GCC 为 find_range 生成图 3-51b 中的代码。这段代码的效率不是很高：它比较了 x 和 0.0 三次，即使一次比较就能获得所需的信息。它还生成了浮点常数两次：一次使用 vxorps，另一次从内存读出这个值。让我们追踪这个函数，看看四种可能的比较结果：

x < 0.0　第 4 行的 ja 分支指令会选择跳转，跳转到结尾，返回值为 0。

x=0.0　ja(第 4 行)和 jp(第 6 行)两个分支语句都会选择不跳转，但是 je 分支(第 8 行)会选择跳转，以% eax 等于 1 返回。

x > 0.0　这三个分支都不会选择跳转。setbe(第 11 行)会得到 0，addl 指令(第 13 行)会把它增加，得到返回值 2。

x=*NaN*　jp 分支(第 6 行)会选择跳转。第三个 vucomiss 指令(第 10 行)会设置进位和零标志位，因此 setbe 指令(第 11 行)和后面的指令会把% eax 设置为 1。addl 指令(第 13 行)会把它增加，得到返回值 3。

家庭作业 3.73 和 3.74 中，你需要试着手动生成 find_range 更高效的实现。

练习题 3.57　函数 funct3 有如下原型：

```
double funct3(int *ap, double b, long c, float *dp);
```

对于此函数，GCC 产生如下代码：

```
    double funct3(int *ap, double b, long c, float *dp)
    ap in %rdi, b in %xmm0, c in %rsi, dp in %rdx
1   funct3:
2     vmovss  (%rdx), %xmm1
3     vcvtsi2sd       (%rdi), %xmm2, %xmm2
4     vucomisd        %xmm2, %xmm0
5     jbe     .L8
6     vcvtsi2ssq      %rsi, %xmm0, %xmm0
7     vmulss  %xmm1, %xmm0, %xmm1
8     vunpcklps       %xmm1, %xmm1, %xmm1
9     vcvtps2pd       %xmm1, %xmm0
10    ret
11  .L8:
12    vaddss  %xmm1, %xmm1, %xmm1
13    vcvtsi2ssq      %rsi, %xmm0, %xmm0
14    vaddss  %xmm1, %xmm0, %xmm0
15    vunpcklps       %xmm0, %xmm0, %xmm0
16    vcvtps2pd       %xmm0, %xmm0
17    ret
```

写出 funct3 的 C 版本。

3.11.7　对浮点代码的观察结论

我们可以看到，用 AVX2 为浮点数上的操作产生的机器代码风格类似于为整数上的操作产生的代码风格。它们都使用一组寄存器来保存和操作数据值，也都使用这些寄存器来传递函数参数。

当然，处理不同的数据类型以及对包含混合数据类型的表达式求值的规则有许多复杂之处，同时，AVX2 代码包括许多比只执行整数运算的函数更加不同的指令和格式。

AVX2 还有能力在封装好的数据上执行并行操作，使计算执行得更快。编译器开发者正致力于自动化从标量代码到并行代码的转换，但是目前通过并行化获得更高性能的最可