc已经被实现。

3.2.14 XOR

根据操作码和指令格式，可得 XOR 指令及其opcode_table插入项内容，如表11所示。其中exec_xor此前已经被实现。

Opcode	Instruction	Table Item
14 ib	ADC AL,imm8	IDEXW(I2a, adc, 1)
15 iw	ADC AX,imm16	IDEX(I2a, adc)
15 id	ADC EAX,imm32	
80 /2 ib	ADC r/m8,imm8	IDEXW(I2E, gp1, 1)
81 /2 iw	ADC r/m16,imm16	IDEX(I2E, gp1)
81 /2 id	ADC r/m32,imm32	
83 /2 ib	ADC r/m16,imm8	IDEX(SI2E, gp1)
	ADC r/m32,imm8	
10 /r	ADC r/m8,r8	IDEXW(G2E, adc, 1)
11 /r	ADC r/m16,r16	IDEX(G2E, adc)
	ADC r/m32,r32	
12 /r	ADC r8,r/m8	IDEXW(E2G, adc, 1)
13 /r	ADC r16,r/m16	IDEX(E2G, adc)
	ADC r32,r/m32	

表 10: ADC 指令实现定义（其中 gp1[2] 为 EX(adc)）

3.2.15 OR

根据操作码和指令格式，可得 XOR 指令及其opcode_table插入项内容，类似指令 ADD，这里不再写重复性内容。其中exec_or的实现见代码段22。

Listing 22: exec_or的实现

```
make_EHelper(or) {
    rtl_or(&t2, &id_dest->val, &id_src->val);
    operand_write(id_dest, &t2);

    rtl_set_CF(&tzero);
    rtl_set_OF(&tzero);
    rtl_update_ZFSF(&t2, id_dest->width);

    print_asm_template2(or);
}
```

3.2.16 SAL/SAR/SHL/SHR

由于opcode_table相应的表项已经填好了，只需要再 gp2 中填入对应的执行函数即可。其中 SAL 和 SHL 指令的操作码完全一样，相应的执行函数借助实现好的 rtl 指令即可实现。

Opcode	Instruction	Table Item
34 ib	XOR AL,imm8	IDEXW(I2a, xor, 1)
35 iw	XOR AX,imm16	IDEX(I2a, xor)
35 id	XOR EAX,imm32	
80 /6 ib	ADC r/m8,imm8	IDEXW(I2E, gp1, 1)
81 /6 iw	ADC r/m16,imm16	IDEX(I2E, gp1)
81 /6 id	ADC r/m32,imm32	
83 /6 ib	XOR r/m16,imm8	IDEX(SI2E, gp1)
	XOR r/m32,imm8	
30 /r	XOR r/m8,r8	IDEXW(G2E, xor, 1)
31 /r	XOR r/m16,r16	IDEX(G2E, xor)
	XOR r/m32,r32	
32 /r	XOR r8,r/m8	IDEXW(E2G, xor, 1)
33 /r	XOR r16,r/m16	IDEX(E2G, xor)
	XOR r32,r/m32	

表 11: XOR 指令实现定义（其中 gp1[2] 为 EX(adc)）

3.2.17 DEC

DEC 指令的实现与 INC 指令对称，只在操作数长度上略有不同。

3.2.18 LEAVE

LEAVE 指令的操作就是"Set ESP to EBP, then pop EBP"，共两个操作的结合体，用于退出函数。在对应位置插入EX(leave)即可，执行函数实现类似 MOV 和 POP。

3.2.19 NOT

由于opcode_table相应的表项已经填好了，只需要再 gp3 中填入对应的执行函数即可。相应执行函数借助rtl_not即可实现。

3.2.20 JMP

与特权指令无关的 JMP 指令的操作码主要分为两类:JMP r/m16/32和JMP rel16/32。由于两者操作码不同，需要使用不同的解码函数和执行函数，前者使用decode_E和exec_jump_rm，后者使用decode_J和exec_jump。

3.2.21 MUL/IMUL/DIV/IDIV

根据不同的操作数类型需要使用不同的解码函数，包括decode_E、decode_E2G、decode_I_E2G。由于操作数为 r/m 类型，在 gp3 中分别对应使用EX(mul)，EX(imul1)，EX(div)，EX(idiv)作为执行函数，其他解码函数根据操作数数量使用exec_imul2或exec_imul3作为执行函数。

