



南开大学
Nankai University

南 开 大 学

计 算 机 学 院
计算机系统设计实验报告

PA3 实验报告

朱浩泽 1911530

年级：2019 级

专业：计算机科学与技术

指导教师：卢冶

2022 年 4 月 22 日

目录

一、 概述	1
(一) 实验目的	1
(二) 实验内容	1
二、 阶段一	1
(一) 环境配置	1
(二) 寄存器传输语言	1
(三) 尝试运行 dummy.c	4
(四) 实验结果	8
三、 阶段二	9
(一) 代码补全	9
(二) Differential Testing	16
(三) 实验结果	17
四、 阶段三	17
五、 感想与体会	17

一、 概述

(一) 实验目的

1. 了解基础设施测试、调试的基本框架与思想
2. 实现 I/O 设备的基本操作
3. 掌握高级语言程序中的各种类型变量对应的表示形式
4. 学习指令周期与指令执行过程，并简单实现现代指令系统
5. 了解冯诺依曼计算机体系结构
6. 在高级语言程序中的变量、机器数和底层硬件（寄存器、加法器、ALU 等）之间建立关联

(二) 实验内容

1. 实现基本的指令
2. 运行第一个 C 程序
3. 补全更多的指令并进行 diff-text
 - (a) 完善 nemu/src/cpu/exec/exec.c 中的 opcode_table
 - (b) 完善 nemu/include/cpu/rtl.h 中的基本操作函数
 - (c) 完善 nemu/src/cpu/* 中的执行函数。（执行函数统一通过宏 make_EHelper 定义）
4. 学习 I/O 的原理，实现屏幕的打印和键盘的输入

二、 阶段一

(一) 环境配置

首先，我们要配置好环境变量，由于我在虚拟机终端中使用的是 zsh（为了使用 oh-my-zsh），所以我们要在 zsh 的配置中（Home/Username/.zshrc）添加如下内容：

```
1 export NEMU_HOME=~/.PA/ics2017/nemu
2 export AM_HOME=~/.PA/ics2017/nexus-am
3 export NAVY_HOME=~/.PA/ics2017/navy-apps
```

(二) 寄存器传输语言

在程序执行过程中，我们都是使用 RTL(寄存器传输语言) 来实现该过程。首先，需要在 nemu/include/cpu/reg.h 文件中补充 EFlags 的标志位

```
1 struct bs {
2     unsigned int CF:1;
3
4     unsigned int one:1;
5     unsigned int :4;
6     unsigned int ZF:1;
7     unsigned int SF:1;
```

```

8
9     unsigned int :1;
10    unsigned int IF:1;
11    unsigned int :1;
12    unsigned int OF:1;
13    unsigned int :20;
14 } 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

```

1 static inline void restart() {
2     /* Set the initial instruction pointer. */
3     cpu.eip = ENTRY_START;
4
5     unsigned int origin = 2;
6     memcpy(&cpu.eflags, &origin, sizeof(cpu.eflags));
7
8     #ifdef DIFF_TEST
9         init_qemu_reg();
10    #endif
11 }

```

```

1 static inline void eflags_modify() {
2     rtl_sub(&t2, &id_dest -> val, &id_src -> val);
3     rtl_update_ZFSF(&t2, id_dest -> width);
4     rtl_sltu(&t0, &id_dest -> val, &id_src -> val);
5     rtl_set_CF(&t0);
6     rtl_xor(&t0, &id_dest->val, &id_src->val);
7     rtl_xor(&t1, &id_dest->val, &t2);
8     rtl_and(&t0, &t0, &t1);
9     rtl_msb(&t0, &t0, id_dest->width);
10    rtl_set_OF(&t0);
11 }

```

rtl 指令在 nemu/include/cpu/rtl.h 文件中。在 rtl.h 中, 指令分为两部分, 一部分是 rtl 指令, 另一部分则是 rtl 伪指令, 它们是通过 rtl 指令实现的。

我们实现 rtl_push 函数让其修改栈顶, 并将指针 src1 中的内容写入栈。

```

1 static inline void rtl_push(const rtlreg_t* src1) {
2     // esp <- esp - 4
3     // M[esp] <- src1
4     //TODO();
5     rtl_subi(&cpu.esp, &cpu.esp, 4);
6     rtl_sm(&cpu.esp, 4, src1);
7 }

```

我们实现 rtl_pop 函数让其将 rtl_pop 读取的数据写入到通用寄存器中。

```

1 static inline void rtl_pop(rtlreg_t* dest) {
2     // dest <- M[esp]
3     rtl_lm(dest, &cpu.esp, 4);
4     // esp <- esp + 4
5     rtl_addi(&cpu.esp, &cpu.esp, 4);
6 }

```

与此同时，我们还实现了其他指令的编写，如下：

EFLAGS 寄存器的标志位读写函数

