3.11 A. 这个指令用来将寄存器%rdx 设置为 0，运用了对任意 x，$x \hat{} x=0$ 这一属性。它对应于 C 语句 x=0。

B. 将寄存器%rdx 设置为 0 的更直接的方法是用指令 movq $0,%rdx。

C. 不过，汇编和反汇编这段代码，我们发现使用 xorq 的版本只需要 3 个字节，而使用 movq 的版本需要 7 个字节。其他将%rdx 设置为 0 的方法都依赖于这样一个属性，即任何更新低位 4 字节的指令都会把高位字节设置为 0。因此，我们可以使用 xorl % edx,% edx(2 字节)或 movl $0,% edx(5 字节)。

3.12 我们可以简单地把 cqto 指令替换为将寄存器%rdx 设置为 0 的指令，并且用 divq 而不是 idivq 作为我们的除法指令，得到下面的代码：

```
void uremdiv(unsigned long x, unsigned long y,
             unsigned long *qp, unsigned long *rp)
x in %rdi, y in %rsi, qp in %rdx, rp in %rcx
uremdiv:
  movq    %rdx, %r8       Copy qp
  movq    %rdi, %rax      Move x to lower 8 bytes of dividend
  movl    $0, %edx        Set upper 8 bytes of dividend to 0
  divq    %rsi            Divide by y
  movq    %rax, (%r8)     Store quotient at qp
  movq    %rdx, (%rcx)    Store remainder at rp
  ret
```

3.13 汇编代码不会记录程序值的类型，理解这点这很重要。相反地，不同的指令确定操作数的大小以及是有符号的还是无符号的。当从指令序列映射回 C 代码时，我们必须做一点儿侦查工作，推断程序值的数据类型。

A. 后缀‘l’和寄存器指示符表明是 32 位操作数，而比较是对补码的< 。我们可以推断 data_t 一定是 int。

B. 后缀‘w’和寄存器指示符表明是 16 位操作数，而比较是对补码的>= 。我们可以推断 data_t 一定是 short。

C. 后缀‘b’和寄存器指示符表明是 8 位操作数，而比较是对无符号数的<= 。我们可以推断 data_t 一定是 unsigned char。

D. 后缀‘q’和寄存器指示符表明是 64 位操作数，而比较是!= ，有符号、无符号和指针参数都是一样的。我们可以推断 data_t 可以是 long、unsigned long 或者某种形式的指针。

3.14 这道题与练习题 3.13 类似，不同的是它使用了 TEST 指令而不是 CMP 指令。

A. 后缀‘q’和寄存器指示符表明是 64 位操作数，而比较是>= ，一定是有符号数。我们可以推断 data_t 一定是 long。

B. 后缀‘w’和寄存器指示符表明是 16 位操作数，而比较是==，这个对有符号和无符号都是一样的。我们可以推断 data_t 一定是 short 或者 unsigned short。

C. 后缀‘b’和寄存器指示符表明是 8 位操作数，而比较是针对无符号数的> 。我们可以推断 data_t 一定是 unsigned char。

D. 后缀‘l’和寄存器指示符表明是 32 位操作数，而比较是<= 。我们可以推断 data_t 一定是 int。

3.15 这个练习要求你仔细检查反汇编代码，并推理跳转目标的编码。同时练习十六进制运算。

A. je 指令的目标为 0x4003fc+ 0x02。如原始的反汇编代码所示，这就是 0x4003fe。

```
4003fa: 74 02                   je     4003fe
4003fc: ff d0                   callq  *%rax
```

B. jb 指令的目标是 0x400431－12(由于 0xf4 是－12 的一个字节的补码表示)。正如原始的反汇编代码所示，这就是 0x400425：

```
40042f: 74 f4                   je     400425
400431: 5d                      pop    %rbp
```

C. 根据反汇编器产生的注释，跳转目标是绝对地址 0x400547。根据字节编码，一定在距离 pop