其中，需要对rtl_sext进行实现，根据注释提示，其实现见代码段23。

Listing 23: rtl_sext的实现

```
static inline void rtl_sext(rtlreg_t* dest, const rtlreg_t* src1, int
    width) {
    // dest <- signext(src1[(width * 8 - 1) .. 0])
    int shift=8*(4-width);
    *dest = (((int32_t)*src1) << shift) >> shift);
}
```

3.2.22 CWD/CDQ

符号扩展和数据移动操作，填入opcode_table对应项即可。i386 手册和 nemu 中名称有所不同。

3.2.23 SUB/SBB

SUB/SBB 指令其他操作码的实现和上文SUB指令的实现方法一致,填入opcode_table对应项即可。

4 阶段三

4.1 Differential Testing

NEMU 中的 Differential Testing 简单来说就是对比 NEMU 和 QEMU 的执行差异，当 CPU 状态不同时暂停运行。实现上，qemu 的寄存器状态保存在gdb_regs中，只需要在difftest_step中插入比较 cpu 状态的代码 (见代码段24)。

Listing 24: difftest_step的实现

```
diff = r.eip != cpu.eip
    || r.eax != cpu.eax
    || r.ebx != cpu.ebx
    || r.ecx != cpu.ecx
    || r.edx != cpu.edx
```

```

|| r.esp != cpu.esp
|| r.ebp != cpu.ebp
|| r.esi != cpu.esi
|| r.edi != cpu.edi
|| r.eflags != cpu.eflags.reg
/* TDOD: 段寄存器未实现 */
;

```

通过对 EFLAGS 的差异测试，又发现并解决了不少 BUG。

4.2 定位死循环

自己思考的话，定位死循环可能有两种思路，一种是根据状态的重复来检测死循环，一种是根据运行的时间来检测死循环。如果根据状态来检测死循环，就记录跳转指令时 cpu 的状态（也应该包括相关内存状态，但是记录内存状态会比较困难），检测某以跳转指令位置出现相同状态次数过多时，可以初步判断可能是进入死循环。如果根据时间来检测死循环，就是在某一给循环内陷入过多时间就认为是死循环（但是许多程序本身就是死循环，这种方法不好区分）。

4.3 输入输出

4.3.1 理解 volatile 关键字

实验指导书中给出了一段示例代码（见代码段25），来引出 volatile 关键字的应用，查看其 O2 即便编译的结果（见代码段26），和去掉 volatile 关键字后的编译结果（见代码段27）。如果不适应 volatile 关键字，while 之后的 p 指针的操作就被当作死代码被删除了，如果该地址映射到一个设备，那么代码不能正常工作。

Listing 25: difftest_step的实现

```

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

```

Listing 26: 保留volatile关键字O2级别编译

```

fun:
    mov     BYTE PTR ds:134516736, 0

```

```
.L2:
    movzx    eax, BYTE PTR ds:134516736
    cmp      al, -1
    jne      .L2
    mov      BYTE PTR ds:134516736, 51
    mov      BYTE PTR ds:134516736, 52
    mov      BYTE PTR ds:134516736, -122
    ret
```

Listing 27: 去除volatile关键字O2级别编译

```
fun:
    mov      BYTE PTR ds:134516736, 0
.L2:
    jmp      .L2
```

4.3.2 运行 Hello World

首先实现 in, out 指令, 并在它们的 helper 函数中分别调用 pio_read() 和 pio_write() 函数。首先在根据操作数类型在 opcode_table 中选择对于解码函数插入表项, 如下。

Listing 28: opcode_table表项

```
/* 0xe4 */ IDEXW(in_I2a, in, 1), IDEX(in_I2a, in), IDEXW(out_a2I, out, 1), IDEX(out_a2I, out),
/* 0xec */ IDEXW(in_dx2a, in, 1), IDEX(in_dx2a, in), IDEXW(out_a2dx, out, 1), IDEX(out_a2dx, out),
```

