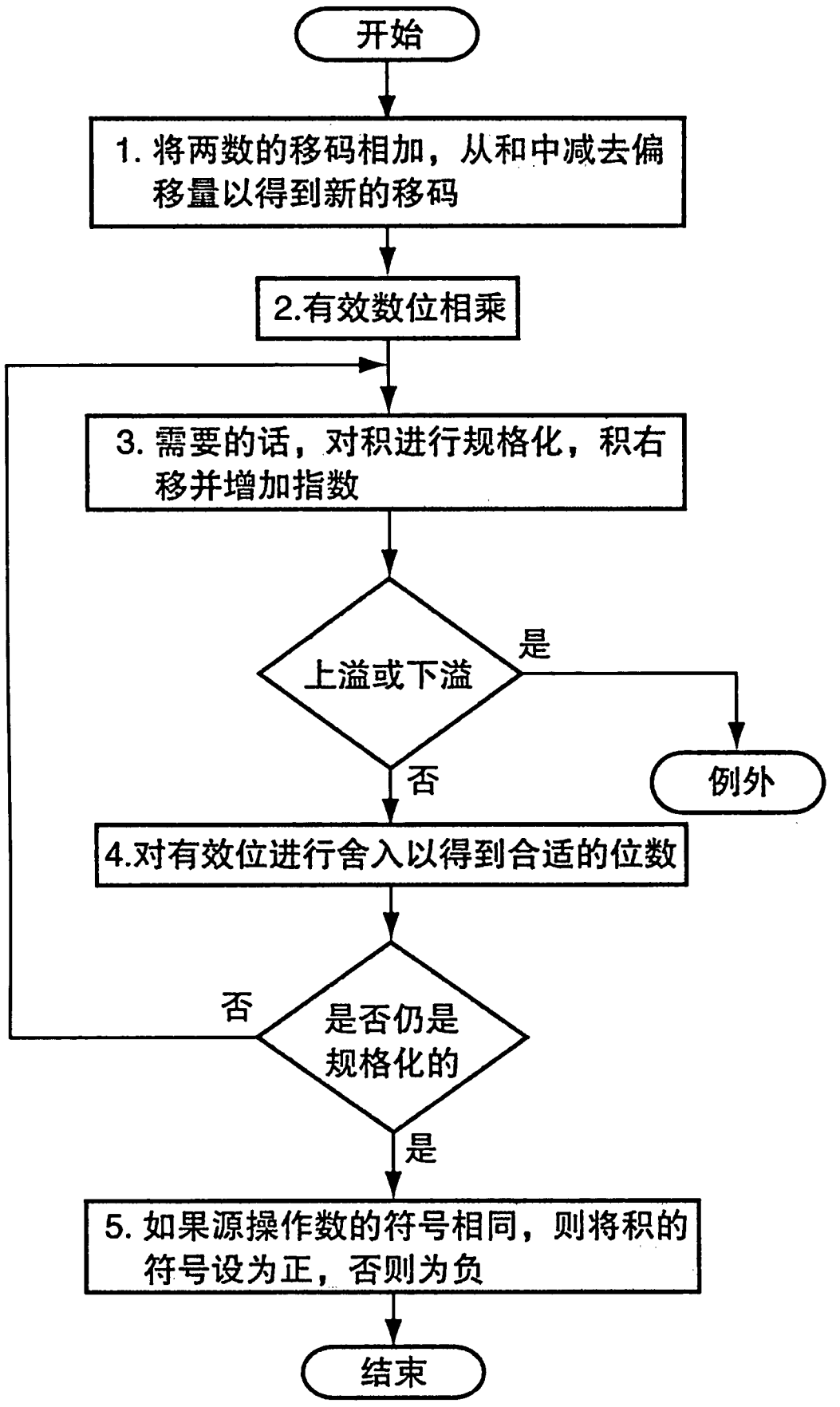

图 3-16　浮点乘法。通常是执行步骤 3 和步骤 4 各一次，但如果舍入导致结果非规格化了，必须重复执行步骤 3

例题 | 二进制浮点乘法

请按照图 3-16 的步骤，求出 0.5_{10} 和 -0.4375_{10} 的积。

答案 |

在二进制下，即是要将 $1.000_2 \times 2^{-1}$ 和 $-1.110_2 \times 2^{-2}$ 相乘。

第一步：将不带偏移值的指数相加：

$$-1+(-2)=-3$$

或使用移码表示为：

$$(-1+127)+(-2+127)-127=(-1-2)+(127+127-127)=-3+127=124$$

第二步：有效数位相乘：

$$\begin{array}{r} 1.000_2 \\ \times\ 1.110_2 \\ \hline 0000 \\ 1000 \\ 1000 \\ 1000 \\ \hline 1110000_2 \end{array}$$

乘积为 $1.110000_2 \times 2^{-3}$，但我们需要保留 4 位有效数字，因此结果为 $1.110_2 \times 2^{-3}$。

第三步：现在我们要检查积以确保它是规格化的，然后检查指数是否上溢或下溢。积已经是规格化的了，并且因为 $127 \geqslant -3 \geqslant -126$，所以没有上溢或下溢。（使用移码表示法，$254 \geqslant 124 \geqslant 1$，因此指数大小合适。）

第四步：对积舍入，结果没什么影响：

$$1.110_2 \times 2^{-3}$$

第五步：因为操作数的符号相反，因此将积的符号设为负。所以，乘积为：

$$-1.110_2 \times 2^{-3}$$

转化为十进制来检查所得结果：

$$-1.110_2 \times 2^{-3} = -0.001110_2 = -0.00111_2$$

$$= -7/2^5{}_{10} = -7/32_{10} = -0.21875_{10}$$

而 0.5_{10} 和 -0.4375_{10} 的乘积正是 -0.21875_{10}。■

3.5.6　RISC-V 中的浮点指令

RISC-V 支持 IEEE 754 标准定义的单精度和双精度格式，有以下这些指令：

- 单精度浮点加法指令（fadd.s）和双精度浮点加法指令（fadd.d）
- 单精度浮点减法指令（fsub.s）和双精度浮点减法指令（fsub.d）
- 单精度浮点乘法指令（fmul.s）和双精度浮点乘法指令（fmul.d）
- 单精度浮点除法指令（fdiv.s）和双精度浮点除法指令（fdiv.d）
- 单精度浮点平方根指令（fsqrt.s）和双精度浮点平方根指令（fsqrt.d）
- 单精度浮点相等指令（feq.s）和双精度浮点相等指令（feq.d）
- 单精度浮点小于指令（flt.s）和双精度浮点小于指令（flt.d）
- 单精度浮点小于或等于指令（fle.s）和双精度浮点小于或等于指令（fle.d）

如果比较结果为假，比较指令 feq、flt 和 fle 将整点寄存器设为 0，如果是真的，则设为 1。因此软件可以使用整数分支指令 beq 和 bne 来比较结果并进行分支。