```

1 #define make_rtl_arith_logic(name) \
2 static inline void concat(rtl_, name) (rtlreg_t* dest, const rtlreg_t* src1, const
   rtlreg_t* src2) { \
3     *dest = concat(c_, name) (*src1, *src2); \
4 } \
5 static inline void concat3(rtl_, name, i) (rtlreg_t* dest, const rtlreg_t* src1, int
   imm) { \
6     *dest = concat(c_, name) (*src1, imm); \
7 }

```

EFLAGS 寄存器的标志位更新函数

```

1 static inline void rtl_eq0(rtlreg_t* dest, const rtlreg_t* src1) {
2     rtl_sltui(dest, src1, 1);
3 }
4
5 static inline void rtl_eqi(rtlreg_t* dest, const rtlreg_t* src1, int imm) {
6     rtl_xori(dest, src1, imm);
7     rtl_eq0(dest, dest);
8 }
9
10 static inline void rtl_neq0(rtlreg_t* dest, const rtlreg_t* src1) {
11     rtl_eq0(dest, src1);
12     rtl_eq0(dest, dest);
13 }
14
15 static inline void rtl_msb(rtlreg_t* dest, const rtlreg_t* src1, int width) {
16     rtl_shri(dest, src1, width*8-1);
17     rtl_andi(dest, dest, 0x1);
18 }
19
20 static inline void rtl_update_ZF(const rtlreg_t* result, int width) {
21     rtl_andi(&t0, result, (0xffffffffu >> (4-width)*8));
22     rtl_eq0(&t0, &t0);
23     rtl_set_ZF(&t0);
24 }
25
26 static inline void rtl_update_SF(const rtlreg_t* result, int width) {
27     assert(result != &t0);
28     rtl_msb(&t0, result, width);
29     rtl_set_SF(&t0);
30 }

```

实现立即数加法

```

1 static inline void rtl_mv(rtlreg_t* dest, const rtlreg_t *src1) {
2     rtl_addi(dest, src1, 0);
3 }

```

实现逻辑非运算

```

1 static inline void rtl_not(rtlreg_t* dest) {
2     rtl_xori(dest, dest, 0xffffffff);
3 }

```

符号拓展，主要与最高位有关

```

1 static inline void rtl_sext(rtlreg_t* dest, const rtlreg_t* src1, int width) {
2     if(width == 0) {
3         rtl_mv(dest, src1);
4     }
5     else {
6         rtl_shli(dest, src1, (4 - width) * 8);
7         rtl_sari(dest, dest, (4 - width) * 8);
8     }
9 }

```

(三) 尝试运行 dummy.c

首先我们什么都不做，运行 `make ARCH=x86-nemu ALL=dummy`，一定是报错。我们通过查看反汇编的代码，可以看出是 `call` 指令和 `endbr` 指令没有实现。我们通过这种方法进行逐一操作，对指令进行补充。

```

+ CC src/monitor/diff-test/protocol.c
+ CC src/monitor/monitor.c
+ CC src/memory/memory.c
+ LD build/nemu
[src/monitor/monitor.c:65,load_img] The image is /home/liighthouse/main/PA/ics2017/nexus-on/tests/cputest/build/dummy-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c:30,welcome] Build time: 16:52:03, Apr 21 2022
For help, type "help"
(nemu) c
invalid opcode(eip = 0x0010000a): e8 01 00 00 00 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
0x0010000a: call 0x0010000a
e8 01 00 00 00 f3 0f
for more details.

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

(nemu) 

```

在这一阶段，我们需要实现的指令有如下

指令	编码	指令	编码
call	e8	xor	31
push	50	pop	58
sub	83	ret	c3

可以看出，每个指令都有自己的编号，这是通过查阅 i386 的指令手册来得到的，其第一个 16 进制数的是指令的 opcode。

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ADD						PUSH	POP	OR						PUSH	2-byte
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	ES	ES	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	CS	escape
1	ADC						PUSH	POP	SBB						PUSH	POP
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	SS	SS	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	DS	DS
2	AND						SEG	DAA	SUB						SEG	DAS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	-ES		Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	-CS	
3	XOR						SEG	AAA	CMP						SEG	AAS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	-ES		Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	-CS	
4	INC general register						DEC general register									
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
5	PUSH general register						POP into general register									
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
6	PUSHA	POPA	BOUND	ARPL	SEG	SEG	Operand	Address	PUSH	IMUL	PUSH	IMUL	INSB	INSW/D	OUTSB	OUTSW/D
			Gv,Ma	Ev,Rv	=FS	=GS	Size	Size	Ib	GvEvIv	Ib	GvEvIv	Yb,DX	Yb,DX	Dx,Xb	DX,Xv
7	Short displacement jump of condition (Jb)						Short-displacement jump on condition (Jb)									

每条指令分别需要执行函数和译码函数，译码函数的主要作用是从 opcode 的后三位中读取通用寄存器的编号，过程是读取 decoding.opcode 中标志的寄存器，将寄存器的内容放入 op->val 中。nemu/src/cpu/exec/exec.c 中的 opcode_table 的每一条记录包括译码函数、执行函数、(两个) 操作数的宽度，如下

