# 南京航空航天大学《计算机组成原理Ⅱ课程设计》报告

- 姓名：傅锦龙
- 班级：1621301
- 学号：162130117
- 报告阶段：PA2.1
- 完成日期：2023.5.24
- 本次实验，我完成了所有内容。

## 目录

# 思考题

## 增加了多少

在不考虑跳转类指令的情况下，x2 - x1 的差值包括了一条指令的操作码和操作数

## 是什么类型

opcode_table每个表项的类型是opcode_entry
opcode_entry.width记录了操作数长度信息
opcode_entry.decode记录了译码函数
opcode_entry.execute记录了执行函数

## 操作数结构体的实现

`Operand`结构体中包含以下各个成员

- `type`：操作数的类型，可以是寄存器、内存或立即数。
- `width`：操作数的宽度，可以是1字节、2字节或4字节。
- `union`：操作数的值，可以是寄存器编号、内存地址、立即数或带符号的立即数。
- `val`：操作数的值，以rtlreg_t类型存储。在执行指令时，CPU会根据操作数的类型和值来访问寄存器、内存或立即数，并执行相应的操作。
- `str`：操作数的字符串表示，用于打印调试信息。

## 复现宏定义

- make_EHelper(mov) //mov 指令的执行函数

```
#define make_EHelper(name) void concat(exec_, name) (vaddr_t *eip)
void exec_mov (vaddr_t *eip)
```

- make_EHelper(push) //push 指令的执行函数

```
#define make_EHelper(name) void concat(exec_, name) (vaddr_t *eip)
void exec_push (vaddr_t *eip)
```

- make_DHelper(I2r) //I2r 类型操作数的译码函数

```
#define make_DHelper(name) void concat(decode_, name) (vaddr_t *eip)
void decode_I2r (vaddr_t *eip)
```

- IDEX(I2a, cmp) //cmp 指令的 opcode_table 表项

```
#define IDEXW(id, ex, w)   {concat(decode_, id), concat(exec_, ex), w}
#define IDEX(id, ex)       IDEXW(id, ex, 0)
{decode_I2a, exec_cmp, 0}
```

- EX(nop) //nop 指令的 opcode_table 表项

```
#define EX(ex)  EXW(ex, 0)
#define EXW(ex, w)  {NULL, concat(exec_, ex), w}
#define NULL ((void *)0)
{((void *)0), exec_nop, 0}
```

- make_rtl_arith_logic(and) //and 运算的 RTL 指令

```
#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); \
}
static inline void rtl_and (rtlreg_t* dest, const rtlreg_t* src1, const
rtlreg_t* src2) {
    *dest = ((*src1) & (*src2));
}
static inline void rtl_andi (rtlreg_t* dest, const rtlreg_t* src1, int imm) {
    *dest = ((*src1) & (imm));
}
```

## 立即数背后的故事

- 要注意机器是大端机还是小端机
- 在大端架构中，字节序是从高位到低位排列的，而在小端架构中，字节序是从低位到高位排列的。
- 在读取和写入数据时，需要根据目标机器的字节序进行转换，以保证数据的正确性。