RISC-V 的设计者决定添加独立的浮点寄存器。它们被称为 f0，f1，…，f31。因此，它们包括独立的用于浮点寄存器的取存指令：fld 和 fsd 用于双精度，flw 和 fsw 用于单精度。浮点数据传输指令的基址寄存器仍为整点寄存器。从内存中取出两个单精度数，相加，然后将总和存回到内存的 RISC-V 代码如下所示：

```
flw     f0, 0(x10) //  Load 32-bit F.P. number into f0
flw     f1, 4(x10) //  Load 32-bit F.P. number into f1
fadd.s  f2, f0, f1 //  f2 = f0 + f1, single precision
fsw     f2, 8(x10) //  Store 32-bit F.P. number from f2
```

单精度寄存器只是双精度寄存器的后半部分。注意，与整点寄存器 x0 不同，浮点寄存器 f0 没有硬连接到常数 0。

图 3-17 总结了本章所涉及的 RISC-V 体系结构的浮点部分，支持浮点的新增部分用灰色标记了出来。浮点指令使用与整点相同的指令格式：载入指令（load）使用 I 型格式，存储指令（store）使用 S 型格式，算术指令使用 R 型格式。

RISC-V浮点操作数

名称	示例	注
32个浮点寄存器	f0～f31	一个f寄存器可以保存一个单精度浮点数或一个双精度浮点数
2^{61}个存储双字	Memory[0], Memory[8],…, Memory[18 446 744 073 709 551 608]	只能通过数据传输指令访问。RISC-V使用字节地址，因此连续的双字相差8。存储器保存数据结构、数组和溢出寄存器的内容

RISC-V浮点汇编语言

类别	指令	示例	含义	注
算术	单精度浮点加法	fadd.s f0, f1, f2	f0 = f1 + f2	(单精度)浮点数加法
	单精度浮点减法	fsub.s f0, f1, f2	f0 = f1 - f2	(单精度)浮点数减法
	单精度浮点乘法	fmul.s f0, f1, f2	f0 = f1 * f2	(单精度)浮点数乘法
	单精度浮点除法	fdiv.s f0, f1, f2	f0 = f1 / f2	(单精度)浮点数除法
	单精度浮点平方根	fsqrt.s f0, f1	f0 = √f1	(单精度)浮点数平方根
	双精度浮点加法	fadd.d f0, f1, f2	f0 = f1 + f2	(双精度)浮点数加法
	双精度浮点减法	fsub.d f0, f1, f2	f0 = f1 - f2	(双精度)浮点数减法
	双精度浮点乘法	fmul.d f0, f1, f2	f0 = f1 * f2	(双精度)浮点数乘法
	双精度浮点除法	fdiv.d f0, f1, f2	f0 = f1 / f2	(双精度)浮点数除法
	双精度浮点平方根	fsqrt.d f0, f1	f0 = √f1	(双精度)浮点数平方根
比较	单精度浮点相等	feq.s x5, f0, f1	x5 = 1 if f0 == f1, else 0	(单精度)浮点数比较
	单精度浮点小于	flt.s x5, f0, f1	x5 = 1 if f0 < f1, else 0	(单精度)浮点数比较
	单精度浮点小于或等于	fle.s x5, f0, f1	x5 = 1 if f0 <= f1, else 0	(单精度)浮点数比较
	双精度浮点相等	feq.d x5, f0, f1	x5 = 1 if f0 == f1, else 0	(双精度)浮点数比较
	双精度浮点小于	flt.d x5, f0, f1	x5 = 1 if f0 < f1, else 0	(双精度)浮点数比较
	双精度浮点小于或等于	fle.d x5, f0, f1	x5 = 1 if f0 <= f1, else 0	(双精度)浮点数比较
数据传输	浮点取字	flw f0, 4(x5)	f0 = Memory[x5 + 4]	从内存中取单精度字到浮点寄存器
	浮点取双字	fld f0, 8(x5)	f0 = Memory[x5 + 8]	从内存中取双精度字到浮点寄存器
	浮点存字	fsw f0, 4(x5)	Memory[x5 + 4] = f0	将浮点寄存器中的单精度字存储到内存
	浮点存双字	fsd f0, 8(x5)	Memory[x5 + 8] = f0	将浮点寄存器中的双精度字存储到内存

图 3-17 截止到目前已涉及的RISC-V浮点体系结构。这些信息也可以在本书前面的RISC-V参考数据卡的第2列中找到

硬件/软件接口 在支持浮点算术时，体系结构设计者面临的一个问题是，选择使用和整点指令相同的寄存器，还是要为浮点添加一组专用寄存器。由于程序通常会在不同的数据集上分别执行整点运算和浮点运算，因此设置独立的寄存器只会略微增加执行程序所需的指令数。其影响主要是来自于要创建一组不同的数据传输指令，以便在浮点寄存器和内存之间传输数据。

独立的浮点寄存器的好处是，在不需要增加指令位长的情况下，可获得倍增的寄存器数目，同时因为有独立的整点和浮点寄存器，可获得倍增的寄存器带宽，并且还能为浮点定制寄存器。例如，一些计算机将寄存器中所有类型的操作数转换为单一的内部格式。

例题 | 将浮点C程序编译为RISC-V汇编代码

将华氏温度转化为摄氏温度的C语言程序：

```
float f2c (float fahr)
{
    return ((5.0f/9.0f) *(fahr - 32.0f));
}
```

假设浮点参数fahr传入到寄存器f10中，而结果也要存放在寄存器f10中。请写出RISC-V汇编代码。

答案 | 我们假设编译器将三个浮点常数存放在内存中，并可通过寄存器x3轻松地访问到。首先用两条指令将常数5.0和9.0加载到浮点寄存器中：

```
f2c:
    flw f0, const5(x3) // f0 = 5.0f
    flw f1, const9(x3) // f1 = 9.0f
```

然后将它们相除，得到分数 5.0/9.0 的值：

```
fdiv.s f0, f0, f1 // f0 = 5.0f / 9.0f
```

（许多编译器在编译时就会计算 5.0 除以 9.0，并将单精度常数 5.0/9.0 保存在内存中，从而避免在运行时进行除法运算。）接下来，我们加载常数 32.0，然后从 fahr（f10）中减去它：

```
flw    f1, const32(x3) // f1 = 32.0f
fsub.s f10, f10, f1    // f10 = fahr - 32.0f
```

最后，我们将两个中间结果相乘，将积作为返回结果放入 f10，然后返回：

```
fmul.s f10, f0, f10 // f10 = (5.0f / 9.0f)*(fahr - 32.0f)
jalr   x0, 0(x1)    // return
```

现在让我们在矩阵上执行浮点运算，其代码在科学计算程序中是很常见的。

例题 | 将二维矩阵的浮点 C 程序编译为 RISC-V 汇编代码

大多数浮点计算都是以双精度执行的。让我们执行 $C = C + A * B$ 的矩阵乘法。这是一种双精度通用矩阵乘法，通常被称为 DGEMM。我们将在 3.8 节以及后续的第 4、5、6 章中再次看到 DGEMM 的版本。现在假设 A、B、C 都是 32×32 的二维矩阵。

```
void mm (double c[][], double a[][], double b[][])
{
  size_t i, j, k;
   for (i = 0; i < 32; i = i + 1)
    for (j = 0; j < 32; j = j + 1)
     for (k = 0; k < 32; k = k + 1)
        c[i][j] = c[i][j] + a[i][k] *b[k][j];
}
```

