```
#include <inttypes.h>

typedef unsigned __int128 uint128_t;

void store_uprod(uint128_t *dest, uint64_t x, uint64_t y) {
    *dest = x * (uint128_t) y;
}
```

在这个程序中，我们显式地把 x 和 y 声明为 64 位的数字，使用文件 inttypes.h 中声明的定义，这是对标准 C 扩展的一部分。不幸的是，这个标准没有提供 128 位的值。所以我们只好依赖 GCC 提供的 128 位整数支持，用名字__int128 来声明。代码用 typedef 声明定义了一个数据类型 uint128_t，沿用的 inttypes.h 中其他数据类型的命名规律。这段代码指明得到的乘积应该存放在指针 dest 指向的 16 字节处。

GCC 生成的汇编代码如下：

```
    void store_uprod(uint128_t *dest, uint64_t x, uint64_t y)
    dest in %rdi, x in %rsi, y in %rdx
1   store_uprod:
2     movq    %rsi, %rax      Copy x to multiplicand
3     mulq    %rdx            Multiply by y
4     movq    %rax, (%rdi)    Store lower 8 bytes at dest
5     movq    %rdx, 8(%rdi)   Store upper 8 bytes at dest+8
6     ret
```

可以观察到，存储乘积需要两个 movq 指令：一个存储低 8 个字节(第 4 行)，一个存储高 8 个字节(第 5 行)。由于生成这段代码针对的是小端法机器，所以高位字节存储在大地址，正如地址 8(%rdi)表明的那样。

前面的算术运算表(图 3-10)没有列出除法或取模操作。这些操作是由单操作数除法指令来提供的，类似于单操作数乘法指令。有符号除法指令 idivl 将寄存器%rdx(高 64 位)和%rax(低 64 位)中的 128 位数作为被除数，而除数作为指令的操作数给出。指令将商存储在寄存器%rax 中，将余数存储在寄存器%rdx 中。

对于大多数 64 位除法应用来说，除数也常常是一个 64 位的值。这个值应该存放在%rax 中，%rdx 的位应该设置为全 0(无符号运算)或者%rax 的符号位(有符号运算)。后面这个操作可以用指令 cqto[⊖]来完成。这条指令不需要操作数——它隐含读出%rax 的符号位，并将它复制到%rdx 的所有位。

我们用下面这个 C 函数来说明 x86-64 如何实现除法，它计算了两个 64 位有符号数的商和余数：

```
void remdiv(long x, long y,
            long *qp, long *rp) {
    long q = x/y;
    long r = x%y;
    *qp = q;
    *rp = r;
}
```

该函数编译得到如下汇编代码：

⊖ 在 Intel 的文档中，这条指令叫做 cqo，这是指令的 ATT 格式名字和 Intel 名字无关的少数情况之一。