南京航空航天大学《计算机组成原理Ⅱ课程设计》报告

- 姓名：唐希
- 班级：1619201
- 学号：161920122
- 报告阶段：PA2.1
- 完成日期：2021.5.13
- 本次实验，我完成了所有内容。

目录

思考题和git

1.增加了多少

包括前缀,操作码,操作数。

2.是什么类型

类型是opcode_entry,结构体定义为

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

操作数长度是int width,译码函数指针是decode,执行函数指针是execute。

3.操作数的结构体实现

结构体和他们储存的信息如下：

```
typedef struct {
uint32_t type;//操作数类型
int width;//操作数长度
union {
uint32_t reg;//寄存器寻址
rtlreg_t addr;//内存寻址
uint32_t imm;//无符号立即数
int32_t simm;//有符号立即数
};
rtlreg_t val;//操作数解码后的值
char str[OP_STR_SIZE];//操作数的字符串表达
} Operand;
```

通过共同体成员共用内存空间的性质来实现复用。

4.复现宏定义

1. 复现宏定义

   make_EHelper(push) //push 指令的执行函数`void exec_mov (vaddr_t *eip)`

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

   `concat`连接为`exec_name`的形式

   `void exec_push (vaddr_t *eip)`

   同上

   make_DHelper(I2r) //I2r`void decode_I2r (vaddr_t *eip)`

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

   `concat`连接为`decode_name`的形式

   IDEX(I2a, cmp) //cmp 指令的 opcode_table 表项`{decode_I2a, exec_cmp, 0}`

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

   EX(nop) //nop 指令的 opcode_table 表项`{((void *)0), exec_nop, 0}`

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

make_rtl_arith_logic(and) //and 运算的 RTL 指令

```
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));
}
```

```
#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); \
  }
```

5.立即数背后的故事

自然地，应该考虑模拟器的运行环境和模拟的环境中大小端方式不匹配的问题。以一种情况我们在instr_fetch读取字节序列时采用逆序读取的方式放到存储变量中。第二种情况我们就在取出立即数的时候逆序放到存储变量中。

6.神奇的eflags

“溢出”的意思是计算的结果超出了结果变量类型的表示范围；不能，因为不溢出的时候也会发生进位；两个运算数符号相同但结果的符号不同时，OF就是1。

```
OF=((a>0&&b>0&&((a+b)<a||(a+<b)<b))||(a<0&&b<0&&((a-b)>a||(a-b)>b)))
```