数组起始地址作为函数参数，它们分别存放在 x10、x11 和 x12 中。假设整数变量分别存放在 x5、x6 和 x7 中。请写出程序主体的 RISC-V 汇编代码。

答案 | 请注意，c[i][j] 用于最内层的循环中。由于该层的循环变量是 k，它不会影响 c[i][j]，所以我们可以避免每次循环加载和存储 c[i][j]。编译器在循环外将 c[i][j] 加载到寄存器中，然后将 a[i][k] 和 b[k][j] 的乘积累加到这个寄存器中，在最内层循环结束后，将总和存放到 c[i][j] 中。我们使用汇编伪指令 li 让代码更简洁，它表示把一个常数加载到寄存器中。

该程序主体首先将循环终止值 32 保存到临时寄存器，然后初始化三个 for 循环的循环变量：

```
    mm:...
       li x28, 32    //  x28 = 32 (row size/loop end)
       li x5, 0      //  i = 0; initialize 1st for loop
L1:    li x6, 0      //  j = 0; initialize 2nd for loop
L2:    li x7, 0      //  k = 0; initialize 3rd for loop
```

要计算出 c[i][j] 的地址，我们需要知道一个 32×32 的二维数组是如何存储在内存中的。如你所料，它的排布和 32 个有 32 个元素的一维数组一样。因此，第一步是跳过 i 个“一维数组”或行，以获得我们想要的一维数组的起始地址。所以我们将第一维中的下标

乘以行的大小 32。由于 32 是 2 的幂，我们可以使用移位操作代替：

```
slli  x30, x5, 5    //  x30 = i * 2^5(size of row of c)
```

现在再加上第 2 个下标去选择我们需要的所在行的第 j 个元素：

```
add  x30, x30, x6    //  x30 = i * size(row) + j
```

为了将这个和变成一个字节索引，我们将它乘以一个矩阵元素所占的字节大小。由于每个元素都是双精度的，占 8 个字节，我们可以左移 3 位：

```
slli  x30, x30, 3    //  x30 = byte offset of [i][j]
```

接下来，我们将这个总和加到 c 的基地址上，得到了 c[i][j] 的绝对地址，然后将双精度数 c[i][j] 加载到 f0 中：

```
add  x30, x10, x30  //  x30 = byte address of c[i][j]
fld  f0, 0(x30)     //  f0 = 8 bytes of c[i][j]
```

以下五条指令几乎与上面五条指令相同：计算地址，然后加载双精度数 b[k][j]：

```
L3: slli  x29, x7, 5     //  x29 = k * 2^5(size of row of b)
    add   x29, x29, x6   //  x29 = k * size(row) + j
    slli  x29, x29, 3    //  x29 = byte offset of [k][j]
    add   x29, x12, x29  //  x29 = byte address of b[k][j]
    fld   f1, 0(x29)     //  f1 = 8 bytes of b[k][j]
```

同样，接下来的五条指令就像上面五条，计算地址然后加载双精度数 a[i][k]：

```
slli  x29, x5, 5     //  x29 = i * 2^5(size of row of a)
add   x29, x29, x7   //  x29 = i * size(row) + k
slli  x29, x29, 3    //  x29 = byte offset of [i][k]
add   x29, x11, x29  //  x29 = byte address of a[i][k]
fld   f2, 0(x29)     //  f2 = a[i][k]
```

现在我们已经加载了所有的数据，终于准备好进行浮点运算了！我们将位于寄存器 f2 和 f1 中的 a 和 b 的元素相乘，然后累加到寄存器 f0 中：

```
fmul.d  f1, f2, f1  // f1 = a[i][k] * b[k][j]
fadd.d  f0, f0, f1  // f0 = c[i][j] + a[i][k] * b[k][j]
```

最后的部分需要将循环变量 k 加 1，如果 k 不等于 32，则返回继续循环；如果 k 等于 32，那么最内层的循环已经结束，我们需要将寄存器 f0 中的累加结果存回到 c[i][j] 中：

```
addi  x7, x7, 1    //  k = k + 1
bltu  x7, x28, L3  //  if (k < 32) go to L3
fsd   f0, 0(x30)   //  c[i][j] = f0
```

同样，最后的 6 条指令递增中间和最外层循环的循环变量，如果循环变量不等于 32 则返回循环，如果等于 32 则退出循环：

```
addi  x6, x6, 1    //  j = j + 1
bltu  x6, x28, L2  //  if (j < 32) go to L2
addi  x5, x5, 1    //  i = i + 1
bltu  x5, x28, L1  //  if (i < 32) go to L1
. . .
```

后面的图 3-20 展示了 x86 的汇编语言代码，和图 3-19 中的 DGEMM 版本略有不同。

详细阐述 C 和许多其他编程语言都使用示例中讨论的数组排布，称为行主序。Fortran

改为使用列主序，即将数组按列依次存储。

详细阐述 将整点和浮点寄存器分开的另一个原因是20世纪80年代的微处理器没有足够的晶体管来将浮点单元和整点单元集成到同一块芯片上。因此，浮点单元，包括浮点寄存器，作为可选的辅助芯片。这样的可选加速器芯片被称为协处理器芯片。自90年代以来，在微处理器芯片上集成了浮点单元（以及几乎所有的其他功能单元），因此在今天，如果有人提及“协处理器”，就像提及“累加器”和“磁芯处理器”一样，已经是陈旧术语了。

详细阐述 如3.4节所述，加速除法比乘法更具挑战性。除了SRT之外，另一种利用快速乘法器的技术是牛顿迭代法，它将除法变为寻找函数的零点来计算倒数$1/c$，然后再乘以另一个操作数。如果不计算许多额外的位，迭代技术就不能进行正确的舍入。TI的一款芯片通过在求倒数时额外多求几位来解决这个问题。

详细阐述 Java在浮点数据类型和运算的定义中也采用了IEEE 754标准。因此，第一个例子中的代码可以很好地生成一类将华氏转换为摄氏温度的方法。

上面的第二个例子使用了多维数组，这个在Java中不被显式支持。Java允许在数组里嵌套数组，但每个数组都可以有自己的长度，这与C语言中的多维数组不同。与第2章中的示例类似，第二个示例的Java版本需要大量的数组边界检查代码，包括在行访问结束时计算新的长度。它还需要检查对象引用是否非空。

3.5.7　精确算术

和整数能够精确表示最小数和最大数之间的每个数不同，浮点数无法做到真正精确的表示，通常只能用近似值来表示。原因在于，任意两个实数（比如1和2）之间都有无穷多个实数，而双精度浮点数能精确表示的最多只有2^{53}个数。我们能做的就是获得和实际数字接近的浮点数表示。因此，IEEE 754提供了几种舍入的方法，让程序员选择想要的近似值。