## 神奇的 eflags

- 当超出表示范围时产生溢出
- CF不能代替OF，因为并不是所有的进位情况都会超出表示范围
- OF=Cn⊕Cn-1

## git branch 和 git log 截图（最新的，一张即可）

```
fujinlong@ubuntu:/mnt/hgfs/course/ics2023/nexus-am/tests/cputest$ git branch
  master
  pa0
* pa2
fujinlong@ubuntu:/mnt/hgfs/course/ics2023/nexus-am/tests/cputest$ git log
commit bba7926773d9d19ef11291cb55ce03709b1287d1
Author: tracer-ics2017 <tracer@njuics.org>
Date:   Wed May 24 17:43:16 2023 +0800

    >  run
    162130117
    fujinlong
    Linux ubuntu 4.15.0-142-generic #146~16.04.1-Ubuntu SMP Tue Apr 13 09:26:57 UTC 2021 i686 athlon i686 GNU/Linux
     17:43:16 up  8:06,  1 user,  load average: 0.12, 0.03, 0.01
    42b18a28b3682f67d2cedf10342cba232b5c5b00

commit 6a7a3ff546ef459881ce6fdce9274a0c51717712
Author: tracer-ics2017 <tracer@njuics.org>
Date:   Wed May 24 17:43:14 2023 +0800

    >  compile
    162130117
    fujinlong
    Linux ubuntu 4.15.0-142-generic #146~16.04.1-Ubuntu SMP Tue Apr 13 09:26:57 UTC 2021 i686 athlon i686 GNU/Linux
     17:43:14 up  8:06,  1 user,  load average: 0.12, 0.03, 0.01
    a97ce9ceca35a3e2762c07754f911b65192f9255
:...skipping...
commit bba7926773d9d19ef11291cb55ce03709b1287d1
Author: tracer-ics2017 <tracer@njuics.org>
Date:   Wed May 24 17:43:16 2023 +0800

    >  run
    162130117
    fujinlong
    Linux ubuntu 4.15.0-142-generic #146~16.04.1-Ubuntu SMP Tue Apr 13 09:26:57 UTC 2021 i686 athlon i686 GNU/Linux
     17:43:16 up  8:06,  1 user,  load average: 0.12, 0.03, 0.01
    42b18a28b3682f67d2cedf10342cba232b5c5b00
```

# 实验内容

## 1.实现标志寄存器

### 实现标志寄存器 eflags

eflags 结构体是用来存储 CPU 的标志寄存器的，而在 NEMU 中, 只会用到 EFLAGS 中以下的 5 个位: CF, ZF, SF, IF, OF。具体实现 eflags 结构体的思路如下：
定义 eflags union，其中结构体根据eflags的结构由五个位域CF, ZF, SF, IF, OF和无名位域组成，val用来赋初值

```
union {
    rtlreg_t val;
    struct {
        rtlreg_t CF : 1;
        rtlreg_t : 5;
        rtlreg_t ZF : 1;
        rtlreg_t SF : 1;
        rtlreg_t : 1;
        rtlreg_t IF : 1;
        rtlreg_t : 1;
        rtlreg_t OF : 1;
    };
} eflags;
```

### eflags设置初值

通过查询i386手册,需要把eflags初始化为0x2,所以在restart函数中添加如下：

```
cpu.eflags.val = 0x2;
```

### 实现所有指令对标志位的设置（如果该指令有设置标志行为）。

对于sub，模仿sbb，除了不需要减去CF外，其余的均相同

```
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);

  print_asm_template2(sub);
}
```

对于xor，需要更新SF，ZF，OF=0、CF=0

```
make_EHelper(xor)
{
    rtl_xor(&id_dest->val, &id_src->val, &id_src2->val);
    operand_write(id_dest, &id_dest->val);

    rtl_li(&t0, 0);
    rtl_set_CF(&t0);
    rtl_set_OF(&t0);
    rtl_update_ZFSF(&id_dest->val, id_dest->width);

    print_asm_template2(xor);
}
```

# 2.实现所有 RTL 指令

## make_rtl_setget_eflags

这个宏会生成两个函数，分别是 rtl_set_f 和 rtl_get_f，其中 f是一个标志位的名称，比如 CF、ZF、SF 等。这两个函数的作用分别是将标志位的值写入到 CPU 标志寄存器中，或者从 CPU 标志寄存器中读取标志位的值。
rtl_set_f 函数的实现思路是将 src 指向的值的最低位（即标志位的值）提取出来，然后将其赋值给 CPU 标志寄存器中对应的标志位。
rtl_get_f函数的实现思路是将 CPU 标志寄存器中对应的标志位的值读取出来，然后将其存储到 dest 指向的内存地址中。

```
#define make_rtl_setget_eflags(f)                                  \
    static inline void concat(rtl_set_, f)(const rtlreg_t *src) \
    {                                                              \
        cpu.eflags.f = *src & 0x1;                                 \
    }                                                              \
    static inline void concat(rtl_get_, f)(rtlreg_t * dest)     \
    {                                                              \
        *dest = cpu.eflags.f;                                      \
    }
```

## rtl_mv

按注释赋值即可

```
static inline void rtl_mv(rtlreg_t *dest, const rtlreg_t *src1)
{
    // dest <- src1
    *dest = *src1;
}
```

## rtl_not

按照注释取非即可

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

## rtl_sext

`rtl_sext` 函数是将 `src1` 指向的值的低 `width` 个字节进行符号扩展，然后将结果存储到 `dest` 指向的内存地址中。
这个函数的实现思路是：

