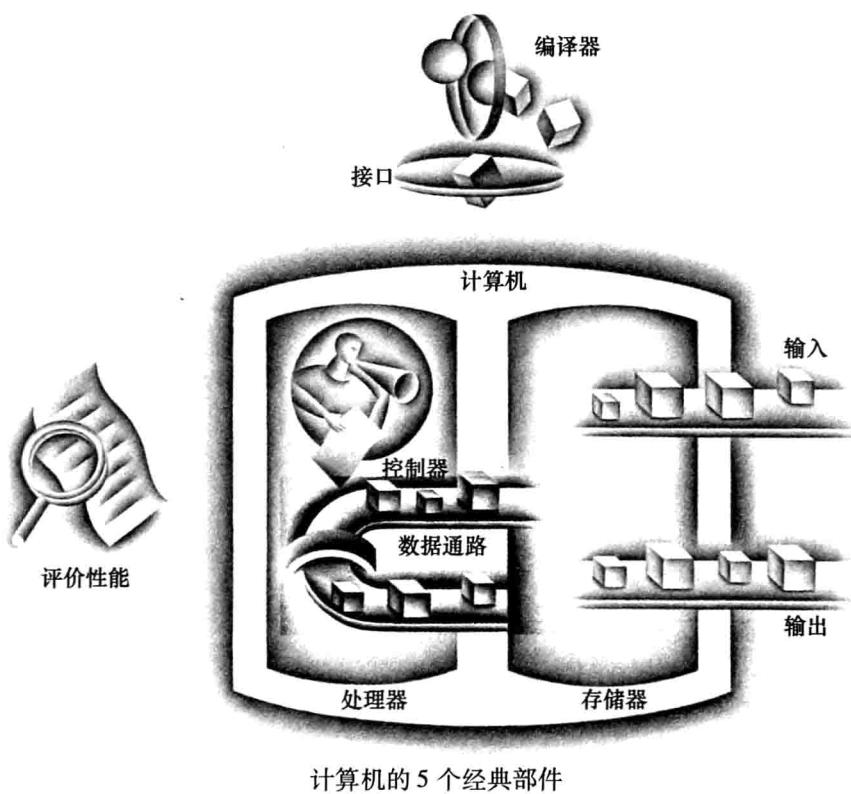


# 计算机的算术运算



数值的精确度是科学的灵魂。

——Sir D'Arcy Wentworth Thompson, 《On Growth and Form》, 1917

## 3.1 引言

计算机中的字由位组成。因此，字可以用二进制数来表示。第2章里提到整数可以表示成十进制或者二进制形式，但是其他常用的数据如何表示？例如：

- 小数和其他实数如何表示？
- 当一个操作生成了一个无法表示的大数时如何处理？
- 上述问题隐含着一个秘密：怎样用硬件真正地做乘法和除法？

本章的目的就是要揭示这些秘密，包括实数的表示方法、算术的算法、实现这些算法的硬件，以及如何在指令集中表示有关的内容。有了这些知识后，你就能解释在使用计算机的过程中遇到的各种不明白的事情了。另外，我们还将介绍如何使用该技术加速算术运算密集型程序的运行。

## 3.2 加法和减法

减法：加法的微妙朋友。

——No. 10, Top Ten Courses for Athletes at a Football Factory,  
David Letterman 等, 《Book of Top Ten Lists》, 1990

加法是计算机中必备的操作。数据从右到左逐位相加，同时进位也相应地向左传播，就如手动计算一样。减法也可采用加法实现：减数在简单的取反之后再进行加法操作。

### 01 例题·二进制加法和减法

在二进制下，首先计算  $7_{10}$  加上  $6_{10}$ ，然后计算  $7_{10}$  减去  $6_{10}$ 。

$$\begin{array}{r} 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0111_2 = 7_{10} \\ + \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0110_2 = 6_{10} \\ \hline = \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 1101_2 = 13_{10} \end{array}$$

只有右边 4 位发生变化。图 3-1 给出了和位与进位。其中，进位放在括号里，箭头标记了进位的方向。

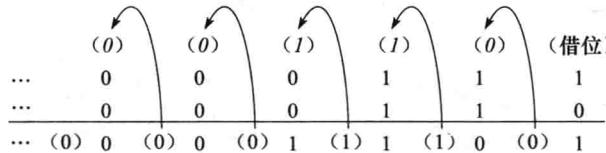


图 3-1 二进制加法，显示了从右到左的进位。最右边的位将 1 和 0 相加，得到该位的和为 1，该位的进位为 0。因此，右边第二位数的操作是  $0+1+1$ 。该操作的和为 0，进位为 1。第三位是  $1+1+1$  的和，得到的进位为 1，和为 1。第四位是  $1+0+0$ ，和为 1，无进位

176  
l  
178

### 01 答案

$7_{10}$  减去  $6_{10}$  可以直接操作：

$$\begin{array}{r} 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0111_2 = 7_{10} \\ - \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0110_2 = 6_{10} \\ \hline = \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0001_2 = 1_{10} \end{array}$$

或者通过加上  $-6$  的二进制补码来实现：

$$\begin{array}{r} 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0111_2 = 7_{10} \\ + \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1010_2 = -6_{10} \\ \hline = \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0001_2 = 1_{10} \end{array}$$

□

硬件规模总是有一定限制的，如字宽只有 32 位。当运算结果超过这个限制时，就会发生溢出。加法在什么情况下会溢出呢？当相加的两个源操作数符号相异时，不会发生溢出，原因是和必然不会大于其中一个操作数。如  $-10 + 4 = -6$ 。因为源操作数可以用 32 位的字表示，而“和”不会大于其中任何一个源操作数，所以和也可以用 32 位来表示。因此，当正数和负数相加时不会发生溢出。

在做减法时也会有类似的情况，只不过采用的规则相反：当源操作数的符号相同时，不会发生溢出。我们知道， $c - a = c + (-a)$ ，这是因为减法是把第二个源操作数变相反符号然后相加，所以当两个同符号的数作减法时，实际上是把两个符号相异的数相加，也不会发生溢出。

知道溢出在加减法中何时不会发生固然重要，但如何检测它何时发生？很明显，加或者减两个 32 位的数可能产生需要用 33 位来表示的结果。

如果缺少了第 33 位，则溢出发生时，符号位就可能被数值位占用而产生错误。因此，当两个正数相加但结果为负时，就说明发生了溢出，反之亦然。这个问题的和表示计算过程中发生了向符号位的进位操作。

在做减法时，如果用一个正数减去一个负数得到一个负的结果，或者用一个负数减去一个正数然后得到一个正的结果，则发生了溢出。这也意味着借位占用了符号位。图 3-2 给出了发生溢出的条件。

179

操作	操作数A	操作数B	表示结果有溢出的条件
$A+B$	$\geq 0$	$\geq 0$	$<0$
$A+B$	$<0$	$<0$	$\geq 0$
$A-B$	$\geq 0$	$<0$	$<0$
$A-B$	$<0$	$\geq 0$	$\geq 0$

图 3-2 加减法的溢出条件

上面介绍了如何检测计算机中的二进制补码操作的溢出，但无符号整数的溢出情况是如何的呢？由于无符号数通常用于表示内存地址，这种情况下的溢出可以忽略。

因此，计算机设计者必须提供一种方法，能够在某些情况下忽略溢出的发生，而在另一些情况下则能进行溢出的检测。MIPS 采用两种类型的算术指令来解决这个问题：

- 加法 (add)、立即数加法 (addi) 和减法 (sub)，这三条指令在溢出时产生异常。
- 无符号加法 (addu)、立即数无符号加法 (addiu) 和无符号减法 (subu)，这三条指令在发生溢出时不会产生异常。

因为 C 语言忽略溢出，所以 MIPS C 编译器总是采用无符号的算术指令 addu、addiu 和 subu，而不必考虑变量的类型。但是 MIPS Fortran 编译器会根据操作数的类型来选择相应的算术指令。

附录 B 描述了做加减法的算术逻辑单元 (arithmetic logic unit, ALU) 的硬件实现。

- ◎ 算术逻辑单元 (ALU)：用于执行加法、减法，通常也包括如逻辑与、逻辑或等逻辑操作的硬件。
- ◎ 异常：也叫中断，一种打断正常程序执行过程的事件，用于溢出检测。
- ◎ 中断：来自处理器外部的异常。(在某些体系结构中所有的异常都称为中断。)

**01 精解** 对于 addiu 的一个常见困惑是它的名字和对其立即数字段做什么操作。**u** 代表无符号数，这意味着加法操作不会产生溢出异常。然而，与 addi、slti 和 sltiu 指令类似，16 位立即数字段要符号扩展为 32 位。因此，即使操作是“无符号”的，立即数字段也是有符号的。

**01 硬件/软件接口** 计算机设计者必须考虑如何处理算术溢出。但是一些编程语言（如 C 和 Java）会忽略整数溢出，而 Ada 和 Fortran 语言则需要通知程序溢出。因此程序员或者是编程环境必须决定在溢出发生时如何处理。

MIPS 检测到溢出时会产生异常 (exception)，在许多计算机系统中也叫作中断 (interrupt)。从本质上来说，异常或中断是一种打断正常过程的系统调用。产生溢出的指令地址保存在一个寄存器中，而后计算机会跳到一个预先设定好的地址去执行相应的异常处理程序。保存异常地址的目的是为了在某些条件下能够在异常处理程序执行完后返回原程序继续执行。（4.9 节给出了有关异常的更多细节；第 5 章和第 6 章中描述了异常和中断发生的其他条件。）

MIPS 使用命名为异常程序计数器 (Exception Program Counter, EPC) 的寄存器来保存导致异常的指令地址。指令 mfc0 (move from system control) 用来将 EPC 存入一个通用寄存器，从而使 MIPS 软件可以通过寄存器跳转指令返回到导致异常的指令那里。

## 小结

本节主要指出，无论采用哪种数的表示方法，具有有限字长的计算机在进行算术操作时都

可能发生溢出。无符号数的溢出是容易检测的，但无符号数通常用于地址计算，因为程序通常并不需要检测地址计算的溢出，通常情况是使用自然数，所以这些溢出往往被忽略。有符号数的溢出检测比较麻烦，但是有些软件系统需要检测溢出，所以今天所有的计算机都支持溢出检测。

### 01 小测验

某些程序语言支持字节或者半字的二进制补码的整数算术，而 MIPS 只有整字的整数算数操作。回顾一下第 2 章的内容，MIPS 中也有字节和半字的数据传送指令。那么将会使用哪些 MIPS 指令？

1. 取数使用 lbu、lhu；算术操作采用 add、sub、mult、div；存数采用 sb、sh。
2. 取数使用 lb、lh；算术操作采用 add、sub、mult、div；存数采用 sb、sh。
3. 取数使用 lb、lh；算术操作采用 add、sub、mult、div；采用 AND 来屏蔽每次运算的结果到 8 位或者 16 位；存数采用 sb、sh。

