Lab1: 大整数运算

小实验说明

小实验 (Labs) 是 ICS 这门课程里的一些综合编程题，旨在结合课堂知识解决一些实际中的问题。因为问题来自实际，所以有时候未必能立即在课本上找到相关知识的答案，而是需要“活学活用”。因此，大家需要利用互联网上的知识解决这些问题，但**不要**试图直接搜索这些问题的答案，即便有也不要点进去 (也请自觉不要公开发布答案)。

Deadline: 2024年10月27日23:59:59

1. 背景

我们在《计算机系统基础》课程中，学习了数据是如何用机器表示的，以及计算机支持的数据的运算。在这个实验中，我们将会灵活地使用这些计算机系统的运算，实现一个非常基础的数学运算，并尝试着做一些性能统计。最后，我们有一个选做的实验帮助大家理解 IEEE754 浮点数的表示和运算。

2. 实验要求

2.1 实验内容

给定 64 位无符号整数 a, b, m (类型为 `uint64_t`，即 $1 \le a, b, m < 2^{64}$)。你的任务是求出 $a \times b \pmod m$ 的数值，即最小的非负整数 t，满足 $a \times b \equiv t \pmod m$。

在这个实验中，你的任务是根据你所掌握的知识，实现功能正确的 `multimod`，并且只使用分支、循环、局部变量 (使用整数的位宽至多为 64-bit，即不得作弊使用 128 位整数)、赋值语句、位运算和加减法。允许定义额外的辅助函数。

我们允许使用数组，并允许你在全局空间初始化不超过 4 KB 的常量数组。`multimod` 函数调用时使用栈上的内存不得超过 4 KB。对挑战自己有兴趣的同学可以尝试不使用数组的实现。

你的代码应当是可移植的：应当同时兼容 32-bit 和 64-bit 系统。我们的框架代码 (和 Online Judge) 会用不同的选项编译 `multimod-32` 和 `multimod-64` 两个二进制文件。

2.2 代码获取与提交

如果你首次开始《计算机系统基础》Labs 系列实验，请从 Github 获取 ics-workbench:

```
git clone https://github.com/NJU-ProjectN/ics-workbench.git
```

进入 `ics-workbench` 后，在终端中执行

```
git pull origin lab1
```

可以获得 `multimod` 的框架代码。请大家保持在 master 分支完成实验。在 Lab 对应的目录中 `make submit`。如 Lab1 的工作目录为 `multimod/`，则在 `multimod` 目录中执行 `make submit`。和 PA 类似，你需要设置 `TOKEN`、`STUID` (学号) 和 `STUNAME` (中文姓名) 环境变量。

2.3 评分与正确性标准

在你提交的代码中，`multimod.c` 应该只包含 $a \cdot b \bmod m$ 的实现。其他代码 (如测试代码) 请存放在其他文件中，以免因包含禁止的操作被 Online Judge 拒绝。我们会将你的 `multimod.c` 复制到指定位置。

我们对禁用的操作 (如乘法和除法) 有一定的检查机制，但不能 100% 确保没有遗漏。请大家自觉遵守题目要求。

Labs 完全客观评分；评分方法请阅读实验须知。我们的测试用例分为 Easy 和 Hard。对于所有 Easy 测试用例，我们保证 $a, b < 2^{31}$。对于一个额外的 Easy 测试用例，我们保证 m 是 2 的整数次幂——这将会极大简化你的实现。**如果你感到实验很困难 (通常说明你的编程基础不足)，在 Deadline 之前通过 Easy 测试用例能获得本次实验的绝大部分分数。**

2.4 常见问题

1. 请不要修改 “ics-workbench” 的名字。否则将导致 Online Judge 不能识别你的目录。
2. 如果你的代码中包含乘法或除法指令，Online Judge 将会拒绝你的提交。如果你收到存在 “mul/div” 指令的反馈，你可以使用 binutils 工具集中的 objdump 命令查看编译后文件的二进制代码 (这个命令在整个计算机系统系列的实验中都非常常用)。另一种方法是使用 gcc 的 `-S` 选项生成汇编代码。

3. 实验指南

3.1 一个错误的实现

你很容易写出如下实现：

```
uint64_t multimod(uint64_t a, uint64_t b, uint64_t m) {
  return (a * b) % m;
}
```

但这段看起来正常的代码并不能满足实验的要求：使用乘法 (`a * b`) 计算两个 64 位整数的乘积，只能保留乘积的低 64 位。这意味着我们计算出的是

$$(a \cdot b \bmod 2^{64}) \bmod m;$$

你很容易能举出一个反例，使得

$$(a \cdot b \bmod 2^{64}) \bmod m \neq a \cdot b \bmod m.$$

3.2 测试建议

我们建议大家在开始写满足要求的 lab 代码之前，首先做一个正确且没有任何限制的参考实现——无论是直接用 C/C++ 实现，还是“作弊”使用 Python——Python 会容易得多，因为它自带大整数，然后使用 `popen` 函数可以立即在 C 语言里得到一个保证正确的大整数运算实现。

```
FILE *fp = popen("python3 -c 'print(10**100 - 1)'", "r");
assert(fp);
fscanf(fp, "%s", buf);
printf("popen() returns: %s\n", buf);
pclose(fp);
```

在有这样一个参考实现之后，你就可以运行大量的测试来确保你的实现没有问题——除了测试一些随机的数据外，你也可以试着测试一些在整数大小边界的数据。此外，你的代码中可能会用到一些辅助函数 (例如乘法等)，你可以把它们的测试像课堂上讲解的 yemu 代码一样做成框架，这能大幅提高你的测试/调试效率。

3.3 如何实现不用乘除法的 `multimod`?