1. 将 `src1` 指向的值赋值给一个临时带符号整数变量 `tmp`。
2. 根据 `width` 的值，选择不同的符号扩展方式，将 `tmp` 左移 (4-width)*8位，然后再右移相同位，这样就将最高位符号位扩展到了高位
3. 将符号扩展后的值存储到 `dest` 指向的内存地址中。

```
static inline void rtl_sext(rtlreg_t *dest, const rtlreg_t *src1, int width)
{
    // dest <- signext(src1[(width * 8 - 1) .. 0])
    int32_t tmp = *src1;
    switch (width) {
        case 1:
            *dest = (tmp<<24)>>24;
            return;
        case 2:
            *dest = (tmp<<16)>>16;
            return;
        case 4:
            *dest = tmp;
            return;
        default:
            assert(0);
    }
}
```

## rtl_push

这个函数的实现思路是：

1. 首先，使用 `rtl_lr_l` 函数将当前栈顶指针 `R_ESP` 的值读取到临时变量 `t0` 中。

2. 然后，使用 `rtl_subi` 函数将 `t0` 减去 4，得到新的栈顶指针，将其存储回 `R_ESP` 中。
3. 接着，使用 `rtl_sm` 函数将 `src1` 指向的值存储到新的栈顶位置，即 `t0` 指向的内存地址中，存储的字节数为 4。

```
static inline void rtl_push(const rtlreg_t *src1)
{
    // esp <- esp - 4
    // M[esp] <- src1
    rtl_lr_l(&t0, R_ESP);
    rtl_subi(&t0, &t0, 4);
    rtl_sr_l(R_ESP, &t0);
    rtl_sm(&t0, 4, src1);
}
```

## rtl_pop

具体实现思路如下：

1. 从寄存器 `R_ESP` 中读取当前栈顶指针的值，保存到临时变量 `t0` 中。
2. 从内存地址 `t0` 中读取 4 字节的值，存储到 `dest` 指向的地址中。
3. 将 `t0` 的值加上 4，表示栈顶指针向下移动 4 个字节。
4. 将新的栈顶指针的值写回寄存器 `R_ESP` 中。

```
static inline void rtl_pop(rtlreg_t *dest)
{
    // dest <- M[esp]
    // esp <- esp + 4
    rtl_lr_l(&t0, R_ESP);
    rtl_lm(dest, &t0, 4);
    rtl_addi(&t0, &t0, 4);
    rtl_sr_l(R_ESP, &t0);
}
```

## rtl_eq0

判断src1是否为0，将结果赋值给dest

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

## rtl_eqi

判断src1与立即数imm是否相等,将结果赋值给dest

```
static inline void rtl_eqi(rtlreg_t *dest, const rtlreg_t *src1, int imm)
{
    // dest <- (src1 == imm ? 1 : 0)
    *dest = (*src1 == imm ? 1 : 0);
}
```

## rtl_neq0

判断src1是否不为0，将结果赋值给dest

```
static inline void rtl_neq0(rtlreg_t *dest, const rtlreg_t *src1)
{
    // dest <- (src1 != 0 ? 1 : 0)
    *dest = (*src1 != 0 ? 1 : 0);
}
```

## rtl_msb

`rtl_msb`作用是将一个指定宽度的整数的最高位（即最高有效位）提取出来，并存储到指定的地址`dest`中。具体实现思路如下：

1. 根据参数`width`的值，选择相应的位移量，将`src1`中的最高位移动到最低位。
2. 将移动后的最低位与 0x1 进行按位与操作，得到最高位的值。
3. 将得到的最高位的值存储到`dest`指向的地址中。

```
static inline void rtl_msb(rtlreg_t *dest, const rtlreg_t *src1, int width)
{
    // dest <- src1[width * 8 - 1]
    switch (width) {
        case 1:
            *dest = (*src1 >> 7) & 0x1;
            return;
        case 2:
            *dest = (*src1 >> 15) & 0x1;
            return;
        case 4:
            *dest = (*src1 >> 31) & 0x1;
            return;
        default:
            assert(0);
    }
}
```