**01 精解** 饱和 (saturating) 操作是通用微处理器中一个不常出现的特性。饱和意味着当计算结果溢出时，结果被设置为最大的正数或者最小的负数，而不像二进制补码运算那样采用取模操作来获得结果。饱和操作一般更适合多媒体操作。例如，当不断旋转收音机音量的旋钮时，起初声音逐渐增大，但如果大到一定值后声音突然变小，那么这样的收音机设计是不合理的。然而，对一台有饱和操作的收音机，当向最大值方向旋转音量旋钮到一定程度后，即使再旋转，音量也只会停在最大值上。标准指令集上媒体扩展通常提供饱和算法。

**181 01 精解** MIPS 在溢出时会产生异常，但和其他许多计算机不同，它没有测试溢出的条件分支。一个 MIPS 指令序列可以发现溢出。对于有符号加法，这个序列如下（见 2.6 节描述 xor 指令的精解）：

```
addu $t0, $t1, $t2 # $t0 = sum, but don't trap
xor $t3, $t1, $t2 # Check if signs differ
slt $t3, $t3, $zero # $t3 = 1 if signs differ
bne $t3, $zero, No_overflow # $t1, $t2 signs *,
                            # so no overflow
xor $t3, $t0, $t1 # signs =; sign of sum match too?
                    # $t3 negative if sum sign different
slt $t3, $t3, $zero # $t3 = 1 if sum sign different
bne $t3, $zero, Overflow # All 3 signs *; goto overflow
```

对于无符号加法 ( $$t0 = $t1 + $t2$ )，测试则为：

```
addu $t0, $t1, $t2      # $t0 = sum
nor $t3, $t1, $zero     # $t3 = NOT $t1
                        # (2's comp - 1:  $2^{32} - $t1 - 1$ )
slt $t3, $t3, $t2       # ( $2^{32} - $t1 - 1$ ) < $t2
                        #  $\Rightarrow 2^{32} - 1 < $t1 + $t2$ 
bne $t3,$zero,Overflow # if( $2^{32}-1 < $t1+$t2$ ) goto overflow
```

**01 精解** 在前文中我们说过，可以通过 mfc0 指令将 EPC 内容复制到一个寄存器，然后通过跳转寄存器返回到被中断的代码。这样做会导致一个有趣的问题：既然必须首先使用跳转寄存器传输 EPC 到一个寄存器，那么跳转寄存器该如何返回到被中断的位置，并恢复所有寄存器的原值呢？如果先恢复所有寄存器的原值，则来自 EPC 的返回地址就会被破坏。如果在恢复所有寄存器的原值时保留那个返回地址的寄存器不变，这样可以进行正确跳转，但是这也意味着在程序执行的任何时刻，异常会导致一个寄存器的值无法被恢复。两者都是不可行的。

为了将硬件设计从这一困境中解救出来，MIPS 允许程序员将寄存器 \$k0 和 \$k1 预留给操作系统。这些寄存器在异常时不会恢复。仅仅当 MIPS 编译器避免使用 \$at 寄存器时，汇编程序可以使用它作为临时寄存器（见 2.10 节的硬件/软件接口），编译器也可以避免使用寄存器 \$k0 和 \$k1，从而使它们空出来给操作系统。异常处理程序将返回地址放在其中的一个寄存器中，然后利用跳转寄存器返回指令地址。

**01 精解** 确定进位到达高位的速度变快，加法的速度也随之加快。有许多方案可以用来加速这个进位，最坏情况下的进位时间是加法器位长的  $\log_2$  的函数。预期信号传输更快是因为它们经过了更少的门电路序列，而加速进位需要更多门电路，最流行的结构是超前进位（carry lookahead）加法器，见附录 B 的 B.6 节。

182

### 3.3 乘法

乘法令人恼怒，除法更甚；比例运算困扰着我，做练习令我发疯。

——佚名，《Elizabethan manuscript》，1570

现在我们已经完成了对加法、减法的解释，本节开始分析更复杂的乘法操作。

首先，通过用普通写法表示的十进制数乘法来回忆一下乘法的步骤和操作数的名称。为简单起见，我们只用十进制数中的 0 和 1 来作为例子，计算  $1000_{10}$  乘以  $1001_{10}$ ：

$$\begin{array}{r}
 \text{被乘数} & 1000_{10} \\
 \times & 1001_{10} \\
 \hline
 & 1000 \\
 & 0000 \\
 & 0000 \\
 & 1000 \\
 \hline
 \text{积} & 1001000_{10}
 \end{array}$$

第一个源操作数称为被乘数（multiplicand），第二个源操作数称为乘数（multiplier），最终的结果称为积（product）。你会回忆起在学校学过的乘法规则：每次从右到左选取乘数的一位，乘以被乘数，然后相对上一个中间积，将当前积左移一位。

可以观察到，积的位数远远大于被乘数和乘数。事实上，如果我们忽略符号位，若被乘数为  $n$  位，乘数为  $m$  位，则积的位数为  $n + m$ 。即需要  $n + m$  位来表示所有可能的积。因此，像加法一样，乘法也需要处理溢出，因为我们经常需要两个 32 位长的数相乘产生一个 32 位长的积。

在这个例子中，我们只使用了十进制中的 0 和 1。因为只有两个选择，所以每一步的乘法都很简单：

- 1) 当乘数位为 1 时，只需要将被乘数 ( $1 \times$  被乘数) 复制到合适的位置。
- 2) 当乘数位为 0 时，将 0 ( $0 \times$  被乘数) 放置到合适的位置。

虽然上面十进制的例子是限制使用了 0 和 1，但二进制数的乘法只能使用 0 和 1，因此也只有这两种选择。

分析了乘法的基本原则之后，一般来讲下一步就会马上开始介绍乘法硬件及其优化。但为了更好地理解这一问题，我们打破这一传统，先通过倍数的生成来展示乘法硬件和算法的进化过程。首先，我们假设只使用正数为源操作数。

183

#### 3.3.1 顺序的乘法算法和硬件

该设计模拟我们在小学学过的算法。图 3-3 给出了硬件结构。我们画出了硬件，使得数据

流从顶至下，很像我们用纸和笔计算的方法。

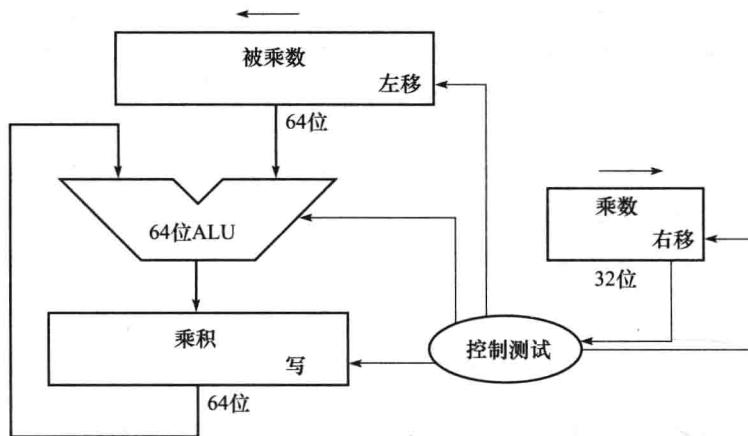


图 3-3 第一版乘法器硬件结构。被乘数寄存器、ALU 和积寄存器都是 64 位长，而乘数寄存器为 32 位长。（附录 B 对 ALU 进行了描述。）32 位的被乘数在开始时放置在被乘数寄存器的右半部分，然后每次左移一位。乘数则每次向相反的方向移动。算法开始时，积被初始化为 0。控制逻辑决定何时对被乘数和乘数寄存器进行移位，以及何时将新值写入积寄存器。

假设乘数放置在 32 位的乘数寄存器中，64 位的积寄存器被初始化为 0。从采用纸和笔计算的方法中，我们可以清楚地看到被乘数在每步需要左移一位，因为它需要与前面的中间结果相加。在经过 32 步后，32 位长的被乘数将要左移 32 位。因此，我们还需要一个 64 位的被乘数寄存器，且在初始化时 32 位的被乘数放在右半部分，左半部分清 0。然后，每执行一步，这个寄存器中的值就左移一位，将被乘数与 64 位积寄存器中的中间结果对齐并累加到中间结果。

图 3-4 给出了对于操作数的每一位的三个基本执行步骤。乘数的最低位（乘数的第 0 位）决定了被乘数是否被加到积寄存器上。第二步中的左移起着将被乘数左移的作用，就如同用纸和笔做乘法一样。第三步中的右移给出了下一个迭代中要用的乘数位。这三个步骤要重复执行 32 次来获得积。如果每步需要一个时钟周期，这个算法将需要大概 100 个时钟周期来完成两个 32 位的数相乘。像乘法这样的算术操作的相对重要性因程序而异，一般加法和减法出现的次数要比乘法频繁 5~100 倍。因此，在许多应用程序中，多步乘法不会显著影响性能。但 Amdahl 定律（见 1.10 节）告诉我们，如果一个慢速操作在程序中占据一定比重的话，也会限制程序的性能。

这个算法和硬件结构可以很容易改进

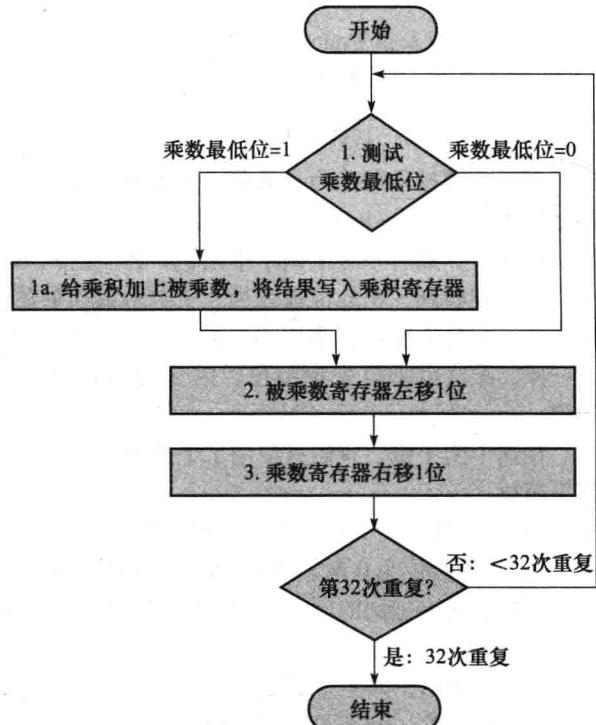


图 3-4 第一种乘法算法。其硬件结构见图 3-3。如果乘数的最低有效位为 1，则将被乘数加在积上，否则，进入下一步。在下两步中进行被乘数的左移和乘数的右移。这三个步骤需要重复 32 次。