```
1 typedef struct {
2     DHelper decode;
3     EHelper execute;
4     int width;
5 } opcode_entry;
```

我们将在这些指令的编号填写在 opcode_table 中，如下

```
1 /* 0x50 */ IDEX(r,push), IDEX(r,push), IDEX(r,push), IDEX(r,push),
2 /* 0x54 */ IDEX(r,push), IDEX(r,push), IDEX(r,push), IDEX(r,push),
3 //...
4 //call
5 /* 0xe8 */ IDEX(J,call), IDEX(J,jmp), EMPTY, IDEXW(J,jmp,1),
6 ....
```

然后在 nemu/src/cpu/exec/all-instr.h 补全执行函数，如下（一次性全部展示，后续将不再说明）

```
1 make_EHelper(mov);
2
3 make_EHelper(operand_size);
4
5 make_EHelper(inv);
6 make_EHelper(nemu_trap);
7 make_EHelper(call);
8 make_EHelper(call_rm);
9 make_EHelper(push);
10 make_EHelper(pop);
11 make_EHelper(sub);
12 make_EHelper(xor);
13 make_EHelper(ret);
14
15 make_EHelper(endbr);
16
17 make_EHelper(add);
18 make_EHelper(inc);
19 make_EHelper(dec);
```

```

20 make_EHelper(cmp);
21 make_EHelper(neg);
22 make_EHelper(adc);
23 make_EHelper(sbb);
24 make_EHelper(mul);
25 make_EHelper(imul1);
26 make_EHelper(imul2);
27 make_EHelper(imul3);
28 make_EHelper(div);
29 make_EHelper(idiv);
30
31 make_EHelper(not);
32 make_EHelper(and);
33 make_EHelper(or);
34 make_EHelper(xor);
35 make_EHelper(sal);
36 make_EHelper(shl);
37 make_EHelper(shr);
38 make_EHelper(sar);
39 make_EHelper(rol);
40 make_EHelper(setcc);
41 make_EHelper(test);
42
43 make_EHelper(leave);
44 make_EHelper(cld);
45 make_EHelper(cwtl);
46 make_EHelper(movsx);
47 make_EHelper(movzx);
48
49 make_EHelper(jmp);
50 make_EHelper(jmp_rm);
51 make_EHelper(jcc);
52
53 make_EHelper(lea);
54 make_EHelper(nop);
55
56 make_EHelper(in);
57 make_EHelper(out);
58
59 make_EHelper(lidt);
60 make_EHelper(int);
61
62 make_EHelper(pusha);
63 make_EHelper(popa);
64 make_EHelper(iret);
65
66 make_EHelper(mov_store_cr);

```

在补充完函数定义后 nemu/src/cpu/exec 中的执行函数和 nemu/src/cpu/exec/decode.c 中的译码函数进行编写

- push

调用上一节中所写的 rtl_push 执行函数进行写栈，代码如下

```

1 make_EHelper(push) {
2     //TODO();

```



```

3   rtl_push(&id_dest -> val);
4   print_asm_template1(push);
5   }

```

- pop

调用上一节中所写的 rtl_pop 执行函数进行读栈，将读取的数据写入到通用寄存器中，代码如下

```

1 make_EHelper(pop) {
2     // TODO();
3     rtl_pop(&t2);
4     operand_write(id_dest, &t2);
5     print_asm_template1(pop);
6 }

```

- call

为 J 形指令，操作数仅一个立即数。CPU 的跳转目标地址 = 当前 eip+ 立即数 offset。所以我们编写 nemu/src/cpu/decode/decode.c 中的译码函数 make_DHelper(J)，调用 decode_op_SI 函数实现立即数的读取，并更新 jmp_eip。

```

1 make_DHelper(J) {
2     decode_op_SI(eip, id_dest, false);
3     // the target address can be computed in the decode stage
4     decoding.jmp_eip = id_dest->simmm + *eip;
5 }

```

然后我们编写 nemu/src/cpu/exe/control.c 中的执行函数 make_EHelper(call)

```

1 make_EHelper(call) {
2     // the target address is calculated at the decode stage
3     //TODO();
4     rtl_li(&t2, decoding.seq_eip);
5     rtl_push(&t2);
6
7     decoding.is_jump = 1;
8
9     print_asm("call %x", decoding.jmp_eip);
10 }

```

- sub

编写译码函数 make_DHelper(I2a)，调用 decode_op_a 读取 AX/EAX 中的数据写入 id_dest，调用 decode_op_I 读取立即数并存入 id_src

```

1 make_DHelper(I2a) {
2     decode_op_a(eip, id_dest, true);
3     decode_op_I(eip, id_src, true);
4 }

```

编写执行函数 make_EHelper(sub)，调用 eflags_modify() 计算减法并将值写回寄存器

