

 **练习题 4.4** 根据下面的 C 代码，用 Y86-64 代码来实现一个递归求和函数 rsum：

```
long rsum(long *start, long count)
{
    if (count <= 0)
        return 0;
    return *start + rsum(start+1, count-1);
}
```

使用与 x86-64 代码相同的参数传递和寄存器保存方法。在一台 x86-64 机器上编译这段 C 代码，然后再把那些指令翻译成 Y86-64 的指令，这样做可能会很有帮助。

 **练习题 4.5** 修改 sum 函数的 Y86-64 代码(图 4-6)，实现函数 absSum，它计算一个数组的绝对值的和。在内循环中使用条件跳转指令。


 **练习题 4.6** 修改 sum 函数的 Y86-64 代码(图 4-6)，实现函数 absSum，它计算一个数组的绝对值的和。在内循环中使用条件传送指令。

4.1.6 一些 Y86-64 指令的详情

大多数 Y86-64 指令是以一种直接明了的方式修改程序状态的，所以定义每条指令想要达到的结果并不困难。不过，两个特别的指令的组合需要特别注意一下。

pushq 指令会把栈指针减 8，并且将一个寄存器值写入内存中。因此，当执行 pushq %rsp 指令时，处理器的行为是不确定的，因为要入栈的寄存器会被同一条指令修改。通常有两种不同的约定：1) 压入 %rsp 的原始值，2) 压入减去 8 的 %rsp 的值。


对于 Y86-64 处理器来说，我们采用和 x86-64 一样的做法，就像下面这个练习题确定出的那样。

 **练习题 4.7** 确定 x86-64 处理器上指令 pushq %rsp 的行为。我们可以通过阅读 Intel 关于这条指令的文档来了解它们的做法，但更简单的方法是在实际的机器上做个实验。C 编译器正常情况下是不会产生这条指令的，所以我们必须用手工生成的汇编代码来完成这一任务。下面是我们写的一个测试程序(网络旁注 ASM:EASM，描绘如何编写 C 代码和手写汇编代码结合的程序)：

```
1  .text
2  .globl pushtest
3  pushtest:
4      movq    %rsp, %rax    Copy stack pointer
5      pushq   %rsp         Push stack pointer
6      popq    %rdx         Pop it back
7      subq    %rdx, %rax    Return 0 or 8
8      ret
```

在实验中，我们发现函数 pushtest 总是返回 0，这表示在 x86-64 中 pushq %rsp 指令的行为是怎样的呢？

对 popq %rsp 指令也有类似的歧义。可以将 %rsp 置为从内存中读出的值，也可以置为加了增量后的栈指针。同练习题 4.7 一样，让我们做个实验来确定 x86-64 机器是怎么处理这条指令的，然后 Y86-64 机器就采用同样的方法。

 **练习题 4.8** 下面这个汇编函数让我们确定 x86-64 上指令 popq %rsp 的行为：

```
1  .text
2  .globl poptest
3  poptest:
```