成每一步只需要一个时钟周期。这些操作可以并行化来加速执行：当乘数位为 1 时，将乘数和被乘数进行移位，同时将被乘数和积相加。这时需要保证硬件测试的是乘数最右边的位，而且得到的是被乘数移位前的值。注意到加法器和寄存器中有未使用的部分后，可以通过将加法器和寄存器的位长减半来进一步优化这个硬件结构。如图 3-5 所示为修正后的硬件。

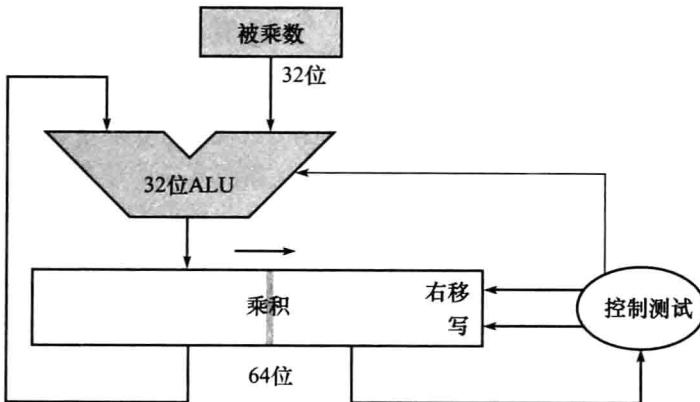


图 3-5 乘法器硬件的改进版。与图 3-3 中的第一版硬件结构相比，被乘数寄存器、ALU、乘数寄存器都是 32 位长，只有积寄存器是 64 位长。现在将积进行右移，单独的乘数寄存器也撤销了。乘数放在积寄存器的右半部分（乘法寄存器实际上应该是 65 位，以保存加法器的进位，但这里给出的是 64 位，以突出从图 3-3 的演变。）

**01 硬件/软件接口** 当乘数为常数时，乘法也可以用移位来替代。一些编译器将有短常数的乘法替换为一系列的移位和加法。因为左移一位就是将一个数放大两倍，左移和乘以 2 为底的指数有着等同的效果。正如第 2 章所提到的，几乎每个编译器都将以 2 为底的指数乘法替换为移位来进行优化。

### 01 例题·乘法算法

为了节省空间，使用的是 4 位长的数，计算  $2_{10} \times 3_{10}$ ，或  $0010_2 \times 0011_2$  的积。

### 01 答案

图 3-6 给出了按图 3-4 中标出的每一步执行后各个寄存器的值，最终结果为  $0000\ 0110_2$ ，即  $6_{10}$ 。加粗数据表示每步中寄存器值的变化。带圈的位用于决定下一步的操作。

迭代次数	步骤	乘数	被乘数	乘积
0	初始值	001①	0000 0010	0000 0000
1	1a: 1⇒乘积=乘积+被乘数	0011	0000 0010	<b>0000 0010</b>
	2: 左移被乘数	0011	<b>0000 0100</b>	0000 0010
	3: 右移乘数	<b>000①</b>	0000 0100	0000 0010
2	1a: 1⇒乘积=乘积+被乘数	0001	0000 0100	<b>0000 0110</b>
	2: 左移被乘数	0001	<b>0000 1000</b>	0000 0110
	3: 右移乘数	<b>000①</b>	0000 1000	0000 0110
3	1: 0⇒无操作	0000	0000 1000	0000 0110
	2: 左移被乘数	0000	<b>0001 0000</b>	0000 0110
	3: 右移乘数	<b>000①</b>	0001 0000	0000 0110
4	1: 0⇒无操作	0000	0001 0000	0000 0110
	2: 左移被乘数	0000	<b>0010 0000</b>	0000 0110
	3: 右移乘数	<b>0000</b>	0010 0000	0000 0110

图 3-6 使用图 3-4 中算法的乘法例子。圆圈圈起来的是下一步需要检测的位



### 3.3.2 有符号乘法

到目前为止，我们处理的对象都是正数。对于理解如何处理有符号乘法，最简单的方法是首先将被乘数和乘数转化为正数，并记住原来的符号位。这样，就可用上述最后的算法迭代31次，符号位不必参与运算。当符号相异时，积为负。

这表明后面的算法对于有符号数同样适用，只要知道这里要使用32位数据来表示通常要处理的无限数字。因此移位步骤需要对有符号数的乘积进行扩展。当算法结束时，低位字将是32位乘积。

### 3.3.3 更快速的乘法

摩尔定律为我们提供了非常充足的资源，使硬件设计者可以设计更快速的乘法器。我们可以在乘法运算开始的时候通过检查乘数的32位，来判定被乘数是否被加上。快速的乘法运算主要的思想是为乘数的每一位提供一个32位的加法器：一个用来输入被乘数和一乘数位相与的结果，另一个是上一个加法器的输出。

一种直接的方法是将每个右边的加法器的输出作为左边加法器的输入，形成一个高32的加法器栈。一种替换的方法是将32个加法器组织成一个并行树，如图3-7所示。这样，我们只需要等待 $\log_2(32)$ ，即5次32位长加法的时间，而不是等待32次加法的时间。

187

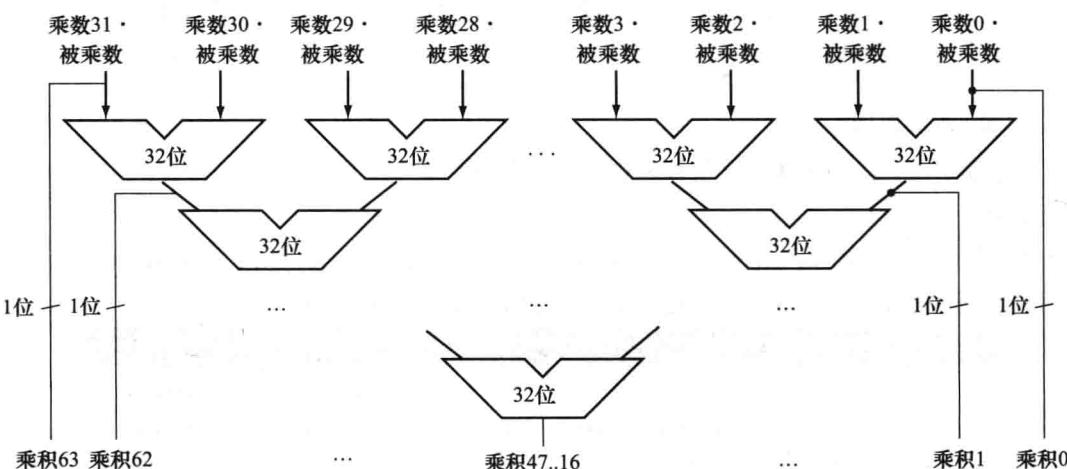


图3-7 快速乘法器硬件结构。这个结构使用31个加法器“展开循环”来实现最小的时延，而不是使用单个32位的加法器31次

事实上，通过使用进位保留加法器（见附录B的B.6节），乘法的计算速度可以快于5次加法。而且由于易于应用流水线设计执行，这样的结构可以同步支持多个乘法运算（见第4章）。

### 3.3.4 MIPS中的乘法

MIPS提供了一对单独的32位寄存器来容纳64位的积，称为Hi和Lo。为了产生正确的有符号积和无符号积，MIPS提供了两条指令：乘法(mult)和无符号乘法(multu)。为了取得32位的整数积，程序员需要使用mflo指令(move from lo)。MIPS汇编器为乘法生成了一条伪指令，它使用了三个通用寄存器，用mflo和mfhi指令将积送入指定的寄存器。

### 3.3.5 小结

乘法硬件只是简单地移位和加法，其算法类似于采用纸和笔的计算方法。编译器甚至会用移位指令来代替乘数为 2 的幂次的乘法操作。通过使用更多硬件的方法，可以并行做加法操作，从而提高运算速度。

**01 硬件/软件接口** MIPS 乘法指令都忽略溢出，所以要由软件来检测是否因积过大而 32 位不够表示。对于 multu 指令，如果 Hi 为 0 则无溢出；对于 mult 指令，如果 Hi 为 Lo 的符号位则也无溢出。可以使用指令 mfhi (move from hi) 将 Hi 的值移入一个通用寄存器来检测溢出。

188

## 3.4 除法

*Divide et impera*

——拉丁语，意为“分而治之”，引自 Machiavelli 的一句政治箴言，1532

和乘法相反的操作是除法，它用得较少，但很诡异。它甚至可能会出现数学上的无效操作：除数为 0。

首先通过十进制数的长除来回忆一下操作数的命名和小学时学习的除法算法。类似于前面，为简单起见，我们只使用十进制中的 0 和 1。这个例子是计算  $1001\ 010_{10}$  除以  $1000_{10}$ ：

$$\begin{array}{r}
 & \overline{1001_{10}} \quad \text{商} \\
 \text{除数 } 1000_{10} & \overline{1001010_{10}} \quad \text{被除数} \\
 -1000 \\
 \hline
 & 10 \\
 & 101 \\
 & 1010 \\
 -1000 \\
 \hline
 & 10_{10} \quad \text{余数}
 \end{array}$$

除法中的两个源操作数，称为被除数 (dividend) 和除数 (divisor)，结果称为商 (quotient)，还有一个第二结果，称为余数 (remainder)。这里用一种方式来表达它们之间的关系：

$$\text{被除数} = \text{商} \times \text{除数} + \text{余数}$$

这里余数要小于除数。在某些场合，程序使用除法指令只是为了获得余数，而忽视商。

- ① 被除数：被除的数。
- ② 除数：用于对被除数进行除法的数。
- ③ 商：除法的主要结果；乘以除数并加上余数产生被除数的数。
- ④ 余数：除法的第二个结果，加在商和除数的乘积上产生被除数的数。

这个过程中每次都尝试看最大能减掉多少，然后以此产生商。我们小心地选择出只用 0 和 1 的十进制例子，从而很容易判断出需要将多少倍的除数从被除数中减去：要么是 1 倍，要么是 0 倍。二进制数仅包含 0 和 1，所以二进制除法也仅有这两种选择，从而简化了二进制除法。