```

1 make_EHelper(sub) {
2     // TODO();
3

```

```

4   eflags_modify();
5   operand_write(id_dest, &t2);
6   print_asm_template2(sub);
7   }

```

- xor

实现执行函数 make_Ehelper(xor)

```

1  make_EHelper(xor) {
2      // TODO();
3      rtl_xor(&t2, &id_dest -> val, &id_src -> val);
4      operand_write(id_dest, &t2);
5
6      rtl_update_ZFSF(&t2, id_dest -> width);
7
8      rtl_set_CF(&tzero);
9      rtl_set_OF(&tzero);
10
11     print_asm_template2(xor);
12 }

```

- ret

实现执行函数 make_EHelper(ret)，用栈的数据修改 IP 的内容，实现近转移

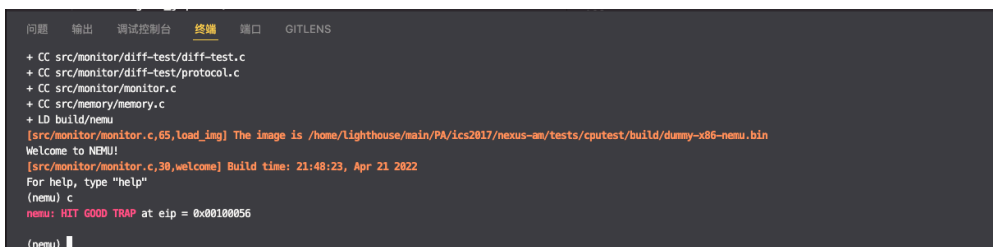
```

1  make_EHelper(ret) {
2      // TODO();
3      rtl_pop(&t2);
4      decoding.jump_eip = t2;
5      decoding.is_jump = 1;
6      print_asm("ret");
7  }

```

(四) 实验结果

在补充完这些所需要的指令后，我们再次执行 `make ARCH=x86-nemu ALL=dummy`，可以看到这时 `dummy.c` 可以在我们的 `nemu` 中正确的运行。



```

问题  输出  调试控制台  终端  窗口  GITLENS
+ CC src/monitor/diff-test/diff-test.c
+ CC src/monitor/diff-test/protocol.c
+ CC src/monitor/monitor.c
+ CC src/memory/memory.c
+ LD build/nemu
[src/monitor/monitor.c:65,load_img] The image is /home/Lighthouse/main/PA/ics2017/nexus-am/tests/cputest/build/dummy-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c:30,welcome] Build time: 21:48:23, Apr 21 2022
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x00100056
(nemu)

```

我们可以看出，指令执行的大概过程是在 `exec.c` 中，我们通过 `exec_wrapper()` 函数开始执行指令，将当前的 `eip` 放进译码信息中（主要是为了诸如 `jmp`，`call` 等转移指令），之后执行 `exec_real()` 函数。在该函数中，我们首先从当前地址中取出一个字节。在指令集执行过程中，我们都是先取出一个字节的操作码，从 `opcode_table` 中寻找匹配项，再执行对应的译码程序和执行程序，完成程序指令的执行。具体来说，就是将当前的 `%eip` 保存到全局译码信息 `decoding` 的成员 `seq_eip` 中，然后将其地址被作为参数送进 `exec_real()` 函数。当代码从 `exec_real()` 返回

时，`decoding.seq_eip` 将会指向下一条指令的地址，调用 `idex()` 对指令进行进一步的译码和执行。

三、 阶段二

在阶段二中，需要我们进一步补全代码，并进行对应的 `diff-test` 对代码正确性进行检验

(一) 代码补全

- `mov`

```
1 make_EHelper(mov) {
2     operand_write(id_dest, &id_src->val);
3     print_asm_template2(mov);
4 }
```

- `leave`

```
1 make_EHelper(leave) {
2     // TODO();
3
4     rtl_mv(&cpu.esp, &cpu.ebp);
5     rtl_pop(&cpu.ebp);
6     print_asm("leave");
7 }
```

- `and`

```
1 make_EHelper(and) {
2     // TODO();
3
4     rtl_and(&t2, &id_dest->val, &id_src->val);
5     operand_write(id_dest, &t2);
6     rtl_update_ZFSF(&t2, id_dest->width);
7     rtl_set_CF(&tzero);
8     rtl_set_OF(&tzero);
9     print_asm_template2(and);
10 }
```

在 i386 手册中为 `CBW` 和 `CWD`，`CBW` 为在 `EAX` 内字节到字的转换，和字到双字的转换，`CWD` 为在 `EAX` 和 `EDX` 间的转换

- `movsx/movzx`

因为 `id_src` 和 `id_dest` 两者的宽度并不相同，所以在进入函数时会重新调整 `id_dest` 的宽度，这意味我们填写译码函数时，按照 `id_src` 的宽度去填写就可以了。（我们是在执行函数中才更正了 `id_dest` 的宽度，`print` 出的结果没有随之更正，但不影响程序执行的结果）