保护位：在浮点运算的中间计算中，保留在右侧的两个额外位的第一位，用于提高舍入精度。

舍入听起来足够简单，但要精确地进行舍入则需要硬件在计算时包含额外的位。在前面的例子中，我们很难说清中间结果到底该占用多少位，但显然，如果每个中间结果必须被截断为精确的位数，那么最后就无法舍入了。因此，IEEE 754在中间计算时，总是在右边保留两个额外的位，分别称为**保护位**（guard）和**舍入位**（round）。我们通过一个十进制的例子来说明它们的好处。

舍入位：使中间结果符合浮点格式的方法，目标通常是找到符合格式的最接近的数。它也是在浮点运算的中间运算中保留在右侧的两个额外位的第二位，可以提高舍入精度。

例题｜利用保护位进行舍入

求$2.56_{10} \times 10^0$和$2.34_{10} \times 10^2$相加之和，假设有3位有效十进制位。将结果舍入到用3位有效十进制位能表示的与精确结果最相近的数，首先使用保护位和舍入位，然后不使用，观察区别。

答案｜ 首先，我们必须将较小的数字向右移以对齐指数，所以$2.56_{10} \times 10^0$变为$0.0256_{10} \times 10^2$。由于有保护位和舍入位，所以当我们对齐指数时，能够表示两个最低有效位。其中保护位为5，舍入位为6，和为：

$$\begin{array}{r} 2.3400_{10} \\ +\ 0.0256_{10} \\ \hline 2.3656_{10} \end{array}$$

总和为 $2.3656_{10} \times 10^2$。因为需要舍入 2 位，所以我们以 50 为界，如果后两位的值在 0~49 之间，则将其直接舍去，如果值在 51~99 之间，则舍入后向高位进 1。因此将和舍入到 3 位有效位的结果为 $2.37_{10} \times 10^2$。

再观察不带保护位和舍入位的情况，会在计算中丢掉 2 位有效位。新的和为：

$$\begin{array}{r} 2.34_{10} \\ +\ 0.02_{10} \\ \hline 2.36_{10} \end{array}$$

结果为 $2.36_{10} \times 10^2$，比前一个结果最后一位小 1。

舍入的最坏情况是实际的数刚好在两个浮点表示之间，浮点的精度通常以有效数位中最低有效位的错误位数来衡量，这种衡量方式称为**最后位置单位的数目**，即 ulp。如果一个数的最低有效位比实际小 2，则称少了 2ulp。在没有上溢、下溢或无效操作引发例外的情况下，IEEE 754 标准保证计算机使用的数的误差在半个 ulp 以内。

最后位置单位：用于表示在实际数和可表示数之间的有效数位中最低有效位上的误差位数。

详细阐述 虽然上面的例子确实只需要一个额外位就够了，但乘法可能需要两个额外位。因为二进制积可能有一个前导 0，因此，规格化时必须将积左移一位。这就将保护位转换为积的最低有效位，留下舍入位来帮助进行精确舍入。

IEEE 754 有四种舍入模式：总是向上舍入（朝 $+\infty$），总是向下舍入（朝 $-\infty$），截断，以及舍入到最接近的偶数。最后一种模式决定了数字恰好在两整数中间一半的情况下该怎么做。美国国内税务局（IRS）总是会在计算时向上舍入 0.50 美元。一个更公平的方法是前一半时间向上舍入，另一半时间向下舍入。IEEE 754 标准规定，如果中间结果的最低有效位是奇数则加 1，如果是偶数则截断。该方法总是在中间结果的最低有效位产生一个 0，因而有了这种舍入模式的名称。这种模式是最常用的，也是 Java 支持舍入的唯一模式。

额外舍入位的目的是让计算机获得相同的结果，就如同先以无限精度计算出中间结果再舍入一样。为了支持这个目标并舍入到最接近的偶数，这个标准设置了除了保护位和舍入位的第三位；只要在舍入位的右边有非零位，就将其设为 1。这个“粘滞位”允许计算机在舍入时能分辨出 0.50...00 和 0.50...01 之间的差异。

粘滞位可能会置 1，例如，在加法中将较小的数右移时。假设在上面的例子中我们将 $5.01_{10} \times 10^{-1}$ 和 $2.34_{10} \times 10^2$ 相加。即使有保护位和舍入位，我们将 0.0050 加到 2.34，也会得到和为 2.3450。因为右边有非零位，所以粘滞位将被置 1。如果没有粘滞位来记住是否有任何 1 被移出，我们会假设该数字等于 2.345000...00，并且舍入到最接近的偶数 2.34。通过粘滞位记住该数字大于 2.345000...00，我们会舍入到 2.35。

粘滞位：用于舍入时除了保护位和舍入位之外的位，当右侧有非零位时就设为 1。

详细阐述 RISC-V、MIPS-64、PowerPC、SPARC64、AMD SSE5 和 Intel AVX 架构都提供了一条单独的指令来对三个寄存器进行乘法和加法操作：$a=a+(b \times c)$。显然，这条指令允许这种常见操作具有更高的浮点性能。同样重要的是，不是在不同的指令中执行两次舍入——在乘法之后，然后在加法之后——乘加指令可以在加法之后只执行一次单独的舍入。该单独舍入步骤提高了乘加操作的精度。这种带单独舍入的操作被称为混合乘加。它被添加到修订的 IEEE 754-2008 标准中（见 3.11 节）。

混合乘加：一条既执行乘法又执行加法的浮点指令，但只在加法后进行舍入。

3.5.8 总结

下面的“重点”强化了第2章中的存储程序概念：信息的含义不能仅仅通过查看位来确定，同样的位可以表示各种不同的意思。本节告诉我们计算机算术是有限精度的，因此和自然算术所得结果可能不同。例如，IEEE 754标准浮点表示

$$(-1)^{S}\times(1+\text{有效位数})\times 2^{(\text{指数}+\text{偏移量})}$$

几乎总是实数的近似值。计算机系统必须注意将现实世界中的算术和计算机算术之间的差距最小化，程序员有时需要意识到这种近似可能带来的后果。

重点 位模式没有固有的含义。它们可能代表有符号整数、无符号整数、浮点数、指令、字符串等。表示的内容取决于指令对字中的这些位执行什么操作。

现实世界中的数和计算机中的数的主要区别在于计算机中数的字长有限，因此精度有限；有可能因为计算一个太大或太小的数而导致无法用一个字表示。程序员必须记住这些限制并相应地编写程序。

硬件/软件接口 在上一章中，我们介绍了C语言的各种存储类型（参见2.7节中的硬件/软件接口部分）。下表显示了一些C语言和Java的数据类型、数据传输指令以及对第2章和本章中出现的那些数据类型的操作指令。请注意，Java没有无符号整型。

C语言数据类型	Java数据类型	数据传输	操作
long long int	long	ld, sd	add, sub, addi, mul, mulh, mulhu, mulhsu, div, divu, rem, remu, and, andi, or, ori, xor, xori
unsigned long long int	–	ld, sd	add, sub, addi, mul, mulh, mulhu, mulhsu, div, divu, rem, remu, and, andi, or, ori, xor, xori
char	–	lb, sb	add, sub, addi, mul, div, divu, rem, remu, and, andi, or, ori, xor, xori
short	char	lh, sh	add, sub, addi, mul, div, divu, rem, remu, and, andi, or, ori, xor, xori
float	float	flw, fsw	fadd.s, fsub.s, fmul.s, fdiv.s, feq.s, flt.s, fle.s
double	double	fld, fsd	fadd.d, fsub.d, fmul.d, fdiv.d, feq.d, flt.d, fle.d

