做任何转换的传送浮点数的指令。引用内存的指令是标量指令，意味着它们只对单个而不是一组封装好的数据值进行操作。数据要么保存在内存中(由表中的 M_{32} 和 M_{64} 指明)，要么保存在 XMM 寄存器中(在表中用 X 表示)。无论数据对齐与否，这些指令都能正确执行，不过代码优化规则建议 32 位内存数据满足 4 字节对齐，64 位数据满足 8 字节对齐。内存引用的指定方式与整数 MOV 指令的一样，包括偏移量、基址寄存器、变址寄存器和伸缩因子的所有可能的组合。

指令	源	目的	描述
vmovss	M_{32}	X	传送单精度数
vmovss	X	M_{32}	传送单精度数
vmovsd	M_{64}	X	传送双精度数
vmovsd	X	M_{64}	传送双精度数
vmovaps	X	X	传送对齐的封装好的单精度数
vmovapd	X	X	传送对齐的封装好的双精度数

图 3-46　浮点传送指令。这些操作在内存和寄存器之间以及一对寄存器之间传送值(X：XMM 寄存器(例如%xmm3)；M_{32}：32 位内存范围；M_{64}：64 位内存范围)

GCC 只用标量传送操作从内存传送数据到 XMM 寄存器或从 XMM 寄存器传送数据到内存。对于在两个 XMM 寄存器之间传送数据，GCC 会使用两种指令之一，即用 vmovaps 传送单精度数，用 vmovapd 传送双精度数。对于这些情况，程序复制整个寄存器还是只复制低位值既不会影响程序功能，也不会影响执行速度，所以使用这些指令还是针对标量数据的指令没有实质上的差别。指令名字中的字母'a'表示 "aligned(对齐的)"。当用于读写内存时，如果地址不满足 16 字节对齐，它们会导致异常。在两个寄存器之间传送数据，绝不会出现错误对齐的状况。

下面是一个不同浮点传送操作的例子，考虑以下 C 函数

```
float float_mov(float v1, float *src, float *dst) {
    float v2 = *src;
    *dst = v1;
    return v2;
}
```

与它相关联的 x86-64 汇编代码为

```
    float float_mov(float v1, float *src, float *dst)
    v1 in %xmm0, src in %rdi, dst in %rsi
1   float_mov:
2     vmovaps %xmm0, %xmm1      Copy v1
3     vmovss  (%rdi), %xmm0     Read v2 from src
4     vmovss  %xmm1, (%rsi)     Write v1 to dst
5     ret                       Return v2 in %xmm0
```

这个例子中可以看到它使用了 vmovaps 指令把数据从一个寄存器复制到另一个，使用了 vmovss 指令把数据从内存复制到 XMM 寄存器以及从 XMM 寄存器复制到内存。

图 3-47 和图 3-48 给出了在浮点数和整数数据类型之间以及不同浮点格式之间进行转换的指令集合。这些都是对单个数据值进行操作的标量指令。图 3-47 中的指令把一个从 XMM 寄存器或内存中读出的浮点值进行转换，并将结果写入一个通用寄存器(例如 %rax、%ebx 等)。把浮点值转换成整数时，指令会执行截断(truncation)，把值向 0 进行舍