现在我们假设被除数和除数都为正，因此商和余数也都非负。除法的源操作数和两个结果都是 32 位宽，我们暂且忽略符号位。

### 3.4.1 除法算法及其硬件结构

图 3-8 给出了模拟小学学过的除法算法的硬件结构。在开始时，32 位的商寄存器设为 0。

算法每次的迭代将除数向右移一位。所以我们需要在开始时将除数放置在 64 位除数寄存器的左半边，然后每次右移一位来和被除数对齐。余数寄存器初始化为被除数。

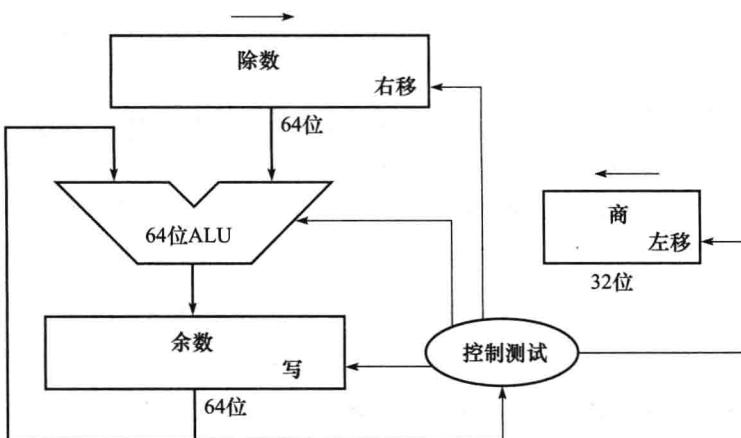


图 3-8 第一种除法器硬件结构。除数寄存器、ALU、余数寄存器都是 64 位宽，只有商寄存器是 32 位宽。32 位的除数开始放置在除数寄存器的左半部分，然后每次迭代右移一位。余数寄存器初始化为被除数。控制逻辑决定何时对除数和商寄存器进行移位以及何时将新值写入余数寄存器

图 3-9 给出了第一种除法算法的 3 个步骤。不像人那样聪明，计算机不可能提前知道除数是否小于被除数。所以需要在第 1 步中减去除数这与在小于时设置指令中的比较操作相同；如果结果为正，则除数小于等于被除数，所以我们取商为 1（第 2a 步）。如果结果为负，则通过将除数加上余数来恢复上一次的值，然后取商为 0（第 2b 步）。除数右移，然后再次迭代。迭代完成后，余数和商存放在以它们命名的寄存器中。

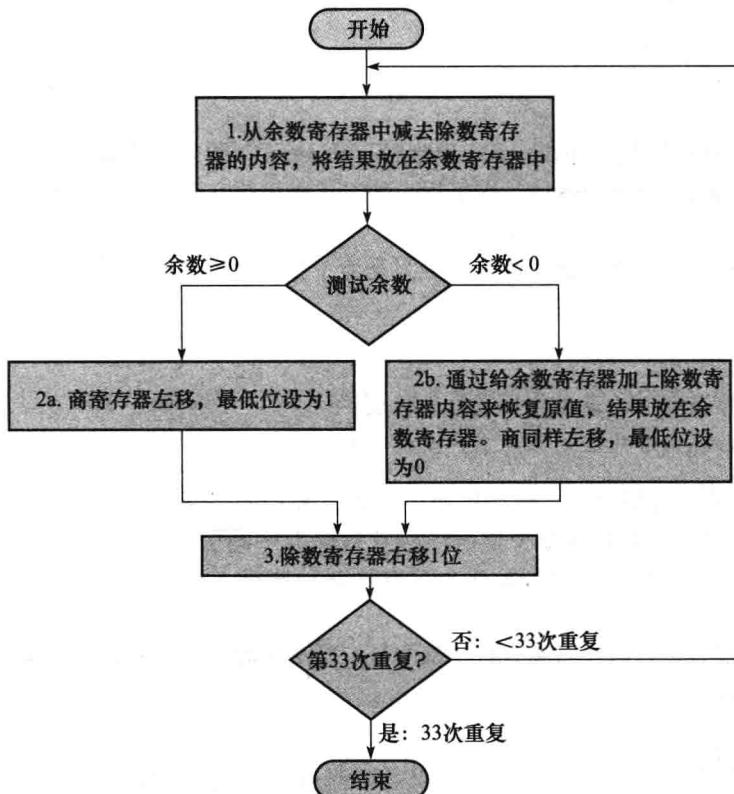


图 3-9 第一种除法算法，其硬件结构见图 3-8。如果余数为正，则将除数从被除数中减去，然后在第 2a 步取商为 1。如果第 1 步之后余数为负，则意味着除数不能从被除数中减去，所以在第 2b 步中商 0 并将除数加到余数上，即做第 1 步减法的逆操作。在第 3 步，进行最后的移位，根据下一个迭代的被除数，将除数适当对齐。这些步骤将要重复 33 次

### 01 例题·除法算法

为了节省篇幅，我们使用 4 位的数据。计算  $7_{10}$  除以  $2_{10}$ ，即  $0000\ 0111_2$  除以  $0010_2$ 。

### 01 答案

图 3-10 给出了每步中各个寄存器的值，其中，商为  $3_{10}$ ，余数为  $1_{10}$ 。注意到，在第 2 步中检测余数的正负只需要简单地测试余数寄存器的符号位是 0 还是 1 即可。令人惊讶的是，这个算法需要  $n+1$  步来获得适当的商和余数。

迭代次数	步骤	商	除数	余数
0	初始值	0000	0010 0000	0000 0111
1	1: 余数=余数-除数	0000	0010 0000	①110 0111
	2b: 余数<0 ⇒ +除数，商左移，上最低位上0	0000	0010 0000	0000 0111
	3: 除数右移	0000	0001 0000	0000 0111
2	1: 余数=余数-除数	0000	0001 0000	①111 0111
	2b: 余数<0 ⇒ +除数，商左移，上最低位上0	0000	0001 0000	0000 0111
	3: 除数右移	0000	0000 1000	0000 0111
3	1: 余数=余数-除数	0000	0000 1000	①111 1111
	2b: 余数<0 ⇒ +除数，商左移，上最低位上0	0000	0000 1000	0000 0111
	3: 除数右移	0000	0000 0100	0000 0111
4	1: 余数=余数-除数	0000	0000 0100	①000 0011
	2a: 余数≥0 ⇒ 商左移，上最低位上1	0001	0000 0100	0000 0011
	3: 除数右移	0001	0000 0010	0000 0011
5	1: 余数=余数-除数	0001	0000 0010	①000 0001
	2a: 余数≥0 ⇒ 商左移，上最低位上1	0011	0000 0010	0000 0001
	3: 除数右移	0011	0000 0001	0000 0001

图 3-10 除法的例子，采用图 3-9 中的算法。图中圈起来的位用于决定下一步的操作

算法和对应的硬件结构分别可以改进得更快、更便宜。加速是通过将源操作数和商移位与减法同时进行。注意到寄存器和加法器有未用的部分，可以通过将加法器和寄存器的位长减半来改进硬件结构，如图 3-11 所示为改进后的硬件结构。

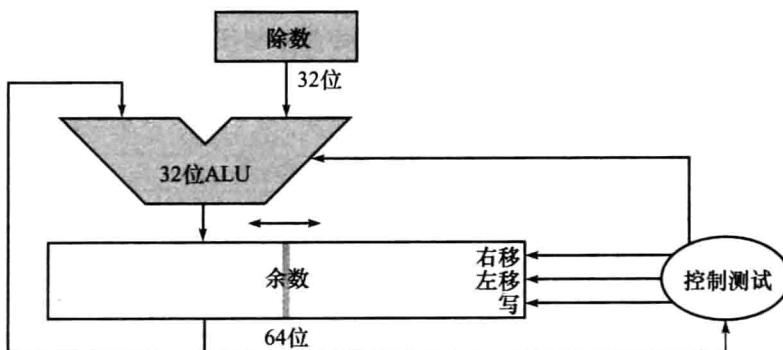


图 3-11 除法器的一种改进版本。除数寄存器、ALU、商寄存器都是 32 位，只有余数寄存器为 64 位。同图 3-8 相比，ALU 和除数寄存器都是位宽减半，余数进行左移。这个结构将商寄存器和余数寄存器的右半部分进行了拼接（正如图 3-5 中的那样，余数寄存器应该是 65 位从而保证加法器的进位不会丢失。）□

### 3.4.2 有符号除法

到目前为止，我们一直忽略有符号数的除法。最简单的办法是记住除数和被除数的符号。  
如果两者的符号相异，则商为负。

**192** **精解** 有符号除法一个比较麻烦的地方是必须设置余数的符号。记住，下面的公式必须满足：

$$\text{被除数} = \text{商} \times \text{除数} + \text{余数}$$

为了理解如何设置余数的符号，我们来看  $\pm 7_{10}$  除以  $\pm 2_{10}$  这个例子的各种情况。第一种情况很简单：

$$( +7 ) \div ( +2 ) : \text{商} = +3, \text{ 余数} = +1$$

检查结果：

$$+7 = 3 \times 2 + (+1) = 6 + 1$$

如果我们改变被除数的符号，商就会改变：

$$(-7) \div (+2) : \text{商} = -3$$

重写基本公式来计算余数：

$$\text{余数} = (\text{被除数} - \text{商} \times \text{除数}) = (-7) - [-3 \times (+2)] = (-7) - (-6) = -1$$

从而，

$$(-7) \div (+2) : \text{商} = -3, \text{ 余数} = -1$$

再次检查结果：

$$-7 = (-3) \times 2 + (-1) = -6 - 1$$

商是  $-4$  且余数是  $+1$  同样满足基本公式，但不能取这个结果，其原因是如果那样，商的绝对值将会根据被除数和除数的符号而改变！很明显，如果

$$-(x \div y) \neq (-x) \div y$$

编程将会面临更大的挑战。保持被除数的符号和余数的符号相同，而不管除数和商的符号如何，就可以避免这种异常的情况。

我们采用相同的规则计算其他情况：

$$( +7 ) \div ( -2 ) : \text{商} = -3, \text{ 余数} = +1$$

$$(-7) \div (-2) : \text{商} = +3, \text{ 余数} = -1$$

因此，正确的有符号除法算法在源操作数的符号相反时商为负，同时使非零余数的符号和被除数的相同。

### 3.4.3 更快速的除法

与乘法相同，摩尔定律同样适用于除法。我们使用许多加法器来加速乘法，但这一招对除法却不管用。因为除法算法每次迭代前需要知道减法结果的符号，而乘法却可以立刻生成 32 个部分积。

有一些技术可以每步生成不仅一个商位。如被称为 SRT 的除法算法，通过查找表的方法来尝试猜测每步几个商位，其中查找表基于被除数和余数的高位部分来进行。它依赖后面的步骤来修正错误的猜测。如今典型值是 4 位。算法的关键是猜测要减的值。对于二进制算法，只有一种选择。可用余数的 6 位和除数的 4 位来索引查找表，从而决定每步的猜测。

这个快速算法的正确性取决于查找表中的值是否合适。在 3.9 节给出了如果查找表不正确将会出现的情况。

### 3.4.4 MIPS 中的除法

你可能已经注意到图 3-5 和图 3-11 中相同的顺序执行硬件结构既可以做乘法，又可以做除法。唯一需要的是一个 64 位的可左右移位的寄存器和一个能做加减法的 32 位宽的 ALU。因此，MIPS 用 32 位的 Hi 和 32 位的 Lo 寄存器来处理乘法和除法。