7. `git log`和远程git仓库提交截图

tangxi@debian: ~/ics2021

```
tangxi@debian:~/ics2021$ git log
commit 8a8ab54a4fd3025d30e09c808825b5a86b9c0499 (HEAD -> pa2, myrepo/pa2)
Author: tracer-ics2017 <tracer@njuics.org>
Date:   Mon May 17 10:41:28 2021 +0800

    >  run
    161920122
    tangxi
    Linux debian 4.19.0-14-686 #1 SMP Debian 4.19.171-2 (2021-01-30) i686 GNU/Linux
     10:41:28 up 45 min,  1 user,  load average: 0.00, 0.00, 0.00
    8bd17404d80a9b2871ba290bd0b76bc25a7fcb90

commit 8af5e89613691c74d80eaa7f7c6d0e4b46f6a2d2
Author: tracer-ics2017 <tracer@njuics.org>
Date:   Mon May 17 10:41:28 2021 +0800

    >  compile
    161920122
    tangxi
    Linux debian 4.19.0-14-686 #1 SMP Debian 4.19.171-2 (2021-01-30) i686 GNU/Linux
     10:41:28 up 45 min,  1 user,  load average: 0.00, 0.00, 0.00
    ab034e6ba1dda6410412ba95e8da7b51d5156d3

commit b0f2128099e9a3092a19abe1de0c64d58d578163
Author: tracer-ics2017 <tracer@njuics.org>
Date:   Mon May 17 10:35:13 2021 +0800

    >  run
    161920122
    tangxi
    Linux debian 4.19.0-14-686 #1 SMP Debian 4.19.171-2 (2021-01-30) i686 GNU/Linux
     10:35:13 up 39 min,  1 user,  load average: 0.00, 0.00, 0.00
    27eb12a46661c946c0ba1787757108a5b75838c

commit 28e6e93e9568dcdc7586a257078c4c1e1448c0c5
Author: tracer-ics2017 <tracer@njuics.org>
Date:   Mon May 17 10:35:13 2021 +0800

    >  compile
    161920122
    tangxi
    Linux debian 4.19.0-14-686 #1 SMP Debian 4.19.171-2 (2021-01-30) i686 GNU/Linux
     10:35:13 up 39 min,  1 user,  load average: 0.00, 0.00, 0.00
    ae1418bd59a740e14be368c378b91f5b422d9c9f

commit 359b4f3aa173748ff78817bffc4fbd7bc250f3ed
Author: tracer-ics2017 <tracer@njuics.org>
Date:   Mon May 17 10:16:41 2021 +0800

    >  run
    161920122
    tangxi
```

```
tangxi@debian:~/ics2021/nexus-am/tests/cputest$ cd ../../..
tangxi@debian:~/ics2021$ git branch
  master
  pa0
  pa1
* pa2
tangxi@debian:~/ics2021$ git push myrepo pa2
Username for 'https://e.coding.net': 15913121302
Password for 'https://15913121302@e.coding.net':
Enumerating objects: 223, done.
Counting objects: 100% (223/223), done.
Compressing objects: 100% (201/201), done.
Writing objects: 100% (201/201), 19.43 KiB | 621.00 KiB/s, done.
Total 201 (delta 180), reused 0 (delta 0)
remote: Resolving deltas: 100% (180/180), completed with 15 local objects.
To https://e.coding.net/tangxi1/tangxi/PA.git
 * [new branch]      pa2 -> pa2
tangxi@debian:~/ics2021$
```

操作题

PA2.1 第一个程序

任务1：实现标志寄存器

实现标志寄存器 `eflags` 并设置初值

进入 `nemu/include/cpu/reg.h` 文件中，在 CPU_state 中增加 eflags 寄存器,根据以下思路：

```
31                  23                  15               7             0
 +------------------+------------------+-------+-+-+-+-+-+-+-------+-+-+
 |                                             |O| |I| |S|Z|       | |C|
 |                      X                      | |X| |X| | |   X   |1| |
 |                                             |F| |F| |F|F|       | |F|
 +------------------+------------------+-------+-+-+-+-+-+-+-------+-+-+
```

查询 `i386` 手册，发现 `Chapter 10 Initialization` 与初始化相关。根据手册需要把 `eflags` 初始化为 `0x2`

在 `ics2020\nemu\src\monitor\monitor.c` 找到 `restart()` 函数，并在其中添加 `cpu.eflags.value=0x2;`

```
static inline void restart() {
   /* Set the initial instruction pointer. */
   cpu.eip = ENTRY_START;
   cpu.eflags.value=0x2;
 #ifdef DIFF_TEST
   init_qemu_reg();
 #endif
 }
```

reg.h中结构体定义如下:

```
  union {
       uint32_t val;//用来赋初值
       struct {
           uint32_t CF:1;
           uint32_t one:1;
           uint32_t :4;//无名位域
           uint32_t ZF:1;
           uint32_t SF:1;
           uint32_t :1;
           uint32_t IF:1;
           uint32_t :1;
           uint32_t OF:1;
           uint32_t :20;
  } CPU_state;
```

实现所有指令对标志位的设置

根据手册`sub`和`xor`需要修改标志位

进入`nemu/include/cpu/rtl.h`中实现相应的函数

`sub`与add类似，只是不需要减去CF。`xor`需要更新SF和ZF，同时OF=0、CF=0

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

任务2： 实现所有RTL指令

进入`nemu/include/cpu/rtl.h`中实现相应的函数

1. `rtl_mv`指令

按照注释赋值

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

2. `rtl_not`指令

按照注释取非

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

3. `rtl_sext`指令

函数参数中有`width`，对不同大小的数进行符号扩展。通过带符号整数的算数右移来实现，先左移相应的位数使得符号处于最高位，再右移回原位置，即可实现符号扩展。

```
static inline void rtl_sext(rtlreg_t* dest, const rtlreg_t* src1, int width) {
  // dest <- signext(src1[(width * 8 - 1) .. 0])
  int32_t t = *src1;//转为带符号整数
  if (width == 1) {//字节长度判断
    *dest = (t<<24)>>24;//符号扩展3位
  }
  else if (width == 2) {
    *dest = (t<<16)>>16//符号扩展2位
  }
  else if (width ==4) {
```

```
    *dest = t//不需要符号扩展
  }
  else {
    assert(0);
  }
}
```