然后分别实现 in 后 out 指令的执行函数。分别调用 pio_read() 和 pio_write() 函数即可实现, 如下。

Listing 29: in和out的执行函数

```
make_EHelper(in) {
    t2 = pio_read(id_src->val, id_dest->width);
    operand_write(id_dest, &t2);
    // log and debug
}

make_EHelper(out) {
    pio_write(id_dest->val, id_dest->width, id_src->val);
    // log and debug
}
```

最终运行 hello 程序可以输出十次 hello world, 如图3所示。

```

./build/nemu -l /home/aqni/ics2017/nexus-am/apps/hello/build/nemu-log.txt /home/aqni/ics2017/nexus-am/apps/hello/build/hello-x86-nemu.bin
[src/monitor/diff-test/diff-test.c,99,init_diffptest] Connect to QEMU successfully
[src/monitor/monitor.c,65,load_img] The image is /home/aqni/ics2017/nexus-am/apps/hello/build/hello-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 11:01:11, Mar 30 2022
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!
nemu: HIT GOOD TRAP at eip = 0x0010006e

```

图 3: 运行 hello

4.3.3 实现 IOE

这部分需要实现计时功能，而 CPU 可以通过 I/O 指令访问 RTC（注册到 0x48）这一寄存器获取当前时间。注意到在 ioe.c 中 `_ioe_init()` 函数将启动时的时间保存在全局变量 `boot_time` 中，只需要采用相同方法获得当前时间即可进行计时，如下。

Listing 30: `_uptime` 的实现

```

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

```

运行 `timetest` 程序，1 秒输出一行字符串，实现正常。如图4。

```

[src/monitor/diff-test/diff-test.c,99,init_diffptest] Connect to QEMU successfully
[src/monitor/monitor.c,65,load_img] The image is /home/aqni/ics2017/nexus-am/test/s/timetest/build/timetest-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 11:13:49, Mar 30 2022
For help, type "help"
(nemu) c
1 second.
2 seconds.
3 seconds.
4 seconds.
5 seconds.
6 seconds.

```

图 4: `timetest` 运行结果

4.3.4 看看 NEMU 跑多快

为了测试 CPU 运行的快慢，首先取消 `DEBUG` 和 `DIFF_TEST` 宏的定义，然后使用 `ref` 测试集运行 `dhrystone`、`coremark`、`microbench` 三个测试，`x86-nemu` 和 `native` 的性能分数见表12。期间对 `setll`、`sbb`、`or`、`neg`、`rol` 指令的部分实现进行补齐。其中 `neg` 指令需要设置 `OF` 位，因为 `0x8000000` 没有对应的正数，取反会导致溢出。

AM Arch	dhrystone	coremark	microbench
native	103027	106395	88284
x86-NEMU	207	657	769

表 12: 性能测试 (GCC11.2.1)

4.3.5 如何检测多个键同时被按下

实验指导书中提到了这个问题：

在游戏中,很多时候需要判断玩家是否同时按下了多个键,例如 *RPG* 游戏中的八方向行走,格斗游戏中的组合招式等等. 根据键盘码的特性,你知道这些功能是如何实现的吗?

由于通码表示按下，断码表示释放，因此，所有被设置了通码而没有被设置断码的按键处于同时按下的状态。

4.3.6 实现 IOE(2)：键盘

在 `keyboard.c` 中定义了键盘数据寄存器和状态寄存器的端口，检测状态寄存器并读入数据即可，代码如下。

Listing 31: `_read_key`的实现

```
int _read_key() {
    if (inb(I8042_STATUS_PORT) & I8042_STATUS_HASKEY_MASK){
        return inl(I8042_DATA_PORT);
    }else{
        return _KEY_NONE;
    }
}
```

运行 `keytest` 程序的结果正常，见图5。

```
[src/monitor/monitor.c,65,load_img] The image is /home/aqni/ics2017
/nexus-am/tests/keytest/build/keytest-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 14:35:01, Mar 30 202
2
For help, type "help"
(nemu) c
Get key: 29 Q down
Get key: 29 Q up
Get key: 44 S down
Get key: 44 S up
Get key: 44 S down
Get key: 44 S up
Get key: 45 D down
```