我们从上面的算法中可能已经猜出，在除法指令执行完后，Hi 存放着余数，Lo 存放着商。

为了处理有符号整数和无符号整数，MIPS 采用两条指令：除（div）和无符号除（divu）。MIPS 汇编器允许除指令使用三个寄存器，且采用 mflo 和 mfhi 指令将运算结果放入指定的通用寄存器。

### 3.4.5 小结

乘法和除法共用硬件的方案允许 MIPS 提供一对单独的 32 位寄存器来支持乘法和除法运算，可以通过预测多位商的方法来加速除法运算，在预测错误时及时进行恢复。图 3-12 汇总了前面两节中 MIPS 体系结构的优化处理。

194

MIPS 汇编语言

类别	指令	例题	含义	备注
算术运算	加	add \$s1,\$s2,\$s3	\$s1 = \$s2 + \$s3	三个操作数；检测溢出
	减	sub \$s1,\$s2,\$s3	\$s1 = \$s2 - \$s3	三个操作数；检测溢出
	加立即数	addi \$s1,\$s2,100	\$s1 = \$s2 + 100	加常数；检测溢出
	无符号加	addu \$s1,\$s2,\$s3	\$s1 = \$s2 + \$s3	三个操作数；不检测溢出
	无符号减	subu \$s1,\$s2,\$s3	\$s1 = \$s2 - \$s3	三个操作数；不检测溢出
	无符号加立即数	addiu \$s1,\$s2,100	\$s1 = \$s2 + 100	加常数；不检测溢出
	从协处理器寄存器中获得	mfc0 \$s1,\$epc	\$s1 = \$epc	复制异常 PC 到专用寄存器
	乘	mult \$s2,\$s3	Hi,Lo = \$s2 × \$s3	64 位有符号积存在 Hi,Lo 中
	无符号乘	multu \$s2,\$s3	Hi,Lo = \$s2 × \$s3	64 位无符号积存在 Hi,Lo 中
	除	div \$s2,\$s3	Lo = \$s2 / \$s3, Hi = \$s2 mod \$s3	Lo = 商,Hi = 余数
	无符号除	divu \$s2,\$s3	Lo = \$s2 / \$s3, Hi = \$s2 mod \$s3	无符号商和余数
	从 Hi 中获得	mfhi \$s1	\$s1 = Hi	用来获得 Hi 的复本
	从 Lo 中获得	mflo \$s1	\$s1 = Lo	用来获得 Lo 的复本
数据传输	取字	lw \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	将一个字从内存中取到寄存器中
	存字	sw \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	将一个字从寄存器中存到内存中
	取无符号半字	lhu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	将半个字从内存中取到寄存器中
	存半字	sh \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	将半个字从寄存器中存到内存中
	取无符号字节	lbu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	将一个字节从内存中取到寄存器中
	存字节	sb \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	将一个字节从寄存器中存到内存中
	取链接字	ll \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	取字作为原子交换的前半部
	存条件字	sc \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1; \$s1 = 0 或 1	存字作为原子交换的后半部
	立即数读入高 16 位	lui \$s1,100	\$s1 = 100 * 2 <sup>16</sup>	取立即数并放在高 16 位

图 3-12 MIPS 核心体系结构。为了节省篇幅，没有给出 MIPS 体系结构的存储器和寄存器，但增加了 Hi 和 Lo 寄存器来支持乘法和除法。在 MIPS 参考数据卡中列出了 MIPS 机器语言

195

类别	指令	举例	含义	备注
逻辑运算	与	AND \$s1,\$s2,\$s3	\$s1 = \$s2 & \$s3	三个寄存器操作数;按位与
	或	OR \$s1,\$s2,\$s3	\$s1 = \$s2   \$s3	三个寄存器操作数;按位或
	或非	NOR \$s1,\$s2,\$s3	\$s1 = ~(\$s2   \$s3)	三个寄存器操作数;按位或非
	与立即数	ANDi \$s1,\$s2,100	\$s1 = \$s2 & 100	和常数按位与
	或立即数	ORi \$s1,\$s2,100	\$s1 = \$s2   100	和常数按位或
	逻辑左移	sll \$s1,\$s2,10	\$s1 = \$s2 << 10	根据常数左移相应位
	逻辑右移	srl \$s1,\$s2,10	\$s1 = \$s2 >> 10	根据常数右移相应位
条件跳转	相等时跳转	beq \$s1,\$s2,25	if (\$s1 == \$s2) 跳至 PC + 4 + 100	相等检测;和 PC 相关的跳转
	不相等时跳转	bne \$s1,\$s2,25	if (\$s1 != \$s2) 跳至 PC + 4 + 100	不相等检测;和 PC 相关的跳转
	小于时置位	slt \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; 否则 \$s1 = 0	比较是否小于;补码形式
	小于立即数时置位	slti \$s1,\$s2,100	if (\$s2 < 100) \$s1 = 1; 否则 \$s1 = 0	比较是否小于常数;补码形式
	无符号数比较小于时置位	sltu \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; 否则 \$s1 = 0	比较是否小于;自然数
	无符号数比较小于立即数时置位	sltiu \$s1,\$s2,100	if (\$s2 < 100) \$s1 = 1; 否则 \$s1 = 0	比较是否小于常数;自然数
	跳转	jr 2500	跳至 10000	跳转到目标地址
无条件跳转	跳转至寄存器所指的位置	jr \$ra	跳至 \$ra	用于 switch 语句,以及过程调用返回
	跳转并链接	jal 2500	\$ra = PC + 4; 跳至 10000	用于过程调用

图 3-12 (续)

**01 硬件/软件接口** MIPS 处理器除法指令忽略溢出, 所以需要软件来检测商是否溢出。除了溢出, 除法还可能产生不适当的计算: 除数为 0。一些计算机会分辨这两种异常事件。而同溢出一样, MIPS 软件必须通过检查除数来确定是否会发此情况。

**01 精解** 一种更快的算法是在余数为负时, 不需要立即将除数加回去。它只是在下一步简单地将被除数加到移位后的余数上, 因为  $(r+d) \times 2 - d = r \times 2 + d \times 2 - d = r \times 2 + d$ 。这种不恢复 (nonrestoring) 除算法每步需要一个时钟周期, 将会在练习题中给出更多的分析; 而前面介绍的算法称为恢复 (restoring) 除法。第三种算法称为不执行 (nonperforming) 除算法, 这种算法在余数为负时, 不保存减法的结果。它平均减少了三分之一的算术操作。

### 3.5 浮点运算

如果方向错了, 再快也白搭。

——美国谚语

除了有符号和无符号整数, 编程语言也支持带小数的数字, 即数学上的实数。如:

3. 14159265 $\cdots_{10}$  (pi)

2. 71828 $\cdots_{10}$  (e)

0. 000000001 $_{10}$  即  $1.0_{10} \times 10^{-9}$  (纳秒级)

3155760000 $_{10}$  即  $3.15576_{10} \times 10^9$  (一百年的秒数)

请注意, 在最后的例子中, 那个数并不是小数, 而是比 32 位的有符号整数还要大的数。

这种表达上述例子中后两个数的记数法称为科学记数法 (scientific notation)。一个采用科学记数法表示的数，若没有前导零且小数点左边只有一位整数，则可称为规格化 (normalized) 数。例如， $1.0_{10} \times 10^{-9}$  就是规格化的科学记数，但  $0.1_{10} \times 10^{-8}$  和  $10.0_{10} \times 10^{-10}$  就不是。

② 科学记数法：十进制小数点左边只有一位整数的记数法。

③ 规格化数：没有前导零的浮点记数法。

正如可以用科学记数法来表示十进制数那样，我们也可以用科学记数法来表示二进制数，如：

$$1.0_2 \times 2^{-1}$$

为了使二进制数规格化，需要定义一个基数，这个基数可用来移位使小数点左边只保留一位非零数。只有基数为 2 才满足要求。因为基数不是 10，所以我们称这时的小数点为二进制小数点 (binary point)。

这类计算机算术支持浮点数 (floating point) 的计算，称为浮点数是因为其表示的二进制小数点是不固定的，与整数相似。C 语言用 float 来表示这类数。正如科学记数那样，数被表示为二进制小数点左边只有一位非零数的形式。在二进制中，其格式为： $1. xxxxxxxx_2 \times 2^{yyy}$ 。（尽管计算机对指数也同其他数一样表示为以 2 为基的形式，但这里为了简化记数，我们用十进制来表示指数。）

④ 浮点数：二进制小数点不固定的表达数的记数法。

对实数采用规格化形式的标准科学记数法有三个优点：简化了浮点数的数据交换；简化了浮点算术算法；提高了用一个字存储的数的精度，因为无用的前导零可能占用的位被二进制小数点右边的有效位替代了。

### 3.5.1 浮点表示

浮点表示的设计者必须在尾数 (fraction) 位宽和指数 (exponent) 位宽之间找出折中的办法，因为字的大小是固定的，有一部分增加一位，则另一部分就要减少一位。折中是在精度和表示范围间进行权衡：增加小数部分会增加表示精度，而增加指数部分会增加数的表示范围。正如我们在第 2 章中所提到的设计方针讲的那样，好的设计需要好的折中。

⑤ 尾数：位于浮点数的尾数字段，其值在 0 和 1 之间。

⑥ 指数：位于浮点数的指数字段，表示小数点的位置。

浮点数通常是多个字的宽度。MIPS 中的浮点数表示如下： $s$  为浮点数的符号 (1 表示负数)，指数域为 8 位宽 (包括指数的符号位)，尾数域为 23 位宽。这种表示称为符号和数值 (sign and magnitude)，因为符号和数值的位置是相互分离的。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
$s$	指数																								尾数							
1位	8位																								23位							

一般浮点数表示为这样的形式：

$$(-1)^s \times F \times 2^E$$

$F$  为小数域的值， $E$  为指数域的值。这些域之间具体的关系后面会详细讲解。（我们将会简单地看到 MIPS 所做的稍有技巧性的改变。）

浮点数表示法使 MIPS 计算机有很大的数值表示范围，可以小到  $2.0_{10} \times 10^{-38}$ ，大到  $2.0_{10} \times$

$10^{38}$ 。但它和无穷还是不同的，所以依然有可能会因数太大而不能表示。因此，正如在整数运算中那样，溢出中断在浮点运算中也会发生。注意，这里的溢出（上溢）（overflow）表示指数太大而不能在指数域表示。

