我们可以看到，计算 13! 溢出了。正如在练习题 2.35 中学到的那样，还可以通过计算 x/n，看它是否等于$(n-1)!$ 来测试 $n!$ 的计算是否溢出了(假设我们已经能够保证$(n-1)!$ 的计算没有溢出)。在此处，我们得到 1 932 053 504/13＝161 004 458.667。另外有个测试方法，可以看到 10! 以上的阶乘数都必须是 100 的倍数，因此最后两位数字必然是 0。13! 的正确值应该是 6 227 020 800。

B. 用数据类型 long 来计算，直到 20! 才溢出，得到 2 432 902 008 176 640 000。

3.23 编译循环产生的代码可能会很难分析，因为编译器对循环代码可以执行许多不同的优化，也因为可能很难把程序变量和寄存器匹配起来。这个特殊的例子展示了几个汇编代码不仅仅是 C 代码直接翻译的地方。

A. 虽然参数 x 通过寄存器%rdi 传递给函数，可以看到一旦进入循环就再也没有引用过该寄存器了。相反，我们看到第 2～5 行上寄存器%rax、%rcx 和%rdx 分别被初始化为 x、x*x 和 x+x。因此可以推断，这些寄存器包含着程序变量。

B. 编译器认为指针 p 总是指向 x，因此表达式(*p)++就能够实现 x 加一。代码通过第 7 行的 leaq 指令，把这个加一和加 y 组合起来。

C. 添加了注释的代码如下：

```
    long dw_loop(long x)
    x initially in %rdi
1   dw_loop:
2     movq    %rdi, %rax          Copy x to %rax
3     movq    %rdi, %rcx
4     imulq   %rdi, %rcx          Compute y = x*x
5     leaq    (%rdi,%rdi), %rdx   Compute n = 2*x
6   .L2:                        loop:
7     leaq    1(%rcx,%rax), %rax  Compute x += y + 1
8     subq    $1, %rdx            Decrement n
9     testq   %rdx, %rdx          Test n
10    jg      .L2                 If > 0, goto loop
11    rep; ret                    Return
```

3.24 这个汇编代码是用跳转到中间方法对循环的相当直接的翻译。完整的 C 代码如下：

```
long loop_while(long a, long b)
{
    long result = 1;
    while (a < b) {
        result = result * (a+b);
        a = a+1;
    }
    return result;
}
```

3.25 这个汇编代码没有完全遵循 guarded-do 翻译的模式，可以看到它等价于下面的 C 代码：

```
long loop_while2(long a, long b)
{
    long result = b;
    while (b > 0) {
        result = result * a;
        b = b-a;
    }
    return result;
}
```

我们会经常看到这样的情况，特别是用较高优化等级编译时，此时 GCC 会自作主张地修改生成代码的格式，同时又保留所要求的功能。

3.26 能够从汇编代码工作回 C 代码，是逆向工程的一个主要例子。