图 5: keytest 运行结果

4.3.7 神奇的调色板

实验指导书中提到了这个问题：

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？

由于一个像素存储的是调色版中的索引，所以只要统一修改调色板本身的内容就可以实现键入渐出的效果。

4.3.8 添加内存映射 I/O: VGA

为了实现内存映射 I/O 的功能，需要对访问的内存进行额外判断，若该地址被映射则进行 I/O 读写，否则进行正常读写。

Listing 32: paddr_read和paddr_write的实现

```
uint32_t paddr_read(paddr_t addr, int len) {
    int mmio_no = is_mmio(addr);
    if (mmio_no != -1) {
        return mmio_read(addr, len, mmio_no);
    } else {
        return pmem_rw(addr, uint32_t) & (~0u >> ((4 - len) << 3));
    }
}

void paddr_write(paddr_t addr, int len, uint32_t data) {
    int mmio_no = is_mmio(addr);
    if (mmio_no != -1) {
```

```
mmio_write(addr, len, data, mmio_no);  
} else {  
    memcpy(guest_to_host(addr), &data, len);  
}  
}
```

随后运行 videotest, 结果如图6。

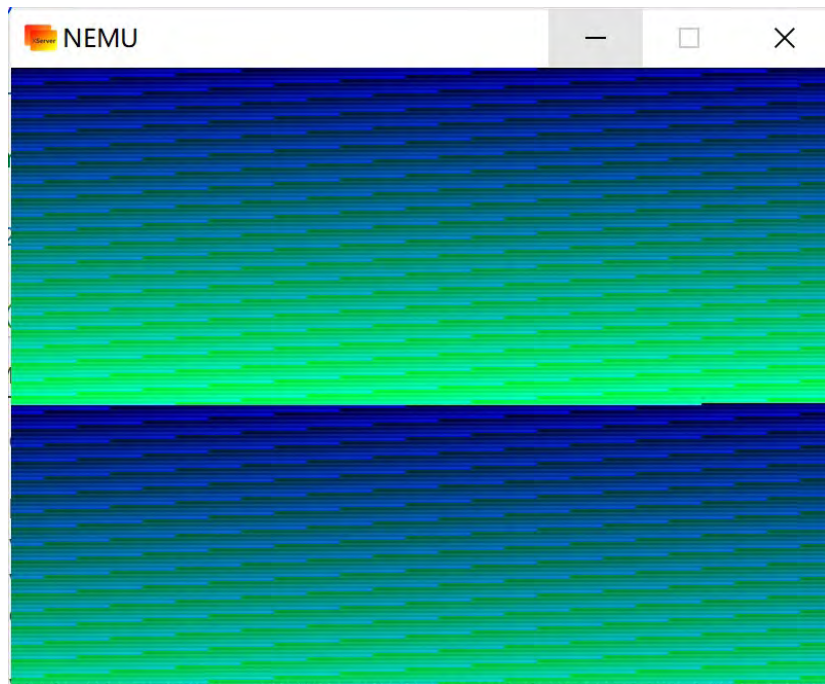


图 6: videotest 运行结果

4.3.9 实现 IOE(3): 动画

在这一节对 `_draw_rect()` 进行正确实现, 代码如下。这样就可以得到正确的视频图像了, 如图7所示。

```
void _draw_rect(const uint32_t *pixels, int x, int y, int w, int h) {  
    for (int row = y; row < y + h; row++)  
        for (int col = x; col < x + w; col++)  
            fb[col+row*_screen.width] = pixels[(row-y)*w+(col-x)];  
}
```