② 溢出（上溢）：正的指数太大而导致指数域放不下的情况。

浮点也会出现一种新的异常事件。正如程序员想要知道何时他们计算的数太大而不能表示那样，他们同样也想知道一个非零的小数是否会太小而不能表示；任何一种事件都会引起程序给出错误答案。为了和上溢区分开来，将其称为下溢（underflow）。下溢发生的条件是负的指数太大而不能在指数域中表示出来。

一种减少上溢和下溢的方法是采用更大指数的格式。在 C 语言中称为 double，基于 double 的操作称为双精度（double precision）浮点算术；单精度（single precision）浮点就是前面的格式。

② 下溢：负的指数太大而导致指数域放不下的情况。

② 双精度：浮点数由两个 32 位的字表示。

② 单精度：浮点数由一个 32 位的字表示。

双精度浮点数占用了两个 MIPS 字，如下所示。其中， $s$  表示符号，指数域为 11 位，尾数域为 52 位。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
$s$	指数																															
1位	11位											20位																				
尾数（续）																																

32位

MIPS 双精度的表示范围从  $2.0_{10} \times 10^{-308}$  到  $2.0_{10} \times 10^{308}$ 。尽管双精度增加了指数范围，它主要的优势还是通过提供更多的有效位数来实现更大的表示精度。

这些格式已经超出了 MIPS 体系结构。实际上它们是 IEEE 754 浮点标准的一部分，从 1980 年以来的每台计算机都遵循该标准。该标准既简化了浮点程序的接口，又提高了计算机算术的质量。

为了将更多的数据位打包到有效位数（significand）部分，IEEE 754 甚至隐藏了规格化二进制数的前导位 1。因此，在单精度下，数有 24 位宽（隐含的 1 和 23 位尾数），在双精度下为 53 位宽（1+52）。为了精确，我们用术语有效位数来表示 24 位或者 53 位的数，就是隐含 1 加尾数。<sup>②</sup>因为 0 没有前导位 1，它的指数保留为 0，所以硬件就不会将前导位 1 加到尾数上面。

因此  $00\cdots 00_2$  代表 0；其他数的表示依然采用前面的形式，就是加上了隐含 1：

$$(-1)^s \times (1 + F) \times 2^E$$

其中， $F$  表示的是 0 和 1 之间的数， $E$  表示的是指数域中的值。如果我们从左到右标记小数为  $s_1, s_2, s_3, \dots$ ，则数的值为

$$(-1)^s \times [1 + (s_1 \times 2^{-1}) + (s_2 \times 2^{-2}) + (s_3 \times 2^{-3}) + (s_4 \times 2^{-4}) + \dots] \times 2^E$$

图 3-13 给出了 IEEE 754 浮点数的编码。IEEE 754 标准的其他特点是用特殊的符号来表示异常事件。例如，软件可以将结果设置成某种格式来表示  $+\infty$  或者  $-\infty$ ，以替代除 0 中断；最大的指数保留下来标识那些特殊符号。当程序员打印结果时，程序会打印出一个无穷符号。（对于有数学训练的人，无穷的目的是形成实数的拓扑闭集。）

② 由于“有效位”和“尾数”相比，“有效位”多了一位“1”，而在浮点运算中，常用“尾数”的术语，因此在后面多处“尾数”即表示“有效位”。——译者注

单精度		双精度		表示对象
指数	尾数	指数	尾数	
0	0	0	0	0
0	非0	0	非0	$\pm$ 非规格化数
1-254	任何值	1-2046	任何值	$\pm$ 浮点数
255	0	2047	0	$\pm$ 无穷
255	非0	2047	非0	NaN (非数, 即不是数)

图 3-13 IEEE 754 浮点数的编码。用一个单独的符号位来决定正负。在 3.5 节的精解中描述了非规格化数。这个信息也可以在 MIPS 参考数据卡的第 4 列中找到。

IEEE 754 甚至给出了一种表示无效操作结果（如  $0/0$  或者无穷减无穷）的符号——NaN（Not a Number），即非数的意思。设立 NaN 的目的是让程序员推迟程序中的一些测试和决定，等到方便的时候再进行。

IEEE 754 的设计者还希望浮点表示能够比较容易处理整数比较，特别是排序的时候。这就是为什么符号放在最高位的原因，这样就可以快速地测试出小于、大于、等于 0 的情况。（比起简单的整数分类，它稍显复杂，因为这种记数法本质上是符号和数值的形式，而不是补码形式。）

将指数放在有效数前也能简化用整数比较指令来处理的浮点数分类，因为在有着相同符号的情况下，指数大的数其值就大。

负的指数对简化分类形成一个挑战。如果我们用补码或者其他记数法，可能会使负指数的高位为1，从而使一个负指数显得是一个大数了。例如， $1.0_2 \times 2^{-1}$ 表示如下：

199

(需要注意的是, 尾数中隐含前导 1。) 而数  $1.0_2 \times 2^{+1}$  看起来似乎是一个较小的二进制数。

因此希望记数法能将最小的负指数表示为  $00\cdots 00_2$ ，而最大的正指数表示为  $11\cdots 11_2$ 。这种记数法称为带偏阶的记数法 (biased notation)。需要从带偏阶的指数中减去偏阶，才能获得真实的值。

IEEE 754 规定单精度的偏阶为 127，所以指数为 -1，则会表示为  $-1 + 127_{10}$ ，即  $126_{10} = 0111\ 1110_2$ ，而 +1 表示为  $1 + 127$ ，即  $128_{10} = 1000\ 0000_2$ 。双精度的指数偏阶为 1023。给指数带偏阶后，浮点数表示为

$$(-1)^s \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

从而，单精度数的表示范围从

$\pm 1,000,000,000,000,000,000,000,000,00, \times 2^{-126}$

到

$$\pm 1.111\,111\,111\,111\,111\,111\,111\,111\,111_2 \times 2^{+127}$$

让我们演示一下浮点表示。

200

## 01 例题·浮点表示

演示用 IEEE 754 的单精度和双精度格式来表示  $-0.75_{10}$ 。

**01 答案**

-0.75<sub>10</sub>也可表示为

$$-3/4_{10} \quad \text{或} \quad -3/2^2_{10}$$

它的二进制小数形式为

$$-11_2 / 2_{10}^2 \quad \text{或} \quad -0.11_2$$

用科学记数表示的形式为

$$= 0.11_2 \times 2^0$$

采用规格化的科学记数，为

$$= 1.1_2 \times 2^{-1}$$

单精度的通用表达式为

$$(-1)^s \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - 127)}$$

将  $-1.1_2 \times 2^{-1}$  的指数减去 127，得到

$$(-1)^1 \times (1 + 0.1000\ 0000\ 0000\ 0000\ 0000_2) \times 2^{(126-127)}$$

所以  $-0.75_{10}$  的单精度二进制格式为

30 29 28 27 26 25 24 23 22 21 20 19 18

1位 8位

8位 23位

双精度表示为

1位 11位

20位

让我们再看反方向的一个例子。

## 01 例题·二进制转十进制浮点

十进制数如何用单精度浮点表示？

01 答案

符号位为 1，指数域的值为 129，尾数域的值为  $1 \times 2^{-2} = 1/4$ ，即 0.25。使用基本公式，

$$(-1)^s \times (1 + \text{Fraction}) \times 2^{(\text{Exponent-Bias})} = (-1)^1 \times (1 + 0.25) \times 2^{(129 - 127)} \\ = -1 \times 1.25 \times 2^2 \\ = -1.25 \times 4 = -5.0$$

在下面的部分，我们将给出浮点加法和乘法的算法。其核心部分是对尾数进行相应的整数操作，但也需要额外的工作去处理指数部分并对结果进行规格化。我们先给出直观上的十进制算法，然后在图中给出有更多细节的二进制版本。

**01 精解** 遵循 IEEE 准则，在标准发布 20 年后，IEEE 754 委员会依据需求对标准进行了更新。更新后的 IEEE 754-2008 标准中包含了 IEEE 754-1985 标准中几乎全部内容，并且增加了 16 位（半精度）和 128 位（四精度）。到目前为止还没有硬件支持四精度，但日后必会出现。更新后的标准也支持十进制浮点运算，这在 IBM 大型机中已经实现。

**01 精解** 为了在保持尾数位宽不变的情况下增大表示范围，一些早于 IEEE 754 标准的计算机采用了大于 2 的基数。例如，IBM 360 和 370 大型计算机以 16 为基数。因此每当 IBM 机的指数改变 1，尾数就将移 4 位，所以基为 16 的规格化数的前导零可能会多达 3 个！也就意味着有 3 个有效位要从尾数中去掉，从而在浮点算术精度上产生较大的问题。最近 IBM 大型机不但支持十六进制模式也开始支持 IEEE 754 标准。

□

202

### 3.5.2 浮点加法

为了说明浮点加法中的问题，我们将用科学记数法表示的两个数相加： $9.999_{10} \times 10^1 + 1.610_{10} \times 10^{-1}$ 。假设我们只能存储 4 个十进制有效数和两个十进制指数。

步骤 1：为了能让两数相加，我们必须将有较小指数的数向有较大指数的数对齐。

因此，需要将指数较小  $1.610_{10} \times 10^{-1}$  的向指数大的数对齐。由于一个非规格化的浮点数可以有多种科学计数法的表示形式，可以利用这一特性完成指数的对齐。

$$1.610_{10} \times 10^{-1} = 0.1610_{10} \times 10^0 = 0.01610_{10} \times 10^1$$

最右边的表示形式是我们所需要的，因为它和较大的数  $9.999_{10} \times 10^1$  的指数相同。因此第一步要向右移动较小数的有效数，使其指数和较大学的指数相同。由于我们只能表示 4 位十进制数，所以，移位后得到的数为

$$0.016_{10} \times 10^1$$

步骤 2：将有效数相加：

$$\begin{array}{r} 9.999_{10} \\ + 0.016_{10} \\ \hline 10.015_{10} \end{array}$$

和为  $10.015_{10} \times 10^1$ 。

步骤 3：因为和不是规格化科学记数的形式，所以我们需要调整：

$$10.015_{10} \times 10^1 = 1.0015_{10} \times 10^2$$

因此，在加法后，我们可能需要对和移位，将其变为规格化形式，同时相应地调整指数。在这个例子中是右移，但是如果一个数为正，一个数为负，则和可能会有许多前导 0，从而需要左移。无论指数是增加还是减少，我们都需要检查上溢或者下溢——我们必须保证指数能够被固定位宽的指数域所表示。

