```
    Inner loop of combine4.  data_t = double, OP = *
    acc in %xmm0, data+i in %rdx, data+length in %rax
1   .L25:                                   loop:
2     vmulsd   (%rdx), %xmm0, %xmm0          Multiply acc by data[i]
3     addq     $8, %rdx                      Increment data+i
4     cmpq     %rax, %rdx                    Compare to data+length
5     jne      .L25                          If !=, goto loop
```

如图 5-13 所示，在我们假想的处理器设计中，指令译码器会把这 4 条指令扩展成为一系列的五步操作，最开始的乘法指令被扩展成一个 load 操作，从内存读出源操作数，和一个 mul 操作，执行乘法。

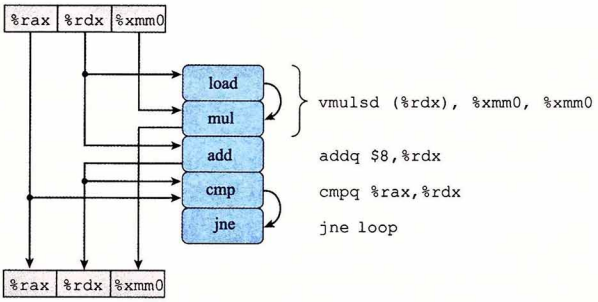

图 5-13　combine4 的内循环代码的图形化表示。指令动态地被翻译成一个或两个操作，每个操作从其他操作或寄存器接收值，并且为其他操作和寄存器产生值。我们给出最后一条指令的目标为标号 loop。它跳转到给出的第一条指令

作为生成程序数据流图表示的一步，图 5-13 左手边的方框和线给出了各个指令是如何使用和更新寄存器的，顶部的方框表示循环开始时寄存器的值，而底部的方框表示最后寄存器的值。例如，寄存器%rax 只被 cmp 操作作为源值，因此这个寄存器在循环结束时有着同循环开始时一样的值。另一方面，在循环中，寄存器%rdx 既被使用也被修改。它的初始值被 load 和 add 操作使用；它的新值由 add 操作产生，然后被 cmp 操作使用。在循环中，mul 操作首先使用寄存器%xmm0 的初始值作为源值，然后会修改它的值。

图 5-13 中的某些操作产生的值不对应于任何寄存器。在右边，用操作间的弧线来表示。load 操作从内存读出一个值，然后把它直接传递到 mul 操作。由于这两个操作是通过对一条 vmulsd 指令译码产生的，所以这个在两个操作之间传递的中间值没有与之相关联的寄存器。cmp 操作更新条件码，然后 jne 操作会测试这些条件码。

对于形成循环的代码片段，我们可以将访问到的寄存器分为四类：

只读：这些寄存器只用作源值，可以作为数据，也可以用来计算内存地址，但是在循环中它们是不会被修改的。循环 combine4 的只读寄存器是%rax。

只写：这些寄存器作为数据传送操作的目的。在本循环中没有这样的寄存器。

局部：这些寄存器在循环内部被修改和使用，迭代与迭代之间不相关。在这个循环中，条件码寄存器就是例子：cmp 操作会修改它们，然后 jne 操作会使用它们，不过这种相关是在单次迭代之内的。

循环：对于循环来说，这些寄存器既作为源值，又作为目的，一次迭代中产生的值会在另一次迭代中用到。可以看到，%rdx 和%xmm0 是 combine4 的循环寄存器，对应于程序