```
1 make_EHelper(movsx) {
2     id_dest->width = decoding.is_operand_size_16 ? 2 : 4;
3     rtl_sext(&t2, &id_src->val, id_src->width);
4     operand_write(id_dest, &t2);
5     print_asm_template2(movsx);
6 }
```

```
7
8 make_EHelper(movzx) {
9     id_dest->width = decoding.is_operand_size_16 ? 2 : 4;
10    operand_write(id_dest, &id_src->val);
11    print_asm_template2(movzx);
12 }
```

- add

add 同 sub，需要修改的地方主要是 CF 和 OF 的判别

```
1 make_EHelper(add) {
2     // TODO();
3
4     rtl_add(&t2, &id_dest->val, &id_src->val);
5     operand_write(id_dest, &t2);
6
7     rtl_update_ZFSF(&t2, id_dest->width);
8
9     rtl_sltu(&t0, &t2, &id_dest->val);
10    rtl_set_CF(&t0);
11
12    rtl_xor(&t0, &id_src->val, &t2);
13    rtl_xor(&t1, &id_dest->val, &t2);
14    rtl_and(&t0, &t0, &t1);
15    rtl_msb(&t0, &t0, id_dest->width);
16    rtl_set_OF(&t0);
17    print_asm_template2(add);
18 }
```

- inc

```
1 make_EHelper(inc) {
2     // TODO();
3
4     rtl_addi(&t2, &id_dest->val, 1);
5     operand_write(id_dest, &t2);
6     rtl_update_ZFSF(&t2, id_dest->width);
7     rtl_eqi(&t0, &t2, 0x80000000);
8     rtl_set_OF(&t0);
9     print_asm_template1(inc);
10 }
```

- dec

```
1 make_EHelper(dec) {
2     // TODO();
3
4     rtl_subi(&t2, &id_dest->val, 1);
5     operand_write(id_dest, &t2);
6     rtl_update_ZFSF(&t2, id_dest->width);
7     rtl_eqi(&t0, &t2, 0x7fffffff);
8     rtl_set_OF(&t0);
9     print_asm_template1(dec);
10 }
```

- cmp

```

1  make_EHelper(cmp) {
2  // TODO();
3
4  eflags_modify();
5
6  print_asm_template2(cmp);
7  }

```

- neg

二进制补码取反

```

1  make_EHelper(neg) {
2  // TODO();
3
4  rtl_sub(&t2, &tzero, &id_dest->val);
5  rtl_update_ZFSF(&t2, id_dest->width);
6  rtl_neq0(&t0, &id_dest->val);
7  rtl_set_CF(&t0);
8  rtl_eqi(&t0, &id_dest->val, 0x80000000);
9  rtl_set_OF(&t0);
10 operand_write(id_dest, &t2);
11 print_asm_template1(neg);
12 }

```

- or

```

1  make_EHelper(or) {
2  // TODO();
3
4  rtl_or(&t2, &id_dest->val, &id_src->val);
5  operand_write(id_dest, &t2);
6  rtl_update_ZFSF(&t2, id_dest->width);
7  rtl_set_CF(&tzero);
8  rtl_set_OF(&tzero);
9  print_asm_template2(or);
10
11 print_asm_template2(or);
12 }

```

- not

```

1  make_EHelper(not) {
2  // TODO();
3
4  rtl_not(&id_dest->val);
5  operand_write(id_dest, &id_dest->val);
6  print_asm_template1(not);
7  }

```

- cltd

当于 cdq 指令，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

```

1 make_EHelper(cld) {
2     if (decoding.is_operand_size_16) {
3         // TODO();
4         rtl_lr_w(&t0, R_AX);
5         rtl_sext(&t0, &t0, 2);
6         rtl_sari(&t0, &t0, 31);
7         rtl_sr_w(R_DX, &t0);
8     }
9     else {
10        // TODO();
11        rtl_sari(&cpu.edx, &cpu.eax, 31);
12    }
13
14    print_asm(decoding.is_operand_size_16 ? "cwtl" : "cltd");
15 }

```

- leave

将栈指针指向帧指针，然后 pop 备份的原帧指针到%ebp

```

1 make_EHelper(leave) {
2     // TODO();
3
4     rtl_mv(&cpu.esp, &cpu.ebp);
5     rtl_pop(&cpu.ebp);
6     print_asm("leave");
7 }

```

- 补全的指令表