详细阐述 为了适应可能包括NaN的比较，该标准提供有序和无序作为比较选项。RISC-V不提供无序比较的指令，但经过设计的有序比较序列具有相同的效果。（Java不支持无序比较。）

为了从浮点运算中获得更大的精确度，该标准允许一些数以非规格化的形式表示。IEEE允许出现非规格化的数字（也称为denorm或subnormal），从而可以缩小0和最小的规格化数之间的差距。它们具有与零相同的指数，但是有效位非零。它们允许有效数位逐渐变小，直到变为0，这称为逐步下溢。例如，最小的单精度规格化正数是：

$$1.0000\ 0000\ 0000\ 0000\ 0000\ 000_2\times 2^{-126}$$

但最小的单精度非规格化数是：

$$0.0000\ 0000\ 0000\ 0000\ 0000\ 001_2\times 2^{-126}，即\ 1.0_2\times 2^{-149}$$

对于双精度，非规格化间隙为 1.0×2^{-1022} 到 1.0×2^{-1074}。

对于试图实现快速浮点计算单元的设计人员来说，偶尔会出现的非规格化操作数是一个令人头疼的问题。因此，许多计算机会在出现非规格化操作数时引发例外，让软件处理相应操作。尽管软件的操作是完全可行的，但它们的低性能影响了非规格化数在可移植的浮点软件中的受欢迎程度。另外，如果程序员并未考虑到非规格化数的情况，那么他们的程序执行结果可能会让他们感到惊讶。

自我检测 修订后的 IEEE 754-2008 标准增加了一个带 5 位指数字段的 16 位浮点格式。你认为它可能代表的数字范围是多少？

1. $1.0000\ 00\times2^{0}$ 到 $1.1111\ 1111\ 11\times2^{31}$，0
2. $\pm1.0000\ 0000\ 0\times2^{-14}$ 到 $\pm\ 1.1111\ 1111\ 1\times2^{15}$，$\pm0$，$\pm\ \infty$，NaN
3. $\pm1.0000\ 0000\ 00\times2^{-14}$ 到 $\pm\ 1.1111\ 1111\ 11\times2^{15}$，$\pm0$，$\pm\ \infty$，NaN
4. $\pm1.0000\ 0000\ 00\times2^{-15}$ 到 $\pm\ 1.1111\ 1111\ 11\times2^{14}$，$\pm0$，$\pm\ \infty$，NaN

3.6 并行性与计算机算术：子字并行

由于手机、平板电脑或笔记本电脑中的每个微处理器都有自己的图形显示器，随着晶体管数量的增加，对于图形操作的支持也不可避免地会增加。

许多图形系统最初使用 8 位数据来表示三原色中的一种，外加 8 位来表示一个像素的位置。在电话会议和视频游戏中添加了扬声器和麦克风对声音进行支持。音频采样需要 8 位以上的精度，但 16 位精度就已经足够了。

所有微处理器都对字节和半字有特殊支持，使其在存储时占用更少的存储器空间（见 2.9 节），但在典型的整数程序中对这类大小数据的算术运算非常少，因此几乎不支持除数据传输之外的其他操作。架构师发现，许多图形和音频应用会对这类数据的向量执行相同操作。通过在 128 位加法器内划分进位链，处理器可以同时对 16 个 8 位操作数、8 个 16 位操作数、4 个 32 位操作数或 2 个 64 位操作数的短向量进行并行操作。

这种分割加法器的开销很小，但带来的加速可能很大。

将这种在一个宽字内部进行的并行操作称为*子字并行*（subword parallelism）。更通用的名称是*数据级并行*（data level parallelism）。对于单指令多数据，它们也被称为向量或 SIMD（参见 6.6 节）。多媒体应用程序的逐渐普及促使了支持易于并行计算的窄位宽操作的算术指令的出现。在撰写本书时，RISC-V 还没有开发利用子字并行性的额外指令。下一节将介绍该体系结构的实例。

3.7 实例：x86 中的 SIMD 扩展和高级向量扩展

x86 的原始 MMX（MultiMedia eXtension，*多媒体扩展*）包含操作整数短向量的指令。而后，SSE（Streaming SIMD Extension，*流式 SIMD 扩展*）提供了操作单精度浮点数短向量的指令。第 2 章指出，在 2001 年，Intel 在其体系结构中增加了包含双精度浮点寄存器及其操作的 144 条指令作为 SSE2 的一部分。它包含 8 个可用于浮点操作数的 64 位寄存器。AMD 将其扩展到 16 个寄存器，作为 AMD64 的一部分，称为 XMM，Intel 将其标记为 EM64T 以供使用。图 3-18 总结了 SSE 和 SSE2 指令。

数据传输指令	算术运算指令	比较指令
MOV[AU]{SS|PS|SD|PD} xmm, {mem|xmm}	ADD{SS|PS|SD|PD} xmm,{mem|xmm}	CMP{SS|PS|SD|PD}
	SUB{SS|PS|SD|PD} xmm,{mem|xmm}	
MOV[HL]{PS|PD} xmm, {mem|xmm}	MUL{SS|PS|SD|PD} xmm,{mem|xmm}	
	DIV{SS|PS|SD|PD} xmm,{mem|xmm}	
	SQRT{SS|PS|SD|PD} {mem|xmm}	
	MAX{SS|PS|SD|PD} {mem|xmm}	
	MIN{SS|PS|SD|PD} {mem|xmm}	

图 3-18　x86 的 SSE/SSE2 浮点指令。xmm 是指一个 128 位 SSE2 寄存器操作数，而 {mem|xmm} 则指一个存储器操作数，或一个 SSE2 寄存器操作数。上表使用正则表达式来表示指令的变体。因此，MOV [AU] { SS | PS | SD | PD} 表示 MOVASS、MOVAPS、MOVASD、MOVAPD、MOVUSS、MOVUPS、MOVUSD 和 MOVUPD 八条指令。方括号 [] 用来表示单字母替换式：A 表示在存储器中对齐的 128 位操作数；U 表示在存储器中未对齐的 128 位操作数；H 表示移动 128 位操作数的高半部分；L 表示移动 128 位操作数的低半部分。大括号 {} 和垂直竖线 | 用来表示基本操作的多个变体：SS 表示标量单精度浮点数，或 128 位寄存器中的 1 个 32 位操作数；PS 表示组合的单精度浮点数，或 128 位寄存器中的 4 个 32 位操作数；SD 表示标量双精度浮点数，或 128 位寄存器中的 1 个 64 位操作数；PD 表示组合的双精度浮点数，或 128 位寄存器中的 2 个 64 位操作数

除了在寄存器中存放单精度或双精度数，Intel 还允许将多个浮点操作数组合（packed）到单个 128 位 SSE2 寄存器中：4 个单精度或 2 个双精度。因此，SSE2 的 16 个浮点寄存器实际为 128 位宽。如果操作数能够在存储器中组织成 128 位对齐的数据，则每条 128 位数据传输指令可以载入和存储多个操作数。这种组合的浮点数格式可以并行运算 4 个单精度（PS）或 2 个双精度（PD）数。