4. `rtl_push`指令

根据注释，使用`rtl`指令实现从寄存器中取值、减法改变指向、存储内容。

```
static inline void rtl_push(const rtlreg_t* src1) {
  // esp <- esp - 4
  // M[esp] <- src1
  cpu.esp -= 4;
  rtl_sm(&cpu.esp, 4, src1);
}
```

5. `rtl_pop`指令

实现思路与`rtl_push`基本相同

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

6. `rtl_eq0`指令

根据注释，判断`src1`是否为0，若是，则给`dest`赋值1，否则赋值0

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

7. `rtl_eqi`指令

根据注释，判断操作数`src1`与立即数`imm`是否相等。若相等，则给`dest`赋值1，否则赋值0

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

8. `rtl_neq0`指令

根据注释，判断操作数是否为0

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

9. `rtl_msb` 指令

获取最高有效位，跟`rtl_sext`指令类似，可以通过逻辑右移获取其最高有效位

```
static inline void rtl_msb(rtlreg_t* dest, const rtlreg_t* src1, int width) {
  // dest <- src1[width * 8 - 1]
  rtl_shri(dest, src1, width*8 - 1);//逻辑右移
}
```

10. `make_rtl_setget_eflags` 指令

此宏定义实现了`rtl_set_XF()`与`rtl_get_XF()`的功能，我们只需要实现对`eflags`寄存器相应位置的存取即可

11. `rtl_update_ZF` 指令

更新`ZF`，根据注释，只需判断从`width*8-1`...`0`这些位是否全为0。通过逻辑左移把原来`width*8-1`位移到最高位，末尾补0。若移位后的数值为0，则`ZF=1`否则`ZF=0`

```
static inline void rtl_update_ZF(const rtlreg_t* result, int width) {
  // eflags.ZF <- is_zero(result[width * 8 - 1 .. 0])
  t0 = *result;
  rtl_shli(&t0,&t0,32-width*8);//逻辑左移，去掉无用位
  if (t0 == 0)//若整体为0，即所有位都是0
    t1=1;
  else
    t1=0;
  rtl_set_ZF(&t1);//设置ZF
}
```

12. `rtl_update_SF` 指令

更新`SF`，获取result符号即可。

```
static inline void rtl_update_SF(const rtlreg_t* result, int width) {
  // eflags.SF <- is_sign(result[width * 8 - 1 .. 0])
  rtl_msb(&t0, result, width);//获取result符号
  rtl_set_SF(&t0);//设置SF
}
```

任务3：实现 6 条 x86 指令

1`call`

运行`dummy`发现`e8`指令没有实现，查询`i386`手册可知需要实现`call`。查阅i386得 E8 cd CALL rel32 7+m Call near, displacement relative to next instruction call指令在 One-Byte Opcode Map 对应e8，填写 opcode_table 中的第e8项，据此，我们可以得到译码函数decode_I

填写`opcode_table`中的第e8项

```
/* 0xe8 */    IDEXW(I,call,0), EMPTY, EMPTY, EMPTY,
```

实现执行函数`make_EHelper(call)`，由于译码函数已经完成了跳转位置的设定，这里我们只需标志跳转并入栈eip即可进入`nemu/src/cpu/exec/control.c`文件中，实现`make_EHelper(call)`函数，并在`all-nstr.h`中声明

```
make_EHelper(call) {
  decoding.is_jmp=1;//标识跳转
  rtl_push(eip);//入栈eip
  rtl_add(&decoding.jmp_eip, eip, &id_dest->val);
  print_asm("call %x", decoding.jmp_eip);
}
```