## rtl_update_ZF

具体实现思路如下：

1. 根据参数`width`的值，计算出需要检查的位数，即`width * 8`。
2. 将`result`左移`32 - width * 8`位，使得需要检查的位数位于最高位。
3. 调用`rtl_eq0`函数，将`tmp`中的值与 0 进行比较，得到一个布尔值，表示`result`中需要检查的位是否全部为 0。
4. 将得到的布尔值存储到标志寄存器`eflags`的零标志位`ZF`中。

```
static inline void rtl_update_ZF(const rtlreg_t *result, int width)
{
    // eflags.ZF <- is_zero(result[width * 8 - 1 .. 0])
    rtlreg_t tmp = (*result << (32 - width * 8));
    rtl_eq0(&tmp, &tmp);
    rtl_set_ZF(&tmp);
}
```

## rtl_update_SF

`rtl_update_SF`作用是根据指定宽度的整数`result`的值更新标志寄存器`eflags`中的符号标志位`SF`。具体实现思路如下：

1. 根据参数`width`的值，计算出需要检查的位数，即`width * 8`。
2. 调用`rtl_msb`函数，将`result`中的最高位提取出来，存储到临时变量`tmp`中。
3. 将得到的最高位的值存储到标志寄存器`eflags`的符号标志位`SF`中。

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

# 3.实现 6 条 x86 指令

## call

选取译码函数make_DHelper(I)用于获取一个立即数即目标地址并将其存入idest中
实现执行函数make_EHelper(call)，具体实现思路如下：

1. 调用`rtl_push`函数，将当前指令的下一条指令的地址`eip`压入栈中，以便函数返回时能够正确返回到调用点。
2. 将全局变量`decoding`中的`is_jmp`字段设置为 1，表示当前指令是一条跳转指令。
3. 调用`rtl_add`函数，将当前指令的下一条指令的地址`eip`和目标地址`id_dest->val`相加，得到函数的入口地址，并将其存储到全局变量`decoding`中的`jmp_eip`字段中。

```
make_EHelper(call) {
  // the target address is calculated at the decode stage
  rtl_push(eip);
  decoding.is_jmp = 1;
  rtl_add(&decoding.jmp_eip,eip, &id_dest->val);
  print_asm("call %x", decoding.jmp_eip);
}
```

填写opcode_table，由于只实现CALL rel32，所以width可默认为0，从而省略

```
IDEX(I,call)
```

实现结果

```
(nemu) si
  10000a:   e8 0f 00 00 00                              call 10001e
```

## push

选取译码函数make_DHelper(r)，用于读取寄存器信息。
实现执行函数make_EHelper(push)，使用rtl_push函数把取到的操作数入栈

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

填写opcode_table，由于只需实现 PUSH r32，所以可省略width

```
IDEX(r,push)
```

运行结果

```
(nemu) si
  10001e:    55                                                  pushl %ebp
```

## pop

选取译码函数make_DHelper(r)，用于读取寄存器信息。
选取执行函数make_EHelper(pop)，使用rtl_pop指令把栈顶内容存入id_dest

```
make_EHelper(pop) {
  rtl_pop(&id_dest->val);
  operand_write(id_dest, &id_dest->val);
  print_asm_template1(pop);
}
```

填写opcode_table，由于只需实现 POP r32，所以可省略width

```
IDEX(r,pop)
```

运行结果

```
(nemu) si
  100013:    5d                                                  popl %ebp
```

## sub

opcode_table中使用了grp1，查表可知grp1[5]应填写执行函数make_EHelper(sub)
实现解码函数make_DopHelper(SI)，按照注释读取指定宽度字节的内存到op->simm中

```
 /* 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 = ???
   */
  op->simm = instr_fetch(eip, op->width);
```

实现执行函数make_EHelper(sub)，设目标操作数为x，源操作数为y，具体思路如下：

1. 它通过调用rtl_sub函数计算目标操作数减去源操作数的结果，并通过调用operand_write函数将计算结果写回目标操作数。
2. 它通过调用rtl_update_ZFSF(x-y)函数更新零标志位（ZF）和符号标志位（SF）的值，通过调用rtl_sltu函数得到x<(x-y)的结果并更新进位标志位（CF）的值，最后得到(x^y)&(x^(x-y))结果并更新溢出标志位（OF）的值。

```
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);

  print_asm_template2(sub);
}
```