2011 年，Intel 使用高级向量扩展（Advanced Vector Extensions，AVX）将寄存器的位宽再次翻倍，现称为 YMM。因此，现在单精度操作可以指定 8 个 32 位浮点运算或 4 个 64 位浮点运算。原有的 SSE 和 SSE2 指令现在可以操作 YMM 寄存器的低 128 位。因此，为了使用 128 位和 256 位操作，在 SSE2 操作的汇编指令前加上字母“v”（表示向量），然后使用 YMM 寄存器名而不是 XMM 寄存器名。例如，执行 2 个 64 位浮点加法的 SSE2 指令。

```
addpd  %xmm0, %xmm4
```

变为

```
vaddpd  %ymm0, %ymm4
```

该指令现产生 4 个 64 位浮点加法。Intel 已经宣布，计划将 AVX 寄存器首先扩展到 512 位，而后在 x86 体系结构版本中再扩展到 1024 位。

详细阐述 AVX 还对 x86 添加了三地址指令。例如，vaddpd 现在可以指定

```
vaddpd  %ymm0, %ymm1, %ymm4    // %ymm4 = %ymm0 + %ymm1
```

而标准的两地址版本为：

```
addpd  %xmm0, %xmm4  // %xmm4 = %xmm4 + %xmm0
```

（和 RISC-V 不同，x86 的目的操作数在右边。）三地址可以减少计算所需的寄存器数和指令数。

3.8 加速：子字并行和矩阵乘法

为了说明子字并行对性能的影响，我们将先后在 Intel Core i7 上运行相同但没有 AVX 和带有 AVX 的代码。图 3-19 展示了一个用 C 编写的未优化版本的矩阵乘法代码。正如 3.5 节中所见，该程序通常被称为 DGEMM，表示双精度通用矩阵乘法。从这个版本开始，我们添加了新的名为“加速”的小节来说明在底层硬件上使用适配软件的性能提升，这里的底层硬件是 Sandy Bridge 版本的 Intel Core i7 微处理器。在第 3、4、5 和 6 章中，新的“加速”小节将使用每章介绍的思想逐步提高 DGEMM 的性能。

```
1.  void dgemm (size_t n, double* A, double* B, double* C)
2.  {
3.    for (size_t i = 0; i < n; ++i)
4.      for (size_t j = 0; j < n; ++j)
5.      {
6.        double cij = C[i+j*n]; /* cij = C[i][j] */
7.        for(size_t k = 0; k < n; k++ )
8.          cij += A[i+k*n] * B[k+j*n]; /*cij+=A[i][k]*B[k][j]*/
9.        C[i+j*n] = cij; /* C[i][j] = cij */
10.     }
11. }
```

图 3-19 未优化的双精度矩阵乘法的 C 语言版本，称为双精度通用矩阵乘法 DGEMM。因为我们将矩阵维数作为参数 n 传递，所以此版本的 DGEMM 使用一维版的矩阵 C、A、B，以及地址运算来获得更好的性能，而不是使用在 3.5 节中看到的更直观的二维数组。注释提醒我们使用这个更加直观的符号

图 3-20 给出了图 3-19 内部循环的 x86 汇编代码。以字母“v”开头的五条浮点指令类似于 AVX 指令，但注意它们使用 XMM 寄存器而非 YMM 寄存器，且它们在指令名称中包含 sd 以表示标量双精度。我们将简要定义子字并行指令。

```
1.  vmovsd (%r10),%xmm0                 // Load 1 element of C into %xmm0
2.  mov    %rsi,%rcx                    // register %rcx = %rsi
3.  xor    %eax,%eax                    // register %eax = 0
4.  vmovsd (%rcx),%xmm1                 // Load 1 element of B into %xmm1
5.  add    %r9,%rcx                     // register %rcx = %rcx + %r9
6.  vmulsd (%r8,%rax,8),%xmm1,%xmm1     // Multiply %xmm1,element of A
7.  add    $0x1,%rax                    // register %rax = %rax + 1
8.  cmp    %eax,%edi                    // compare %eax to %edi
9.  vaddsd %xmm1,%xmm0,%xmm0            // Add %xmm1, %xmm0
10. jg     30 <dgemm+0x30>              // jump if %eax > %edi
11. add    $0x1,%r11                    // register %r11 = %r11 + 1
12. vmovsd %xmm0,(%r10)                 // Store %xmm0 into C element
```

图 3-20 图 3-19 中未优化的 C 代码编译后生成的嵌套循环体的 x86 汇编语言。虽然它只处理 64 位数据，但编译器使用 AVX 版本的指令而非 SSE2，因此它的每条指令可以使用三地址而非二地址（参见 3.7 节中的详细阐述）

由于编译器编写者最终可以使用 x86 的 AVX 指令生成高质量代码，因此现在我们必须使用 C 循环体的内部函数来“作弊”，它或多或少地告诉编译器如何确切地生成高质量代码。图 3-21 是图 3-19 的加强版，给出了 GNU C 编译器生成的 AVX 代码。图 3-22 给出了带注释的 x86 代码，它是使用 gcc -O3 级优化编译后的输出。

```
1.  //include <x86intrin.h>
2.  void dgemm (size_t n, double* A, double* B, double* C)
3.  {
4.    for ( size_t i = 0; i < n; i+=4 )
5.      for ( size_t j = 0; j < n; j++ ) {
6.        __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
7.        for( size_t k = 0; k < n; k++ )
8.          c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
9.                  _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.                 _mm256_broadcast_sd(B+k+j*n)));
11.       _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.     }
13. }
```

图 3-21　优化的 C 语言版本的 DGEMM，使用 C 内部函数为 x86 生成 AVX 子字并行指令。图 3-22 展示了内部循环编译后生成的汇编语言

```
1.  vmovapd (%r11),%ymm0              // Load 4 elements of C into %ymm0
2.  mov     %rbx,%rcx                 // register %rcx = %rbx
3.  xor     %eax,%eax                 // register %eax = 0
4.  vbroadcastsd (%rax,%r8,1),%ymm1   // Make 4 copies of B element
5.  add     $0x8,%rax                 // register %rax = %rax + 8
6.  vmulpd (%rcx),%ymm1,%ymm1         // Parallel mul %ymm1,4 A elements
7.  add     %r9,%rcx                  // register %rcx = %rcx + %r9
8.  cmp     %r10,%rax                 // compare %r10 to %rax
9.  vaddpd %ymm1,%ymm0,%ymm0          // Parallel add %ymm1, %ymm0
10. jne     50 <dgemm+0x50>           // jump if not %r10 != %rax
11. add     $0x1,%esi                 // register % esi = % esi + 1
12. vmovapd %ymm0,(%r11)              // Store %ymm0 into 4 C elements
```

图 3-22　图 3-21 中优化 C 代码编译生成的嵌套循环体的 x86 汇编语言。注意与图 3-20 的相似之处，主要区别在于五个浮点操作现在使用 YMM 寄存器和 pd 版本的双精度组合指令，而非 sd 版本的双精度标量指令