4.3.10 运行打字小游戏

现在可以运行打字小游戏了, 如图8所示。

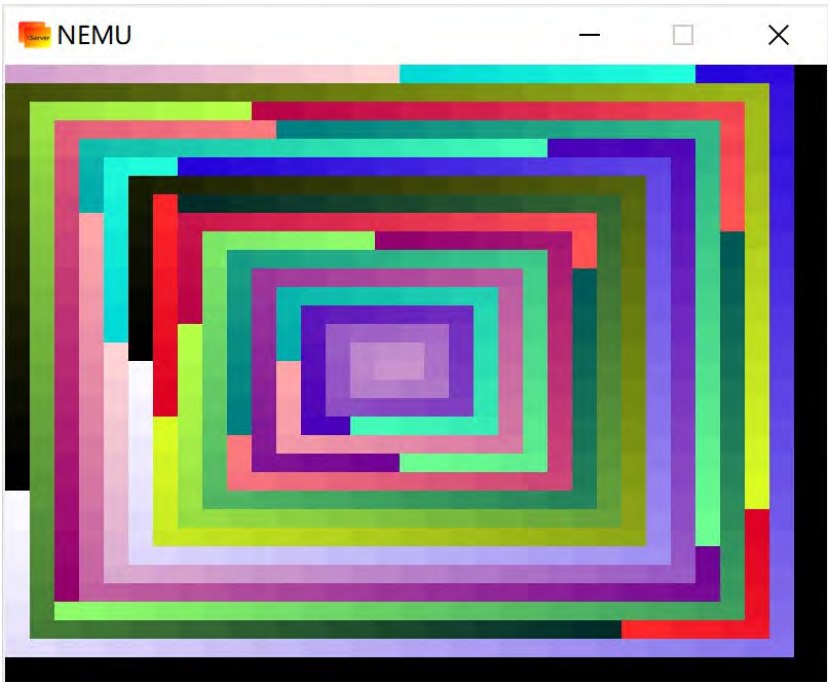


图 7: videotest 正确运行结果



图 8: typing 运行结果

5 必答题

5.1 编译与链接

实验指导书中的原问题是：

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？

下面尝试去掉 `static`，去掉 `inline` 或去掉两者：

- **去掉 `static`**。没有错误，因为该函数被成功内联了。
- **去掉 `inline`**。警告 “defined but not used”，因为 `static` 关键字标识的函数作为本地局部符号，不能被其他文件所访问，因此不会在链接时出现重定义的冲突。
- **去掉两者**。错误 “multiple definition of ‘xxxx’”，不止一个文件包含了 `rtl.h` 文件，因此在不同的源文件中对同一个函数都又一个定义，会出现重定义的错误。

内联证明比较简单，只需要查看其反汇编代码即可指定该函数被内联了。而 `static` 效果的证明需要对多个编译出的可重定位目标文件进行链接，在链接后的结果中查看函数是否有两个不同的代码序列。

例如，我们在两个不同的源文件中定义同一个函数，代码如下。

```
static int add(int a,int b){  
    return a+b;  
}
```

然后生成可重定位目标文件，对该可重定位目标文件反汇编得到如下结果。

```
0000000000000000 <add>:  
0: 55                push    %rbp  
1: 48 89 e5          mov     %rsp,%rbp  
4: 89 7d fc          mov     %edi,-0x4(%rbp)  
7: 89 75 f8          mov     %esi,-0x8(%rbp)  
a: 8b 55 fc          mov     -0x4(%rbp),%edx  
d: 8b 45 f8          mov     -0x8(%rbp),%eax  
10: 01 d0            add     %edx,%eax  
12: 5d                pop     %rbp  
13: c3                ret
```

然后将两个相同的 (其中一个包含 `main`) 可重定位目标文件进行链接，对生成可执行目标文件进行反汇编得到：

```
# 其他代码
0000000000401116 <add>:

# 其他代码

000000000040112a <main>:
    40112a:      55                push    %rbp
    40112b:      48 89 e5          mov     %rsp,%rbp
    40112e:      be 01 00 00 00   mov     $0x1,%esi
    401133:      bf 01 00 00 00   mov     $0x1,%edi
    401138:      e8 d9 ff ff ff   call    401116 <add>
    40113d:      5d                pop     %rbp
    40113e:      c3                ret

000000000040113f <add>:

# 其他代码
```

发现两个函数并没有产生冲突。

5.2 了解 Makefile

实验指导书中的原问题是：

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*。

make 程序根据 *Makefile* 中定义的动作和依赖关系对指定的目标 (默认是第一个目标) 进行生成。这段内容声明了最终的可执行目标文件依赖于一系列可重定位目标文件, 并通过链接行为进行生成。其中 *\$* 符号是 *Makefile* 中的变量, 会替换为具体的字符串。*git_commit* 是在另一个文件 *Makefile.git* 中定义的函数, 该文件被实验 *include* 包含在本文件中。

```
$(BINARY): $(OBJS)
    $(call git_commit, "compile")
    @echo + LD $@
    @$(LD) -O2 -o $@ $^ -lSDL2 -lreadline
```

