


```
arith2:
    orq    %rsi, %rdi
    sarq   $3, %rdi
    notq   %rdi
    movq   %rdx, %rax
    subq   %rdi, %rax
    ret
```

基于这些汇编代码，填写 C 语言代码中缺失的部分。

 **练习题 3.11** 常常可以看见以下形式的汇编代码行：

```
xorq %rdx,%rdx
```

但是在产生这段汇编代码的 C 代码中，并没有出现 EXCLUSIVE-OR 操作。

- 解释这条特殊的 EXCLUSIVE-OR 指令的效果，它实现了什么有用的操作。
- 更直接地表达这个操作的汇编代码是什么？
- 比较同样一个操作的两种不同实现的编码字节长度。

3.5.5 特殊的算术操作

正如我们在 2.3 节中看到的，两个 64 位有符号或无符号整数相乘得到的乘积需要 128 位来表示。x86-64 指令集对 128 位(16 字节)数的操作提供有限的支持。延续字(2 字节)、双字(4 字节)和四字(8 字节)的命名惯例，Intel 把 16 字节的数称为八字(oct word)。图 3-12 描述的是支持产生两个 64 位数字的全 128 位乘积以及整数除法的指令。

指令	效果	描述
imulq S	$R[\%rdx] \leftarrow R[\%rax] \leftarrow S \times R[\%rax]$	有符号全乘法
mulq S	$R[\%rdx] \leftarrow R[\%rax] \leftarrow S \times R[\%rax]$	无符号全乘法
cltq	$R[\%rdx] \leftarrow R[\%rax] \leftarrow \text{符号扩展}(R[\%rax])$	转换为八字
idivq S	$R[\%rdx] \leftarrow R[\%rdx] : R[\%rax] \bmod S$ $R[\%rdx] \leftarrow R[\%rdx] : R[\%rax] \div S$	有符号除法
divq S	$R[\%rdx] \leftarrow R[\%rdx] : R[\%rax] \bmod S$ $R[\%rdx] \leftarrow R[\%rdx] : R[\%rax] \div S$	无符号除法

图 3-12 特殊的算术操作。这些操作提供了有符号和无符号数的全 128 位乘法和除法。

一对寄存器 %rdx 和 %rax 组成一个 128 位的八字

imulq 指令有两种不同的形式。其中一种，如图 3-10 所示，是 IMUL 指令类中的一种。这种形式的 imulq 指令是一个“双操作数”乘法指令。它从两个 64 位操作数产生一个 64 位乘积，实现了 2.3.4 和 2.3.5 节中描述的操作 \times_{64}^u 和 \times_{64}^s 。(回想一下，当将乘积截取到 64 位时，无符号乘和补码乘的位级行为是一样的。)

此外，x86-64 指令集还提供了两条不同的“单操作数”乘法指令，以计算两个 64 位值的全 128 位乘积——一个是无符号数乘法(mulq)，而另一个是补码乘法(imulq)。这两条指令都要求一个参数必须在寄存器 %rax 中，而另一个作为指令的源操作数给出。然后乘积存放在寄存器 %rdx(高 64 位)和 %rax(低 64 位)中。虽然 imulq 这个名字可以用于两个不同的乘法操作，但是汇编器能够通过计算操作数的数目，分辨出想用哪条指令。

下面这段 C 代码是一个示例，说明了如何从两个无符号 64 位数字 x 和 y 生成 128 位的乘积：