步骤 4：因为我们假设有效数只有 4 位十进制数那么长（不包括符号位），所以我们需要进行舍入。如果右边多余的数在 0 和 4 之间就舍掉，如果右边多余的数在 5 和 9 之间，则加 1。数

$$1.0015_{10} \times 10^2$$

舍入为有 4 个十进制数有效位的数

$$1.002_{10} \times 10^2$$

这是因为第四位右边的数在 5 和 9 之间。注意，如果我们不幸将 1 加到了一串 9 上，则和不能再规格化，我们需要返回到步骤 3。

图 3-14 按照这个十进制例子给出了二进制浮点加算法。步骤 1 和步骤 2 与上面例子讨论的类似：调整有较小指数的数，使其指数与有较大指数的数对齐，然后将两个数的有效数相加。步骤 3 规格化结果，并强制检查上溢和下溢。步骤 3 中上溢和下溢的检查依赖于源操作数的精度。回忆一下，指数全 0 保留下用来表示 0；指数全 1 保留下标记指定的值和超出正常浮点数范围的情况（见 3.5 节的精解）。在下面的例子中需要注意，对于单精度，最大的指数为 127，最小的指数为 -126。

203

### 01 例题·二进制浮点加法

按照图 3-14 中的算法，尝试将  $0.5_{10}$  和  $-0.4375_{10}$  用二进制相加。

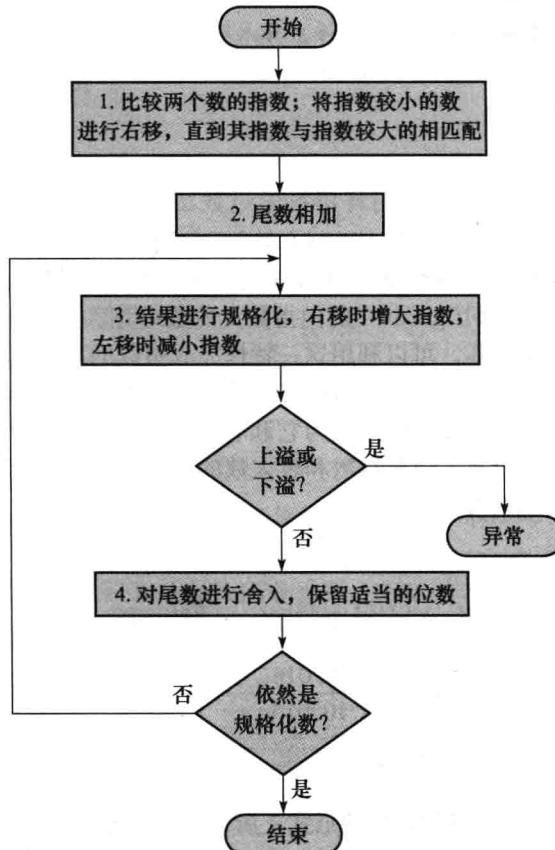


图 3-14 浮点加。正常的路径是执行一次步骤 3 和步骤 4，但如果舍入使和变为未规格化，则需要重复步骤 3

### 01 答案

首先，我们看一下这两个数用规格化科学记数法的二进制表示，这里假设我们使用 4 位精度：

$$\begin{aligned}
 0.5_{10} &= 1/2_{10} & = 1/2^1_{10} \\
 &= 0.1_2 & = 0.1_2 \times 2^0 &= 1.000_2 \times 2^{-1} \\
 -0.4375_{10} &= -7/16_{10} & = -7/2^4_{10} \\
 &= -0.1111_2 & = -0.0111_2 \times 2^0 &= -1.110_2 \times 2^{-2}
 \end{aligned}$$

现在，我们按照如下的算法执行：

步骤 1：将有最小指数的数 ( $-1.110_2 \times 2^{-2}$ ) 的有效数进行右移，直到其指数和较大数相匹配：

$$-1.110_2 \times 2^{-2} = -0.111_2 \times 2^{-1}$$

步骤 2：将有效数相加：

$$1.000_2 \times 2^{-1} + (-0.111_2 \times 2^{-1}) = 0.001_2 \times 2^{-1}$$

步骤 3：将和规格化，并检查上溢和下溢：

$$0.001_2 \times 2^{-1} = 0.010_2 \times 2^{-2} = 0.100_2 \times 2^{-3} = 1.000_2 \times 2^{-4}$$

因为  $127 \geq +4 \geq -126$ ，所以没有上溢和下溢。（带偏阶的指数为  $-4 + 127$ ，即 123，其在

最小的指数 1 和最大的未保留的带偏阶指数 254 之间。)

步骤 4：舍入和：

$$1.000_2 \times 2^{-4}$$

这个和已经是精确地用 4 位来表示了，所以不需要再做舍入。

这个和是

$$\begin{aligned} 1.000_2 \times 2^{-4} &= 0.000\ 100\ 0_2 = 0.000\ 1_2 \\ &= 1/2^4 \quad = 1/16_{10} \quad = 0.062\ 5_{10} \end{aligned}$$

这就是  $0.5_{10}$  和  $-0.437\ 5_{10}$  的和。 □

许多计算机会使用硬件来尽可能快地运行浮点操作。图 3-15 给出了浮点加的基本结构示意图。

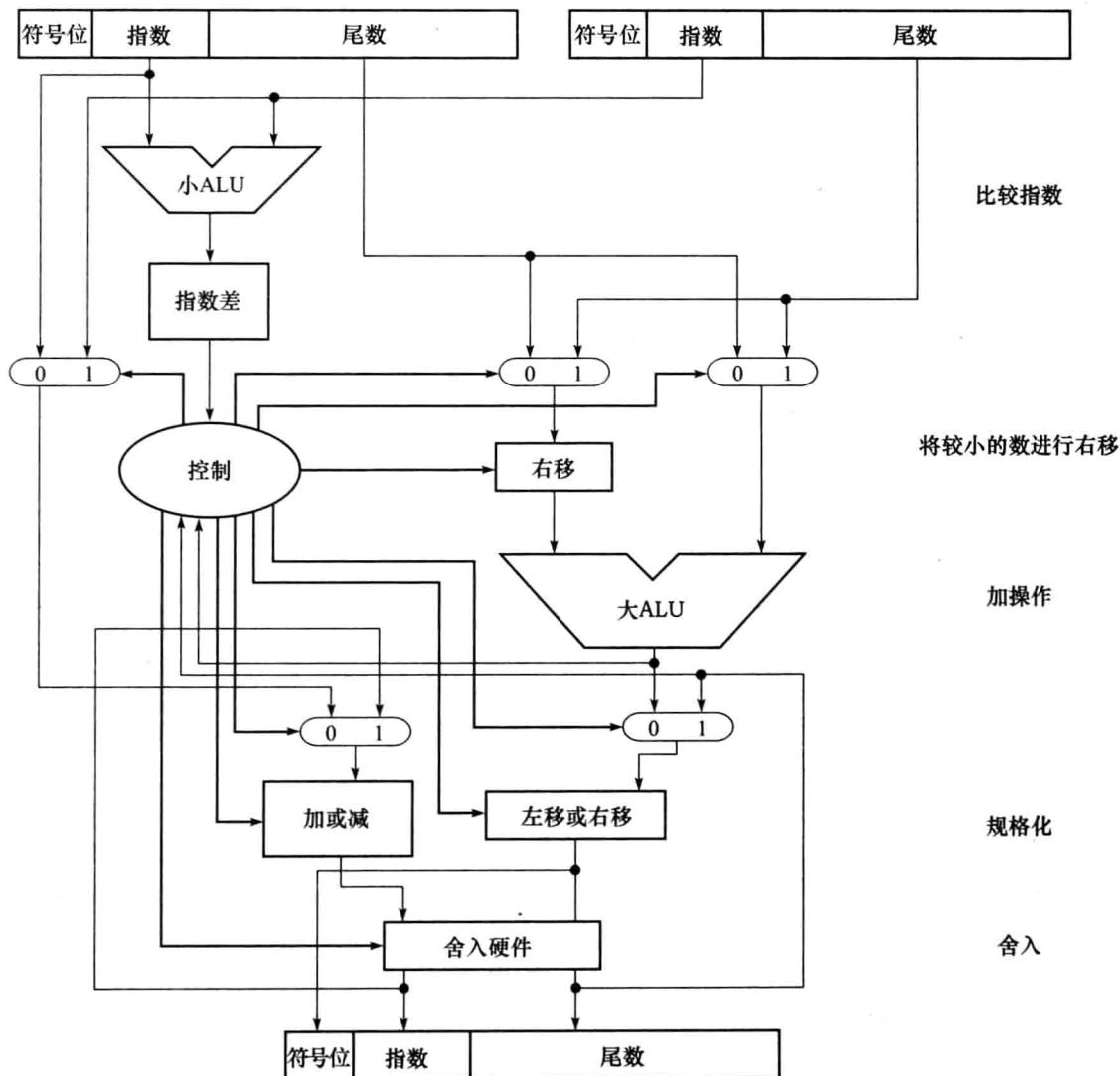


图 3-15 用于浮点加的算术单元的结构框图。图 3-14 的每步从顶向下对应到每个方框。首先，使用一个小的 ALU 将两个指数相减来决定哪个指数大及大多少。指数差将控制三个多路复用器；从左到右，选择出较大的指数、较小数的有效数和较大型的有效数。较小数的有效数通过右移后，和较大型的有效数用一个大的 ALU 相加。规格化步骤将和左移或者右移，同时增加或者减少指数。舍入产生最后的结果，这样也有可能需要再次规格化，然后产生最后的结果。

### 3.5.3 浮点乘法

首先，我们手算一个十进制乘法的例子，其中数用科学记数法表示： $1.110_{10} \times 10^{10} \times 9.200_{10} \times 10^{-5}$ 。假设我们只可以存储4位尾数和2位指数。

步骤1：不像加法，我们只是简单地将源操作数的指数相加来作为积的指数：

$$\text{新的指数} = 10 + (-5) = 5$$

现在我们处理带有偏阶的指数并要确定获得相同的结果： $10 + 127 = 137$ ，而 $-5 + 127 = 122$ ，所以

$$\text{新的指数} = 137 + 122 = 259$$

这个结果对于8位的指数域来说太大，所以肯定有什么地方出错了！问题出在偏阶上，当我们将指数相加时，也对偏阶实行了相加：

$$\text{新的指数} = (10 + 127) + (-5 + 127) = (5 + 2 \times 127) = 259$$

因此，当将带偏阶的数相加时，为了得到正确的带偏阶的和，我们需要将一个偏阶从和中减去：

$$\text{新的指数} = 137 + 122 - 127 = 259 - 127 = 132 = (5 + 127)$$

5就是我们刚开始计算的实际指数。

步骤2：下面计算有效数的乘法：

$$\begin{array}{r}
 1.110_{10} \\
 \times 9.200_{10} \\
 \hline
 0000 \\
 0000 \\
 2220 \\
 9990 \\
 \hline
 10212000_{10}
 \end{array}$$