执行结果如下

```
(nemu) si 5
  100000:   bd 00 00 00 00                      movl $0x0,%ebp
  100005:   bc 00 7c 00 00                      movl $0x7c00,%esp
  10000a:   e8 0f 00 00 00                      call 10001e
invalid opcode(eip = 0x0010001e): 55 89 e5 83 ec 18 e8 e6 ...
```

2. `push`

运行`dummy`发现`55`指令没有实现，查询`i386`手册可知需要实现`push`。

选取译码函数`decode_r`，其功能是读取操作码中的寄存器信息。

根据手册填写`opcode_table`中的第50、54项

选取执行函数`make_EHelper(push)`

```
/* 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),
```

进入`nemu/src/cpu/exec/data-mov.c`文件中编写`make_EHelper(push)`函数使用,`rtl_push`指令把取到的操作数入栈

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

执行结果如下

```
(nemu) si 10
  100000:   bd 00 00 00 00                      movl $0x0,%ebp
  100005:   bc 00 7c 00 00                      movl $0x7c00,%esp
  10000a:   e8 0f 00 00 00                      call 10001e
  10001e:   55                                  pushl %ebp
  10001f:   89 e5                               movl %esp,%ebp
invalid opcode(eip = 0x00100021): 83 ec 18 e8 e6 ff ff ff ...
```

3. `pop`

运行`dummy`发现`5d`指令没有实现，查询`i386`手册可知需要实现`pop`。

查阅i386得`58 + rd POP r32 4 Pop top of stack into dword register`

选取译码函数`decode_r`，其功能是读取操作码中的寄存器信息。

填写`opcode_table`

```
/* 0x58 */ IDEX(r, pop), IDEX(r, pop), IDEX(r, pop), IDEX(r, pop),
/* 0x5c */ IDEX(r, pop), IDEX(r, pop), IDEX(r, pop), IDEX(r, pop),
```

选取执行函数`make_EHelper(pop)`

进入`nemu/src/cpu/exec/data-mov.c`文件中,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);
}
```

执行结果如下

```
100013:    5d                                       popl %ebp
```

4.`sub`

运行`dummy`发现`83`指令没有实现，查询反汇编结果可知需要实现`sub`。

发现`opcode_table`中要使用了`grp1`，查表可知grp1[5]应填写执行函数`make_EHelper(sub)`

查阅i386得`83 /5 ib SUB r/m16,imm8 2/7 Subtract sign-extended immediate byte from r/m word`

`83 /5 ib SUB r/m32,imm8 2/7 Subtract sign-extended immediate byte from r/m dword`

`opcode_table 中0x83`已填写好，对应 gp1

```
/* 0x80 */ IDEXW(I2E, gp1, 1), IDEX(I2E, gp1), EMPTY, IDEX(SI2E, gp1),
```

sub命令对应101，在make_group中第五个位置填写

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

译码函数确定为`decode_SI2E`，进入`nemu/src/cpu/exec/arith.c`文件中编写`make_EHelper(sub)`函数，并在`all-instr.h`中声明`make_EHelper(sub)`

```
make_EHelper(sub) {
  rtl_sub(&t2, &id_dest->val, &id_src->val);
  rtl_sltu(&t3, &id_dest->val, &t2);//t3记录是否借位，0表示借位

  operand_write(id_dest, &t2);//最终结果写入对应寄存器或内存

  rtl_update_ZFSF(&t2, id_dest->width);//更新ZF,SF

  rtl_sltu(&t0, &id_dest->val, &t2);//与减去借位后再比
  rtl_or(&t0, &t3, &t0);
  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);
}
```

执行结果如下

```
100021:   83 ec 18                                  subl $0x18,%esp
```

5. `xor`

运行`dummy`发现`31`指令没有实现，查询`i386`手册可知需要实现`xor`。

查阅i386得`31 /r XOR r/m16,r16 2/6 Exclusive-OR word register to r/m word`

`31 /r XOR r/m32,r32 2/6 Exclusive-OR dword register to r/m dword`

选取译码函数`decode_G2E`,进入`nemu/src/cpu/exec/logic.c`文件中编写`make_EHelper(xor)`函数，并在`all-instr.h`中声明`make_EHelper(xor)`

选取执行函数`make_EHelper(xor)`，只需要把两个源操作数异或后的结果存入目的操作数即可。