```

1  /* 0x80, 0x81, 0x83 */
2  make_group(gp1,
3      EX(add), EX(or), EX adc), EX(sbb),
4      EX(and), EX(sub), EX(xor), EX(cmp))
5
6  /* 0xc0, 0xc1, 0xd0, 0xd1, 0xd2, 0xd3 */
7  make_group(gp2,
8      EX(rol), EMPTY, EMPTY, EMPTY,
9      EX(shl), EX(shr), EMPTY, EX(sar))
10
11  /* 0xf6, 0xf7 */
12  make_group(gp3,
13      IDEX(test_I, test), EMPTY, EX(not), EX(neg),
14      EX(mul), EX(imul1), EX(div), EX(idiv))
15
16  /* 0xfe */
17  make_group(gp4,
18      EX(inc), EX(dec), EMPTY, EMPTY,
19      EMPTY, EMPTY, EMPTY, EMPTY)
20
21  /* 0xff */
22  make_group(gp5,
23      EX(inc), EX(dec), EX(call_rm), EMPTY,
24      EX(jmp_rm), EMPTY, EX(push), EMPTY)
25

```

```

26  /* 0x0f 0x01*/
27  make_group(gp7,
28      EMPTY, EMPTY, EMPTY, EMPTY,
29      EMPTY, EMPTY, EMPTY, EMPTY)
30
31  /* TODO: Add more instructions!!! */
32
33  opcode_entry opcode_table [512] = {
34      /* 0x00 */ IDEXW(G2E, add, 1), IDEX(G2E, add), IDEXW(E2G, add, 1), IDEX(E2G, add
35      ),
36      /* 0x04 */ IDEXW(I2a, add, 1), IDEX(I2a, add), EMPTY, EMPTY,
37      /* 0x08 */ IDEXW(G2E, or, 1), IDEX(G2E, or), IDEXW(E2G, or, 1), IDEX(E2G, or),
38      /* 0x0c */ IDEXW(I2a, or, 1), IDEX(I2a, or), EMPTY, EX(2byte_esc),
39      /* 0x10 */ IDEXW(G2E, adc, 1), IDEX(G2E, adc), IDEXW(E2G, adc, 1), IDEX(E2G, adc
40      ),
41      /* 0x14 */ IDEXW(I2a, adc, 1), IDEX(I2a, adc), EMPTY, EMPTY,
42      /* 0x18 */ IDEXW(G2E, sbb, 1), IDEX(G2E, sbb), IDEXW(E2G, sbb, 1), IDEX(E2G, sbb
43      ),
44      /* 0x1c */ IDEXW(I2a, sbb, 1), IDEX(I2a, sbb), EMPTY, EMPTY,
45      /* 0x20 */ IDEXW(G2E, and, 1), IDEX(G2E, and), IDEXW(E2G, and, 1), IDEX(E2G, and
46      ),
47      /* 0x24 */ IDEXW(I2a, and, 1), IDEX(I2a, and), EMPTY, EMPTY,
48      /* 0x28 */ IDEXW(G2E, sub, 1), IDEX(G2E, sub), IDEXW(E2G, sub, 1), IDEX(E2G, sub
49      ),
50      /* 0x2c */ IDEXW(I2a, sub, 1), IDEX(I2a, sub), EMPTY, EMPTY,
51      /* 0x30 */ IDEXW(G2E, xor, 1), IDEX(G2E, xor), IDEXW(E2G, xor, 1), IDEX(E2G, xor
52      ),
53      /* 0x34 */ IDEXW(I2a, xor, 1), IDEX(I2a, xor), EMPTY, EMPTY,
54      /* 0x38 */ IDEXW(G2E, cmp, 1), IDEX(G2E, cmp), IDEXW(E2G, cmp, 1), IDEX(E2G, cmp
55      ),
56      /* 0x3c */ IDEXW(I2a, cmp, 1), IDEX(I2a, cmp), EMPTY, EMPTY,
57      /* 0x40 */ IDEX(r, inc), IDEX(r, inc), IDEX(r, inc), IDEX(r, inc),
58      /* 0x44 */ IDEX(r, inc), IDEX(r, inc), IDEX(r, inc), IDEX(r, inc),
59      /* 0x48 */ IDEX(r, dec), IDEX(r, dec), IDEX(r, dec), IDEX(r, dec),
60      /* 0x4c */ IDEX(r, dec), IDEX(r, dec), IDEX(r, dec), IDEX(r, dec),
61      /* 0x50 */ IDEX(r, push), IDEX(r, push), IDEX(r, push), IDEX(r, push),
62      /* 0x54 */ IDEX(r, push), IDEX(r, push), IDEX(r, push), IDEX(r, push),
63      /* 0x58 */ IDEX(r, pop), IDEX(r, pop), IDEX(r, pop), IDEX(r, pop),
64      /* 0x5c */ IDEX(r, pop), IDEX(r, pop), IDEX(r, pop), IDEX(r, pop),
65      /* 0x60 */ EMPTY, EMPTY, EMPTY, EMPTY,
66      /* 0x64 */ EMPTY, EMPTY, EX(operand_size), EMPTY,
67      /* 0x68 */ IDEX(I, push), IDEX(I_E2G, imul3), IDEXW(push_SI, push, 1), IDEX(
68      SI_E2G, imul3),
69      /* 0x6c */ EMPTY, EMPTY, EMPTY, EMPTY,
70      /* 0x70 */ IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc,
71      1),
72      /* 0x74 */ IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc,
73      1),
74      /* 0x78 */ IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc,
75      1),
76      /* 0x7c */ IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc, 1), IDEXW(J, jcc,
77      1),
78      /* 0x80 */ IDEXW(I2E, gp1, 1), IDEX(I2E, gp1), EMPTY, IDEX(SI2E, gp1),
79      /* 0x84 */ IDEXW(G2E, test, 1), IDEX(G2E, test), EMPTY, EMPTY,
80      /* 0x88 */ IDEXW(mov_G2E, mov, 1), IDEX(mov_G2E, mov), IDEXW(mov_E2G, mov, 1),

```

