



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 21 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
三、 感想与体会	8

## 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: e8 01 00 00 00 f3 0f
call 0x0010000a
endbr
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		SUB						SEG	DAS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	-ES	DAA	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	-CS	
3	XOR						SEG		CMP						SEG	AAS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	-ES	AAA	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->siml + *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  }

```

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 NEHU!
[src/monitor/monitor.c,38,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。

### 三、 阶段二

### 四、 感想与体会

opcode 有一位和两位的