这段内容定义了一系列可重定位目标文件依赖于对应的源文件, 通过编译动作进行生成。

```
$(OBJ_DIR)/%.o: src/%.c
    @echo + CC $<
    @mkdir -p $(dir $@)
    @$(CC) $(CFLAGS) -c -o $@ $<
```

下面通过 `make -n` 来查看生成过程的细节。发现 `make` 的执行过程就是根据依赖关系自底向上进行生成的过程。

```
echo + CC src/monitor/monitor.c
mkdir -p build/obj/monitor/
gcc -O2 -MMD -Wall -Werror -ggdb -I./include -c -o build/obj/monitor/
    monitor.o src/monitor/monitor.c
此处省略若干行
echo + LD build/nemu
gcc -O2 -o build/nemu build/obj/monitor/monitor.o 此处省略若干obj文件
```

6 BUG 汇总

6.1 直接移位 32 位

实验中，对 `int32` 类型数据直接右移 32 位，结果触发了异常，原因是 `sar` 指令不支持移位超过 31 位的操作，因此需要分两次进行移位。

6.2 diff-test 中的 eflags 初始化

实验中，`qemu` 的 `eflags` 在执行第一条指令时的值为 `0x6`，而 `nemu` 的值为 `0x2`，出现不一致。因此在 `regcpy_from_nemu` 中补充一条 `regs.eflags=cpu.eflags.reg;`。此外，由于 `NEMU` 中 `eflags` 中没有实现 `PF` 位，因此不检查 `PF` 位的差异。

6.3 高版本环境 diff-test 不能正常运行

```
qemu-system-i386: ../util/oslib-posix.c:288: qemu\_set\_nonblock:
    Assertion 'f == 0' failed.
```

参考最新的 `NEMU` 源码，修改 `qemu` 的启动指令解决，如下。

```
execlp("qemu-system-i386", "qemu-system-i386", "-S", "-gdb",
    "tcp::1234", "-nographic", "-serial", "none", "-monitor",
    "none", NULL);
```


6.4 ADC/SBB 指令的 CF 位

假设执行指令 *adcdest,src*，其中 *dest* 为 0，*src* 为 -1，CF 位为 1，则执行后 CF 位的情况：在 QEMU 中 CF 为 1；在 NEMU 原始代码（不是我后来加的，就是他原来的）中 CF 为 0。修复 BUG 后的代码如下，其中修改前的代码用注释标出。SBB 指令同理。

```
make_EHelper(adc) {
    // rtl_add(&t2, &id_dest->val, &id_src->val);
    rtl_add(&t0, &id_dest->val, &id_src->val);
    // rtl_sltu(&t3, &t2, &id_dest->val);
    rtl_sltu(&t3, &t0, &id_dest->val);
    rtl_get_CF(&t1);
    // rtl_add(&t2, &t2, &t1);
    rtl_add(&t2, &t0, &t1);
    operand_write(id_dest, &t2);

    rtl_update_ZFSF(&t2, id_dest->width);

    // rtl_sltu(&t0, &t2, &id_dest->val);
    rtl_sltu(&t0, &t2, &t0);
    rtl_or(&t0, &t3, &t0);

    rtl_set_CF(&t0);

    rtl_xor(&t0, &id_dest->val, &id_src->val);
    rtl_not(&t0);
    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(adc);
}
```