```

        IDEX(mov_E2G, mov),
69  /* 0x8c */ EMPTY, IDEX(lea_M2G, lea), EMPTY, IDEX(E, pop),
70  /* 0x90 */ EX(nop), EMPTY, EMPTY, EMPTY,
71  /* 0x94 */ EMPTY, EMPTY, EMPTY, EMPTY,
72  /* 0x98 */ EMPTY, EX(c1td), EMPTY, EMPTY,
73  /* 0x9c */ EMPTY, EMPTY, EMPTY, EMPTY,
74  /* 0xa0 */ IDEXW(O2a, mov, 1), IDEX(O2a, mov), IDEXW(a20, mov, 1), IDEX(a20, mov
    ),
75  /* 0xa4 */ EMPTY, EMPTY, EMPTY, EMPTY,
76  /* 0xa8 */ IDEXW(I2a, test, 1), IDEX(I2a, test), EMPTY, EMPTY,
77  /* 0xac */ EMPTY, EMPTY, EMPTY, EMPTY,
78  /* 0xb0 */ IDEXW(mov_I2r, mov, 1), IDEXW(mov_I2r, mov, 1), IDEXW(mov_I2r, mov,
    1), IDEXW(mov_I2r, mov, 1),
79  /* 0xb4 */ IDEXW(mov_I2r, mov, 1), IDEXW(mov_I2r, mov, 1), IDEXW(mov_I2r, mov,
    1), IDEXW(mov_I2r, mov, 1),
80  /* 0xb8 */ IDEX(mov_I2r, mov), IDEX(mov_I2r, mov), IDEX(mov_I2r, mov), IDEX(
    mov_I2r, mov),
81  /* 0xbc */ IDEX(mov_I2r, mov), IDEX(mov_I2r, mov), IDEX(mov_I2r, mov), IDEX(
    mov_I2r, mov),
82  /* 0xc0 */ IDEXW(gp2_Ib2E, gp2, 1), IDEX(gp2_Ib2E, gp2), EMPTY, EX(ret),
83  /* 0xc4 */ EMPTY, EMPTY, IDEXW(mov_I2E, mov, 1), IDEX(mov_I2E, mov),
84  /* 0xc8 */ EMPTY, EX(leave), EMPTY, EMPTY,
85  /* 0xcc */ EMPTY, EMPTY, EMPTY, EMPTY,
86  /* 0xd0 */ IDEXW(gp2_1_E, gp2, 1), IDEX(gp2_1_E, gp2), IDEXW(gp2_cl2E, gp2, 1),
    IDEX(gp2_cl2E, gp2),
87  /* 0xd4 */ EMPTY, EMPTY, EX(nemu_trap), EMPTY,
88  /* 0xd8 */ EMPTY, EMPTY, EMPTY, EMPTY,
89  /* 0xdc */ EMPTY, EMPTY, EMPTY, EMPTY,
90  /* 0xe0 */ EMPTY, EMPTY, EMPTY, EMPTY,
91  /* 0xe4 */ IDEXW(in_I2a, in, 1), IDEXW(in_I2a, in, 1), IDEXW(out_a2I, out, 1),
    IDEXW(out_a2I, out, 1),
92  /* 0xe8 */ IDEX(J, call), IDEX(J, jmp), EMPTY, IDEXW(J, jmp, 1),
93  /* 0xec */ IDEXW(in_dx2a, in, 1), IDEX(in_dx2a, in), IDEXW(out_a2dx, out, 1),
    IDEX(out_a2dx, out),
94  /* 0xf0 */ EMPTY, EMPTY, EMPTY, EXW(endbr, 3),
95  /* 0xf4 */ EMPTY, EMPTY, IDEXW(E, gp3, 1), IDEX(E, gp3),
96  /* 0xf8 */ EMPTY, EMPTY, EMPTY, EMPTY,
97  /* 0xfc */ EMPTY, EMPTY, IDEXW(E, gp4, 1), IDEX(E, gp5),
98
99  /*2 byte_opcode_table */
100
101  /* 0x00 */ EMPTY, IDEX(gp7_E, gp7), EMPTY, EMPTY,
102
103  ... ..
104
105  /* 0x80 */ IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc),
106  /* 0x84 */ IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc),
107  /* 0x88 */ IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc),
108  /* 0x8c */ IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc), IDEX(J, jcc),
109  /* 0x90 */ IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E,
    setcc, 1),
110  /* 0x94 */ IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E,
    setcc, 1),
111  /* 0x98 */ IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E,
    setcc, 1),

```