```
/* 0x30 */ EMPTY, IDEX(E2G, xor), EMPTY, EMPTY,
```

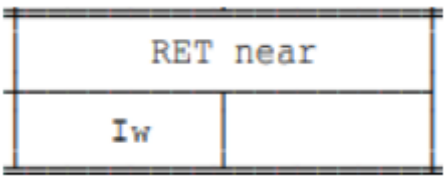
```
make_EHelper(xor) {
   rtl_xor(&t0, &id_dest->val, &id_src->val);//异或
   operand_write(id_dest,&t0);

   rtl_update_ZFSF(&t0,id_dest->width);//更新CF、ZF

   rtl_set_OF(&tzero);//CF==0
   rtl_set_CF(&tzero);//ZF==0

   print_asm_template2(xor);
}
```

执行结果如任务4

6. `ret`

运行`dummy`发现`c3`指令没有实现，查询`i386`手册可知需要实现`ret`。

查阅i386手册得，`C3 RET 10+m Return (near) to caller`



由上图可知`ret`无译码函数,进入`nemu/src/cpu/exec/control.c`文件中编写`make_EHelper(ret)`函数，并在`all-instr.h`中声明`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，并出把要跳转的位置出栈赋给`decoding.jmp_eip`

```
/* 0xc0 */ IDEXW(gp2_Ib2E, gp2, 1), IDEX(gp2_Ib2E, gp2), EMPTY, EX(ret),
```

```
make_EHelper(ret) {
  decoding.is_jmp=1;//设置跳转
  rtl_pop(&decoding.jmp_eip);//获取跳转位置
  print_asm("ret");
}
```

执行结果如任务4

任务4：成功运行dummy

```
[src/monitor/monitor.c,65,load_img] The image is /home/tangxi/ics2021/nexus-am/
u.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 23:14:45, May 16 2021
For help, type "help"
(nemu) si 30
  100000:   bd 00 00 00 00                      movl $0x0,%ebp
  100005:   bc 00 7c 00 00                      movl $0x7c00,%esp
  10000a:   e8 0f 00 00 00                      call 10001e
  10001e:   55                                  pushl %ebp
  10001f:   89 e5                               movl %esp,%ebp
  100021:   83 ec 18                            subl $0x18,%esp
  100024:   e8 e6 ff ff ff                      call 10000f
  10000f:   55                                  pushl %ebp
  100010:   89 e5                               movl %esp,%ebp
  100012:   90                                  nop
  100013:   5d                                  popl %ebp
  100014:   c3                                  ret
  100029:   e8 14 00 00 00                      call 100042
  100042:   55                                  pushl %ebp
  100043:   89 e5                               movl %esp,%ebp
  100045:   b8 00 00 00 00                      movl $0x0,%eax
  10004a:   5d                                  popl %ebp
  10004b:   c3                                  ret
  10002e:   89 45 f4                            movl %eax,-0xc(%ebp)
  100031:   83 ec 0c                            subl $0xc,%esp
  100034:   ff 75 f4                            pushl -0xc(%ebp)
  100037:   e8 d9 ff ff ff                      call 100015
  100015:   55                                  pushl %ebp
  100016:   89 e5                               movl %esp,%ebp
  100018:   8b 45 08                            movl 0x8(%ebp),%eax
nemu: HIT GOOD TRAP at eip = 0x0010001b

  10001b:   d6                                  nemu trap (eax = 0)
(nemu)
```

任务5：实现 Diff-test

在`nemu/include/common.h`中定义宏

```
#define DIFF_TEST
```

进入`nemu/src/monitor/diff-test/diff-test.c`,通用函数体如下，写入`difftest_step()`中

```
test_reg(reg)
    if (r.reg != cpu.reg) {
        diff = true;
        Log("reg error NEMU.reg=0x%08x QEMU.reg=0x%08x\n",cpu.reg,r.reg);
  }
```

如果QEMU和NEMU的寄存器结果不一样，中断程序并报错

实验心得

通过这次的实验我学会了如何查阅i386手册来选择、实现指令的译码函数和执行函数，并且在思考题中对宏有了进一步的认识。还学会了用differential testing来找bug。

其他备注

无