6.5 QEMU 会出现令人费解的情况

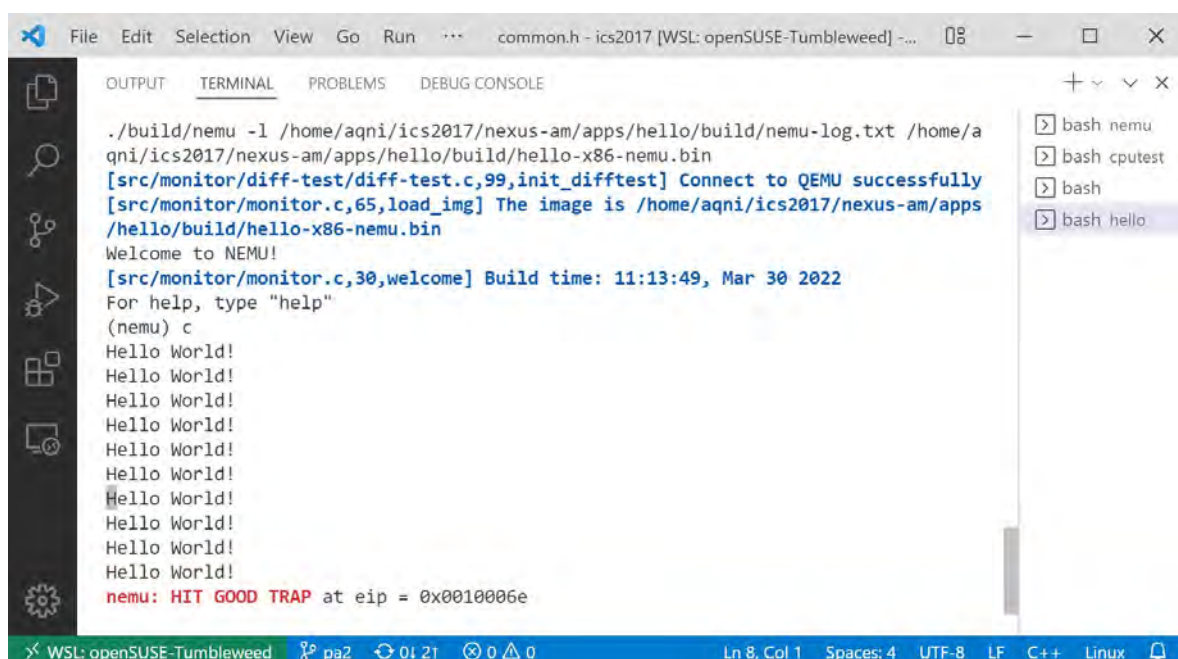
在差异测试中执行指令 *shll %cl,%eax*，其中 *eax* 为 1、*cl* 为 0，理论上 ZF 应当为 0，但是 QEMU 中 ZF 却设为了 1。由于不能解决这种情况，而且 EFLAGS 的比较不是必要的，所以后面不对 EFLAGS 进行差异比较。

6.6 忘记设置 HAS_IOE

在实验过程中，运行 hello 时，忘记设置 HAS_IOE 宏，导致程序中 inb 读取的值始终为 0，进入死循环，不能输出 hello world。

6.7 打开 HAS_IOE 后，WSL 终端卡死

实验时，直接使用了 vscode 远程连接 wsl 的内嵌终端运行 hello 程序，由于无法输出 GUI，导致卡死。在 wsl 中设置 DISPLAY 环境变量为本地 XClient 地址后可以打开 SDL 库产生的窗口并正常输出 hello world，如图9所示。这里暂时使用 GWSL 作为客户端。



```
./build/nemu -l /home/aqni/ics2017/nexus-am/apps/hello/build/nemu-log.txt /home/aqni/ics2017/nexus-am/apps/hello/build/hello-x86-nemu.bin
[src/monitor/diff-test/diff-test.c,99,init_difftest] Connect to QEMU successfully
[src/monitor/monitor.c,65,load_img] The image is /home/aqni/ics2017/nexus-am/apps/hello/build/hello-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 11:13:49, Mar 30 2022
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!
nemu: HIT GOOD TRAP at eip = 0x0010006e
```

图 9: vscode 中 remote-wsl 运行 hello

6.8 高版本 linux 环境下 microbench 出现 FAILED

高版本 linux 环境即使是在 native 架构下，microbench 也不正确，推测是编译器版本过高导致，不过多探究。

7 体会与感悟

随着实验不断深入，高版本环境带来的问题越来越难以解决，后面会回到推荐版本进行实验。

参考文献

- [1] YU Z. Ics2017 programming assignment[EB/OL]. [March 22, 2022]. <https://nju-ics.gitbooks.io/ics2017-programming-assignment/>.