首先，你会有一个非常坚定的信念：这件事是办得到的。我们已经在《数字逻辑电路》以及《计算机系统基础》课程和习题课中反复强调，我们今天的计算机也不过是用逻辑门电路实现的。因此位运算和分支/循环 (甚至我们还允许使用加减法) 绝对有足够的表达能力实现**任何算法**。因此，大不了我们用逻辑门 (对应着位运算) 搭一个乘法器、除法器，不就行了吗?

3.3 使用位运算实现加法

虽然我们的实验允许使用加减法，但你完全可以使用位运算构造加法电路 (直接使用大家学过的数字逻辑电路中的加法器)，例如假设 `a` 和 `b` 是两位二进制数，存储在 `int` 型变量中：

```
static inline int bit_of(int x, int i) {
  return (x >> i) & 1;
}
static inline int full_add(int x, int y, int z) {
  return x ^ y ^ z;
}
static inline int carry(int x, int y, int z) {
  return (x & y) | (x & z) | (y & z);
}

int add(int a, int b) {
  int s0 = full_add(bit_of(a, 0), bit_of(b, 0), 0);
  int c0 = carry   (bit_of(a, 0), bit_of(b, 0), 0);
  int s1 = full_add(bit_of(a, 1), bit_of(b, 1), c0);
  int c1 = carry   (bit_of(a, 1), bit_of(b, 1), c0);
  return (c1 << 2) | (s1 << 1) | (s0 << 0);
}
```

虽然代码看起来有点笨 (并且能写得更好)，但原理就是如此。顺着这个思路，你可以用加法器实现乘法器/除法器 (当然，除法其实困难得多……)

3.4 实现 `multimod`

盯着 “$a \cdot b \bmod m$” 看是没办法解决问题的。正确的解题方法是把式子写出来，然后尝试做一些公式变形。这个例子里的公式变形是很直观的：

$$a \cdot b = (a_0 \cdot 2^0 + a_1 \cdot 2^1 + \ldots + a_{63} \cdot 2^{63}) \cdot b$$

对于上面表达式括号中的每一项，都是形如 $2^i \cdot b$ 的形式 (因为 $a_i \in \{0, 1\}$)——因此

$$a \cdot b \bmod m = \left(\sum_{0 \le i < 64} a_i \cdot b \cdot 2^i \bmod m \right) \bmod m$$

因此，你只要能实现 $\bmod m$ 的加法即 $(x + y) \bmod m$，就能实现 $(b \cdot 2^i) \bmod m$，进而实现 $a \cdot b \bmod m$。在这里你要小心 $x + y$ 溢出 64-bit 整数的问题：当 $x + y = t + 2^{64}$ 发生溢出 $(0 \le t < 2^{64} - 1)$ 时，注意加法 wraparound 后得到的结果是

$$(x+y) \bmod 2^{64} = t.$$

我们实际需要求解的是

$$(t+2^{64}) \bmod m = \big((t \bmod m) + (2^{64} \bmod m)\big) \bmod m.$$

这里还有一个潜在的溢出问题——如果这个加法依然溢出怎么办？这个聪明的问题留给你。

4. 补充：一段神秘代码

求 $a \cdot b \bmod m$ 是程序设计竞赛中的一个常见操作。在 ICPC 圈子中存在一份广为流传的一段神奇代码，出处和年代太久远已经很难考证 (我们保留了这份代码本来的样子)：

```
int64_t multimod_fast(int64_t a, int64_t b, int64_t m) {
  int64_t t = (a * b - (int64_t)((double)a * b / m) * m) % m;
  return t < 0 ? t + m : t;
}
```

这段代码直接使用了 `a * b`，并且直接假设整数溢出时 wraparound (可以通过 `-fwrapv` 编译选项实现)。然后，这段代码竟然神奇地在 $O(1)$ 的时间里就解决了 $a \cdot b \bmod m$ 的求解？其中最精髓的一段在于：

```
(int64_t)((double)a * b / m)
```

把整数 `a` 强行转换成了浮点数 (假设使用 IEEE754 64 位双精度类型)。我们知道浮点数的精度也不是无限的，仅仅类型转换就已经损失精度了 (64 位双精度浮点数只有 52 位有效数字)，`double` 更不可能准确保存 `a * b` 的结果。

如果你对这一段代码有兴趣，请你利用学习过的数据的机器表示知识，理解 `multimod_fast` 中每一个子表达式的含义，并且分析其正确性 (完全处于自愿目的，不计分，也不需要提交任何代码)。