图 3-21 第 6 行的声明使用 __m256d 数据类型，它告诉编译器该变量将保存 4 个双精度浮点值。同样位于第 6 行的内部函数 _mm256_load_pd() 使用 AVX 指令将 4 个双精度浮点数从矩阵 C 并行地（_pd）加载到 c0 中。在第 6 行中的地址计算 C+i+j*n 表示元素 C[i+j*n]。相对应地，第 11 行的最后一步使用内部函数 _mm256_store_pd() 把来自

c0 的 4 个双精度浮点数存储到矩阵 C 中。由于每次迭代都要处理 4 个元素，第 4 行的外部 for 循环的 i 递增 4，而不是像图 3-19 中第 3 行那样递增 1。

在循环内部，第 9 行首先使用 _mm256_load_pd() 再次载入 A 的 4 个元素。为了将这些元素乘以 B 的一个元素，在第 10 行中先使用内部函数 _mm256_broadcast_sd()，它在一个 YMM 寄存器中产生标量双精度数的 4 个相同副本（本例中的元素 B）。在第 9 行中，使用 _mm256_mul_pd() 同时乘 4 个双精度结果。最后，第 8 行的 _mm256_add_pd() 将 4 个乘积加到 c0 的 4 个和中。

图 3-22 给出了由编译器生成的内部循环体的 x86 代码。可以看到 5 条 AVX 指令（均以字母“v”开头且 5 条中的 4 条都是使用 pd 的双精度组合指令）与上面提到的 C 的内部函数相对应。该代码与图 3-20 中的代码非常相似：都使用 12 条指令，整数指令几乎相同（但寄存器不同），浮点指令的不同之处仅在于使用 XMM 寄存器的双精度标量指令（sd）和使用 YMM 寄存器的双精度组合指令（pd）；不同之处在于图 3-22 中的第 4 行，A 的每个元素必须乘以 B 的一个元素。一种解决方案是将 64 位 B 元素的 4 个相同副本依次放入 256 位的 YMM 寄存器，这正是 vbroadcastsd 指令所做的工作。

对于 32×32 的矩阵，图 3-19 中未优化的 DGEMM 在 2.6GHz Intel Core i7（Sandy Bridge）的一个内核上运行时性能为 1.7GigaFLOPS（每秒浮点操作数）。图 3-21 中优化代码的性能为 6.4GigaFLOPS。通过使用子字并行，很多操作可以获得 4 倍的加速，因此 AVX 版本要比原始版本快 3.85 倍，性能提升接近 4.0 倍。

详细阐述 正如 1.6 节详细阐述中所提到的，Intel 提供的 Turbo 模式暂时以相对较高的时钟频率运行，直到芯片过热。在 Turbo 模式下，Intel Core i7（Sandy Bridge）可以从 2.6GHz 增加 3.3GHz。以上是关闭 Turbo 模式的结果。若将其打开，由于时钟频率提高了 3.3 / 2.6 = 1.27 倍，使得未优化的 DGEMM 性能提升到 2.1GFLOPS 且 AVX 性能提升到 8.1GFLOPS。当一个八核芯片仅使用单个核时，Turbo 模式效果很好，因为在该情况下，其他核均空闲，所以单个核可以使用比共享情况下更多的功耗。

3.9 谬误与陷阱

算术谬误和陷阱通常来源于计算机算术的有限精度和自然算术的无限精度之间的差异。

> 数学可能被定义为我们永远不知道自己在说什么，也不知道自己所说是否属实的学科。
> *Bertrand Russell, Recent Words on the Principles of Mathematics, 1901*

谬误：正如左移指令可以代替一个乘以 2 的幂的整数，右移等同于除以一个 2 的幂的整数。

回想用二进制数 x（其中 x_i 表示第 i 位）表示数字

$$\cdots+(x^3\times2^3)+(x^2\times2^2)+(x^1\times2^1)+(x^0\times2^0)$$

将 c 位数字右移 n 位等同于除以 2^n。对于无符号整数也是如此。问题出在有符号整数。例如，假设要用 -5_{10} 除以 4_{10}，商应该是 -1_{10}。-5_{10} 的二进制补码表示是

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111011_2$$

根据这个谬误，右移两位应该除 4_{10}（2^2）：

$$00111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_2$$

符号位是 0，这个结果显然是错的。右移产生的值实际上是 $4\ 611\ 686\ 018\ 427\ 387\ 902_{10}$ 而不是 -1_{10}。

一种解决方法是进行算术右移，扩展符号位而移入0。产生 -5_{10} 的2位算术右移结果

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_2$$

结果是 -2_{10} 而不是 -1_{10}，虽然接近但仍不正确。

陷阱：浮点加法不满足结合律。

即使计算溢出，结合律也适用于二进制补码整数加法序列。然而，由于浮点数是实数的近似值，且计算机算术的精度有限，因此结合律不适用于浮点数。假定可以用浮点数表示大范围数字，当两个不同符号的大数加上一个小数字时会出现问题。例如，让我们来看看是否有 $c+(a+b)=(c+a)+b$。假设 $c=-1.5_{10}\times10^{38}$，$a=1.5_{10}\times10^{38}$，$b=1.0$，且都是单精度数。

$$\begin{aligned}c+(a+b)&=-1.5_{10}\times10^{38}+(1.5_{10}\times10^{38}+1.0)\\&=-1.5_{10}\times10^{38}+(1.5_{10}\times10^{38})\\&=0.0\\c+(a+b)&=(-1.5_{10}\times10^{38}+1.5_{10}\times10^{38})+1.0\\&=(0.0_{10})+1.0\\&=1.0\end{aligned}$$

由于浮点数的精度有限且结果为实数结果的近似值，$1.5_{10}\times10^{38}$ 远大于 1.0_{10}，所以 $1.5_{10}\times10^{38}+1.0$ 仍是 $1.5_{10}\times10^{38}$。这就是为什么 c、a、b 三者之和有0.0或1.0两种结果，这取决于浮点加法的顺序，所以 $c+(a+b)\neq(c+a)+b$。因此，浮点加法不满足结合律。

谬误：适用于整型数据类型的并行执行策略也适用于浮点数据类型。

一般来说，先编写串行运行程序，再编写并行运行程序，因此自然会产生一个问题，“这两个版本会得到相同的结果吗？”如果答案是否定的，你就假设在并行版本中有一个需要查找的bug。

该方法假定计算机算术从串行到并行不会影响结果。也就是说，如果你要同时加100万个数，无论使用1个处理器还是1000个处理器，都会得到相同的结果。这个假设适用于二进制补码整数，因为整数加法满足结合律。然而，由于浮点加法不满足结合律，因此该假设不适用。

这个谬误有一个更令人烦恼的情况可能出现在并行计算机上，其中操作系统调度器可能使用不同数量的处理器，这取决于并行计算机上正在运行的其他程序。由于每次参与运行的处理器数量不同，会导致浮点求和按不同顺序计算。即使运行相同的代码和相同的输入，每次得到的答案也会略有不同，这可能会让对并行无意识的编程人员感到困惑。

