南開大學
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

(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：

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

(二) 寄存器传输语言

在程序执行执行过程中，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

```
struct bs {
  unsigned int CF:1;

  unsigned int one:1;
  unsigned int :4;
  unsigned int ZF:1;
  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

```
#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); \
}
```

EFLAGS 寄存器的标志位更新函数

```
static inline void rtl_eq0(rtlreg_t* dest, const rtlreg_t* src1) {
  rtl_sltui(dest, src1, 1);
}

static inline void rtl_eqi(rtlreg_t* dest, const rtlreg_t* src1, int imm) {
  rtl_xori(dest, src1, imm);
  rtl_eq0(dest, dest);
}

static inline void rtl_neq0(rtlreg_t* dest, const rtlreg_t* src1) {
  rtl_eq0(dest, src1);
  rtl_eq0(dest, dest);
}

static inline void rtl_msb(rtlreg_t* dest, const rtlreg_t* src1, int width) {
  rtl_shri(dest, src1, width*8-1);
  rtl_andi(dest, dest, 0x1);
}

static inline void rtl_update_ZF(const rtlreg_t* result, int width) {
  rtl_andi(&t0, result, (0xffffffffu >> (4-width)*8));
  rtl_eq0(&t0, &t0);
  rtl_set_ZF(&t0);
}

static inline void rtl_update_SF(const rtlreg_t* result, int width) {
  assert(result != &t0);
  rtl_msb(&t0, result, width);
  rtl_set_SF(&t0);
}
```

实现立即数加法

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

实现逻辑非运算

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

符号拓展，主要与最高位有关

```
static inline void rtl_sext(rtlreg_t* dest, const rtlreg_t* src1, int width) {
  if(width == 0) {
    rtl_mv(dest, src1);
  }
  else {
    rtl_shli(dest, src1, (4 - width) * 8);
    rtl_sari(dest, dest, (4 - width) * 8);
  }
}
```

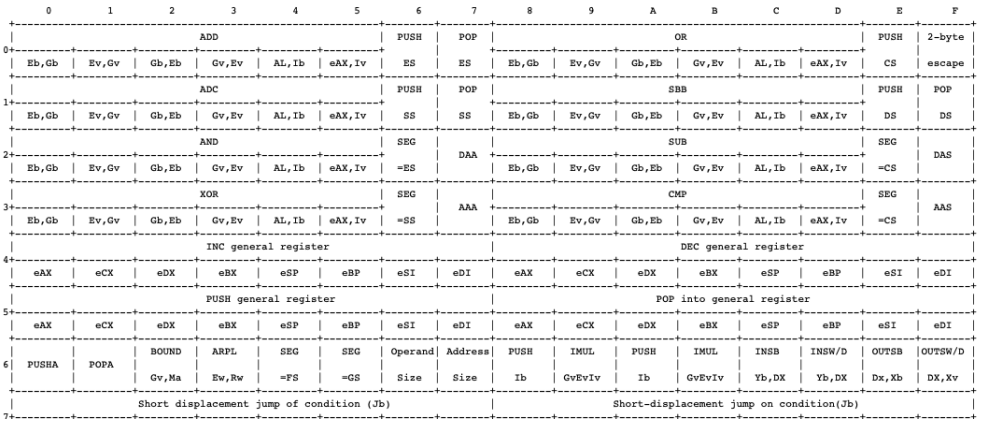
(三) 尝试运行 dummy.c

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。

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。



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

```
typedef struct {
  DHelper decode;
  EHelper execute;
  int width;
} opcode_entry;
```

我们将在这些指令的编号填写在 opcode_table 中，如下

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

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）

```
make_EHelper(mov);

make_EHelper(operand_size);

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

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(cltd);
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);
```

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

```
make_EHelper(push) {
  //TODO();
```

```
    rtl_push(&id_dest -> val);
    print_asm_template1(push);
  }
```

- 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

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

- 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。

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

  然后我们编写 nemu/src/cpu/exe/control.c 中的执行函数 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_jmp = 1;

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

- 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

```
make_DHelper(I2a) {
  decode_op_a(eip, id_dest, true);
  decode_op_I(eip, id_src, true);
}
```

  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

```
make_EHelper(sub) {
  // TODO();

```

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

- xor

  实现执行函数 make_Ehelper(xor)

```
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);
}
```

- 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

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

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。

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 有一位的和两位的