```

112  /* 0x9c */ IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E, setcc, 1), IDEXW(E,
    setcc, 1),
113  /* 0xa0 */ EMPTY, EMPTY, EMPTY, EMPTY,
114  /* 0xa4 */ EMPTY, EMPTY, EMPTY, EMPTY,
115  /* 0xa8 */ EMPTY, EMPTY, EMPTY, EMPTY,
116  /* 0xac */ EMPTY, EMPTY, EMPTY, IDEX(E2G, imul2),
117  /* 0xb0 */ EMPTY, EMPTY, EMPTY, EMPTY,
118  /* 0xb4 */ EMPTY, EMPTY, IDEXW(mov_E2G, movzx, 1), IDEXW(mov_E2G, movzx, 2),
119  /* 0xb8 */ EMPTY, EMPTY, EMPTY, EMPTY,
120  /* 0xbc */ EMPTY, EMPTY, IDEXW(mov_E2G, movsx, 1), IDEXW(mov_E2G, movsx, 2),
121  };

```

- 补全 nemu/src/cpu/exec/cc.c 中的 rtl_setcc 函数

```

1  switch (subcode & 0xe) {
2      case CC_0:
3          rtl_get_OF(dest);
4          break;
5      case CC_B:
6          rtl_get_CF(dest);
7          break;
8      case CC_E:
9          rtl_get_ZF(dest);
10         break;
11     case CC_BE:
12         assert(dest!=&t0);
13         rtl_get_CF(dest);
14         rtl_get_ZF(&t0);
15         rtl_or(dest,dest,&t0);
16         break;
17     case CC_S:
18         rtl_get_SF(dest);
19         break;
20     case CC_L:
21         assert(dest!=&t0);
22         rtl_get_SF(dest);
23         rtl_get_OF(&t0);
24         rtl_xor(dest,dest,&t0);
25         break;
26     case CC_LE:
27         assert(dest!=&t0);
28         rtl_get_SF(dest);
29         rtl_get_OF(&t0);
30         rtl_xor(dest,dest,&t0);
31         rtl_get_ZF(&t0);
32         rtl_or(dest,dest,&t0);
33         break;
34     default:
35         panic("should not reach here");
36     case CC_P:
37         panic("n86 does not have PF");
38 }

```

- 此外，需要补充 nemu/src/cpu/decode.c 中的 make_DopHelper(SI) 函数

```

1 static inline make_DopHelper(I) {
2     /* eip here is pointing to the immediate */
3     op->type = OP_TYPE_IMM;
4     op->imm = instr_fetch(eip, op->width);
5     rtl_li(&op->val, op->imm);
6
7     #ifdef DEBUG
8         snprintf(op->str, OP_STR_SIZE, "$0x%x", op->imm);
9     #endif
10 }

```

(二) Differential Testing

每执行一条指令之后，为了及时地捕捉到 error，让 NEMU 和 QEMU 逐条指令地执行同一个客户程序，都会检查各自的寄存器和内存的状态，根据手册，这里不需要比较 eflags 是否相同，因为有一些指令我们实现的并不一样。

```

1  if(r.eax!=cpu.eax) {
2      diff=true;
3  }
4  if(r.ecx!=cpu.ecx) {
5      diff=true;
6  }
7  if(r.edx!=cpu.edx) {
8      diff=true;
9  }
10 if(r.ebx!=cpu.ebx) {
11     diff=true;
12 }
13 if(r.esp!=cpu.esp) {
14     diff=true;
15 }
16 if(r.ebp!=cpu.ebp) {
17     diff=true;
18 }
19 if(r.esi!=cpu.esi) {
20     diff=true;
21 }
22 if(r.edi!=cpu.edi) {
23     diff=true;
24 }
25 if (diff) {
26     nemu_state = NEMU_END;
27 }
28 if(r.eip!=cpu.eip) {
29     diff=true;
30     Log("different:qemu.eip=0x%x,nemu.eip=0x%x",r.eip,cpu.eip);
31 }

```

(三) 实验结果

```
问题 输出 测试控制 终端 窗口 GITLENS
~/main/PA1cs2017/main 2 ps2
$ ./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!
[ min] PASS!
[ move-c] PASS!
[ newex] 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!
```

四、 阶段三

五、 感想与体会

opcode 有一位的和两位的