填写gpr1

```
make_group(gp1,
    EMPTY, EMPTY, EMPTY, EMPTY,
    EMPTY, EX(sub), EMPTY, EMPTY)
```

运行结果

```
(nemu) si
  100021:   83 ec 18                                subl $0x18,%esp
```

## xor

实现执行函数make_EHelper(xor)，具体思路如下：

1. 对源操作数和源操作数2进行异或运算，并将结果存储到目标操作数中
2. 通过调用`rtl_update_ZFSF`函数更新零标志位（ZF）和符号标志位（SF）的值，通过调用`rtl_set_CF`和`rtl_set_OF`函数将进位标志位（CF）和溢出标志位（OF）的值都设置为0。

```
make_EHelper(xor)
{
    rtl_xor(&id_dest->val, &id_src->val, &id_src2->val);
    operand_write(id_dest, &id_dest->val);

    rtl_li(&t0, 0);
    rtl_set_CF(&t0);
    rtl_set_OF(&t0);
    rtl_update_ZFSF(&id_dest->val, id_dest->width);

    print_asm_template2(xor);
}
```

## ret

由手册可知ret无译码函数
实现执行函数make_EHelper(ret)，由于需要跳转回上次call指令的下条指令处，需要设置跳转标志，并把要跳转的位置出栈赋给decoding.jmp_eip

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

填写opcode_table

```
EX(ret)
```

运行结果

```
(nemu) si
  100014:    c3                                                   ret
```

## 4.成功运行 dummy

```
fujinlong@ubuntu:/mnt/hgfs/course/ics2023/nexus-am/tests/cputest$ make ARCH=x86-
nemu ALL=dummy run
Building dummy [x86-nemu]
make[1]: Warning: File 'Makefile.dummy' has modification time 2.2 s in the futur
e
Building am [x86-nemu]
make[2]: *** 没有指明目标并且找不到 makefile。 停止。
[src/monitor/monitor.c,68,load_img] The image is /mnt/hgfs/course/ics2023/nexus-
am/tests/cputest/build/dummy-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,33,welcome] Build time: 19:28:24, May 23 2023
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x0010001b
```

## 5.实现 Diff-test

首先需要在在 nemu/include/common.h 中定义宏 DIFF_TEST
在nemu/src/monitor/diff-test/diff-test.c中先定义一个宏定义

```
#define diff_reg(reg)                                          \
    if (cpu.reg != r.reg) {                                    \
        diff = true;                                           \
        printf("NEMU.reg=%#x,QEMU.reg=%#x\n", cpu.reg, r.reg); \
    }
```

再修改difftest_step()，加入以下内容

```
diff_reg(eip);
diff_reg(eax);
diff_reg(ecx);
diff_reg(edx);
diff_reg(ebx);
diff_reg(ebp);
diff_reg(esp);
diff_reg(edi);
diff_reg(esi);
```

运行结果

```
fujinlong@ubuntu:/mnt/hgfs/course/ics2023/nexus-am/tests/cputest$ make ARCH=x86-
nemu ALL=dummy run
Building dummy [x86-nemu]
make[1]: Warning: File 'Makefile.dummy' has modification time 2.4 s in the futur
e
Building am [x86-nemu]
make[2]: *** 没有指明目标并且找不到 makefile。 停止。
+ CC src/monitor/diff-test/diff-test.c
+ LD build/nemu
[src/monitor/diff-test/diff-test.c,105,init_difftest] Connect to QEMU successful
ly
[src/monitor/monitor.c,68,load_img] The image is /mnt/hgfs/course/ics2023/nexus-
am/tests/cputest/build/dummy-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,33,welcome] Build time: 16:54:46, May 24 2023
For help, type "help"
(nemu) c
nemu: HIT GOOD TRAP at eip = 0x0010001b
```

# 遇到的问题及解决办法

1. 遇到问题：ret指令不知道如何实现
   解决方案：根据call指令获取跳转地址并设置跳转指令
2. 遇到问题：文件在本地修改后，传输到虚拟机后运行仍然是之前的错误
   解决方案：先执行make clean,再运行

# 实验心得

这次实验难度较大，但通过这次实验我对于对之前的寄存器和汇编指令的知识有了更加深入的了解，能够更加顺畅地阅读大的项目，对于我未来的计算机课程学习有着一定的帮助。

# 其他备注

助教真帅