在这种困惑下，编写浮点数并行代码的程序员需要验证结果是否可信，即使结果与串行代码的答案可能不完全相同。涉及此问题的领域称为数值分析，该问题本身就可以成为一本教科书的主题。这也是用于数值计算的LAPACK和ScaLAPAK等函数库流行的原因之一，这些函数库已经在串行和并行形式下验证有效。

谬误：只有理论数学家关心浮点精度。

1994年11月的报纸头条证明了这种说法是谬误（见图3-23）。下面是头条幕后的故事。

Pentium使用标准浮点除法算法，每步生成多个商位，使用除数和被除数的最高有效位来猜测商的下两位。猜测取自包含 -2、-1、0、$+1$ 或 $+2$ 的查找表。猜测结果乘以除数并从余数中减去，从而产生一个新的余数。与不恢复余数除法一样，如果先前的猜测得到的余数太大，则在随后的过程中调整部分余数。

显然，Intel 工程师认为 80486 表中有五个元素永远不会被访问，并且优化了 Pentium 中的 PLA，使在该情况下返回 0 而不是 2。Intel 错了：前 11 位总是正确的，错误会偶尔出现在第 12 到 52 位或十进制数字的第 4 到第 15 位。

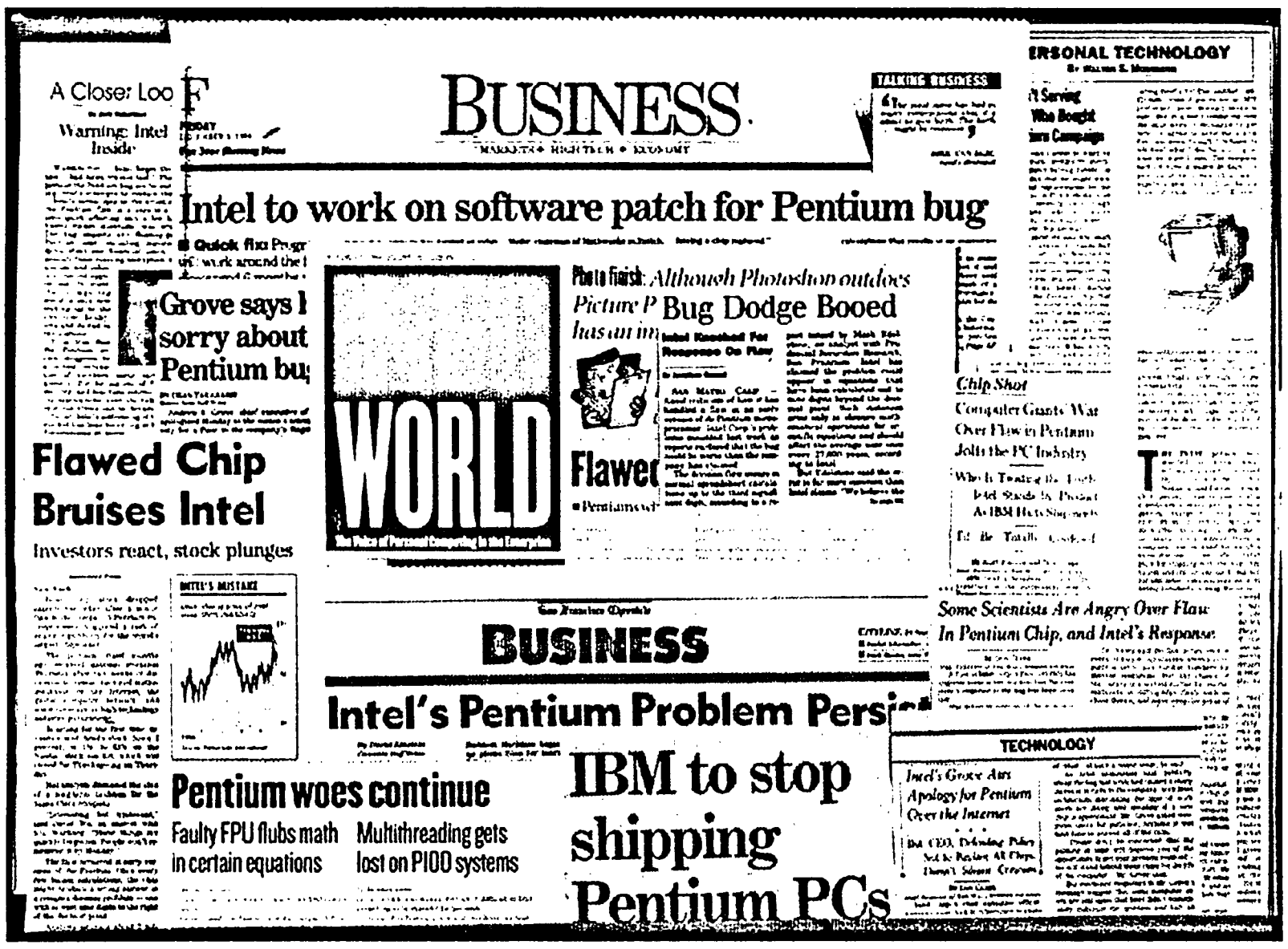

图 3-23 1994 年 11 月的报刊文章样本，包括《纽约时报》《圣何塞信使报》《旧金山纪事报》和《信息世界》。Pentium 的浮点除法 bug 甚至成为电视节目“David Letterman Late Show”的“十大新闻”。Intel 最终花费 3 亿美元替换掉了有 bug 的芯片

弗吉尼亚 Lynchburg 学院的数学教授 Thomas Nicely 在 1994 年 9 月发现了这个 bug。在致电 Intel 技术支持却没有得到官方回应后，他在网上公布了该发现。这在商业杂志上引发了一则故事，并引发 Intel 发布了一条声明。Intel 称其为仅会影响理论数学家的小故障，对于电子制表软件用户来说，该漏洞平均每 27 000 年才会发现一次。IBM 研究院很快反驳说电子制表软件用户平均每 24 天就能遇到一次这样的错误。很快，在 12 月 21 日，Intel 发布了以下声明表示认输：

> Intel 对近期发布的 Pentium 处理器缺陷的处理表示诚挚的歉意。“Intel Inside”标记意味着你的计算机拥有一个质量和性能都首屈一指的微处理器。成千上万的 Intel 员工非常努力工作以确保其真实有效。但没有微处理器总是完美的。Intel 仍相信，从技术上来说，一个极其微小的问题也有自己的生命周期。虽然 Intel 一定会对当前版本的 Pentium 处理器负责到底，但我们也认识到许多用户都有顾虑。我们想要解决这些顾虑。在计算机生命周期内的任何时候，Intel 都会为所有有需求的用户免费更换新版 Pentium 处理器（其中浮点除法缺陷已被更正）。

分析师估计，这次召回会导致 Intel 损失 5 亿美元，而 Intel 工程师当年没有拿到圣诞节奖金。

这个故事带给大家几点思考。如果在 1994 年 7 月修复这个漏洞会少花多少钱？修复 Intel 受损声誉的代价有多大？在像微处理器这样被广泛使用和信赖的产品中出现 bug 的相关责任是什么？