每个源操作数十进制小数点右边都有三位，所以积的尾数的十进制小数点在从右边数第6位处：

$$10.212\ 000_{10}$$

假设我们只可以保留十进制小数点右边三位数，则积为 $10.212_{10} \times 10^5$ 。

步骤3：这个积是未规格化的，所以我们需要规格化它：

$$10.212_{10} \times 10^5 = 1.0212_{10} \times 10^6$$

因此，在乘法后，积需要右移一位来变成规格化形式，同时指数加1。此刻，我们要检查上溢和下溢。当两个源操作数都很小时——两者都有非常大的负指数时，就有可能发生下溢。

步骤4：因为之前我们假设有效数只有4位宽（不包括符号），所以我们必须对结果进行舍入。将

$$1.0212_{10} \times 10^6$$

舍入为只有4位的有效数

$$1.021_{10} \times 10^6$$

步骤5：积的符号取决于原始源操作数的符号。当它们相同时，符号为正；否则，符号为负。因此，积为

$$+1.021_{10} \times 10^6$$

在加法算法中，和的符号由有效数相加来决定，但在乘法中，积的符号由源操作数来决定。

再一次，如图3-16所示，二进制浮点乘的步骤和我们刚做完的步骤类似。首先，我们将

带偏阶的指数相加，并减去一个偏阶，获得积的指数。接着是有效数的乘法，紧跟一个可选的规格化步骤。指数的大小用来检查上溢和下溢，然后对积进行舍入。当舍入引起进一步的规格化时，我们需要再次检查指数的大小。最后，如果源操作数的符号相异，就将符号位设为1（积为负）；如果相同，设为0（积为正）。

### 01 例题·二进制浮点乘法

按照图3-16中的步骤，试计算 $0.5_{10}$ 和 $-0.4375_{10}$ 的乘积。

### 01 答案

在二进制下，也就是将 $1.000_2 \times 2^{-1}$ 和 $-1.110_2 \times 2^{-2}$ 相乘。

步骤1：将不带偏阶的指数相加：

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

或者，使用带偏阶的表达：

$$\begin{aligned} (-1 + 127) + (-2 + 127) - 127 \\ = (-1 - 2) + (127 + 127 - 127) \\ = -3 + 127 = 124 \end{aligned}$$

步骤2：将有效数相乘：

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

积是 $1.110\ 000_2 \times 2^{-3}$ ，但是我们需要保存4位，所以为 $1.110_2 \times 2^{-3}$ 。

步骤3：现在我们检查积以确保其是规格化的，然后检查指数以确定上溢和下溢是否发生。这个积已经是规格化的，并且，因为 $127 \geq -3 \geq -126$ ，所以没有上溢和下溢。（使用带偏阶的表达， $254 \geq 124 \geq 1$ ，所以指数域可以表达。）

步骤4：对积舍入没有使其发生变化：

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

步骤5：因为初始的源操作数的符号相异，所以积的符号为负。因此，积为

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

为了检查结果，将其转化为十进制：

$$-1.110_2 \times 2^{-3} = -0.001110_2 = -0.00111_2 = -7/2^5 = -7/32_{10} = -0.21875_{10}$$

而 $0.5_{10}$ 和 $-0.4375_{10}$ 的积确实是 $-0.21875_{10}$ 。

209

210

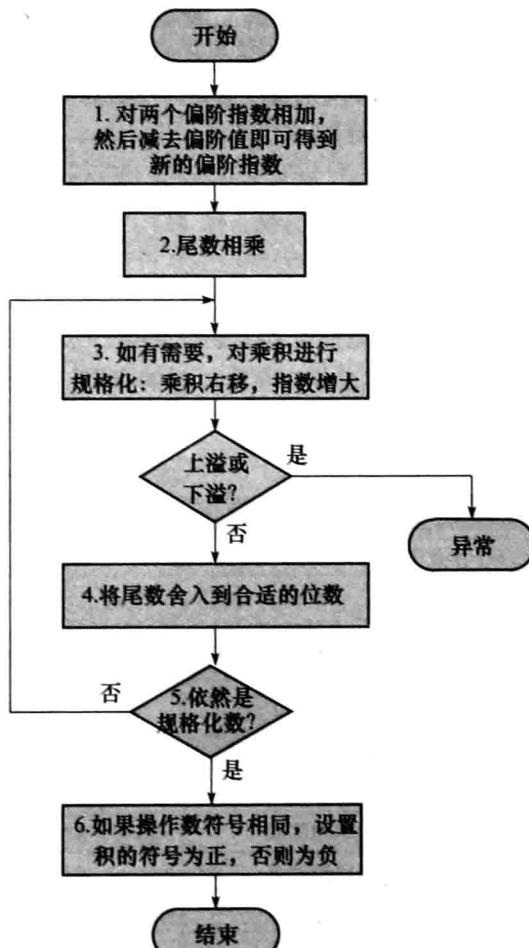


图3-16 浮点乘法。正常的路径是执行一次步骤3和步骤4，但如果舍入使积变为非规格化数，则需要重复步骤3

### 3.5.4 MIPS 中的浮点指令

MIPS有如下指令来支持IEEE 754的单精度和双精度格式：

- 浮点单精度加 (add.s) 和双精度加 (add.d)。
- 浮点单精度减 (sub.s) 和双精度减 (sub.d)。
- 浮点单精度乘 (mul.s) 和双精度乘 (mul.d)。
- 浮点单精度除 (div.s) 和双精度除 (div.d)。
- 浮点单精度比较 (c.x.s) 和双精度比较 (c.x.d)，其中，x 可能是等于 (eq)、不等于 (neq)、小于 (lt)、小于等于 (le)、大于 (gt) 或大于等于 (ge)。
- 浮点比较为真跳转 (bclt) 和浮点比较为假跳转 (bc1f)。

根据比较条件，浮点比较将比较结果设为真或者假，然后浮点跳转将比较结果作为条件决定是否跳转。

MIPS 设计增加了单独的浮点寄存器——称为 \$f0, \$f1, \$f2, …——用于单精度及双精度。因此，也有单独的针对浮点寄存器的存和取指令：lwc1 和 swc1。浮点数据传送的基寄存器仍然采用整数寄存器。从内存载入两个单精度数，将其相加，然后再将和存入内存的 MIPS 代码可能是这样的：

```
lwc1    $f4,c($sp)  # Load 32-bit F.P. number into F4
lwc1    $f6,a($sp)  # Load 32-bit F.P. number into F6
add.s   $f2,$f4,$f6  # F2 = F4 + F6 single precision
swc1    $f2,b($sp)  # Store 32-bit F.P. number from F2
```

双精度寄存器是一组单精度寄存器的偶数 - 奇数对，并使用偶数寄存器编号作为其名称。因此，一对单精度寄存器 \$f2 和 \$f3 形成一个双精度寄存器，称为 \$f2。

图 3-17 汇总了本章介绍过的 MIPS 体系结构中的浮点部分，其中为支持浮点而增加的部分用加粗标记。类似于第 2 章中的图 2-19，图 3-18 给出了这些指令的编码。

MIPS 浮点操作数				
名称	例题	备注		
32 个浮点寄存器	\$f0, \$f1, \$f2, ..., \$f31	成对地使用 MIPS 浮点寄存器来保存双精度数		
2 <sup>30</sup> 个存储字	Memory [0], Memory [4], ..., Memory [4294967292]	仅仅被数据传输指令访问。MIPS 使用字节地址，所以连续的字地址相差 4。存储器用来保存像数组这样的数据结构和在过程调用中换出的寄存器		

MIPS 浮点汇编语言				
分类	指令	例题	含义	备注
算术	浮点单精度加	add.s \$f2, \$f4, \$f6	$\$f2 = \$f4 + \$f6$	浮点加（单精度）
	浮点单精度减	sub.s \$f2, \$f4, \$f6	$\$f2 = \$f4 - \$f6$	浮点减（单精度）
	浮点单精度乘	mul.s \$f2, \$f4, \$f6	$\$f2 = \$f4 \times \$f6$	浮点乘（单精度）
	浮点单精度除	div.s \$f2, \$f4, \$f6	$\$f2 = \$f4 / \$f6$	浮点除（单精度）
	浮点双精度加	add.d \$f2, \$f4, \$f6	$\$f2 = \$f4 + \$f6$	浮点加（双精度）
	浮点双精度减	sub.d \$f2, \$f4, \$f6	$\$f2 = \$f4 - \$f6$	浮点减（双精度）
	浮点双精度乘	mul.d \$f2, \$f4, \$f6	$\$f2 = \$f4 \times \$f6$	浮点乘（双精度）
数据 传输	从内存中取字到浮点 寄存器	lwc1 \$f1,100 (\$s2)	$\$f1 = \text{存储} [\$s2 + 100]$	32 位的数据传给浮点 寄存器
	从浮点寄存器中存字 到内存	swc1 \$f1,100 (\$s2)	存储 $[\$s2 + 100] = \$f1$	32 位的数据传给存 储器

图 3-17 以前介绍过的 MIPS 浮点体系结构。附录 A.10 有更详细的介绍。这个信息也可以在 MIPS 参考数据卡的第 2 列里找到

分类	指令	例题	含义	备注
条件跳转	浮点标志真则跳转	bclt 25	如果 (cond == 1) 跳至 PC + 4 + 100	如果浮点标志为真则执行 PC 相关联的跳转
	浮点标志假则跳转	bclf 25	如果 (cond == 0) 跳至 PC + 4 + 100	如果浮点标志为假则执行 PC 相关联的跳转
	浮点单精度比较 (eq, ne, lt, le, gt, ge)	c.lt.s \$f2, \$f4	如果 (\$f2 < \$f4) 则 cond = 1；否则 cond = 0	浮点单精度比较，如果小于则置 cond
	浮点双精度比较 (eq, ne, lt, le, gt, ge)	c.lt.d \$f2, \$f4	如果 (\$f2 < \$f4) 则 cond = 1；否则 cond = 0	浮点双精度比较，如果小于则置 cond

MIPS 浮点机器语言

名称	格式	例题							备注
add.s	R	17	16	6	4	2	0	add.s \$f2, \$f4, \$f6	
sub.s	R	17	16	6	4	2	1	sub.s \$f2, \$f4, \$f6	
mul.s	R	17	16	6	4	2	2	mul.s \$f2, \$f4, \$f6	
div.s	R	17	16	6	4	2	3	div.s \$f2, \$f4, \$f6	
add.d	R	17	17	6	4	2	0	add.d \$f2, \$f4, \$f6	
sub.d	R	17	17	6	4	2	1	sub.d \$f2, \$f4, \$f6	
mul.d	R	17	17	6	4	2	2	mul.d \$f2, \$f4, \$f6	
div.d	R	17	17	6	4	2	3	div.d \$f2, \$f4, \$f6	
lwcl	I	49	20	2		100		lwcl \$f2, 100 (\$s4)	
swcl	I	57	20	2		100		swcl \$f2, 100 (\$s4)	
bclt	I	17	8	1		25		bclt 25	
bclf	I	17	8	0		25		bclf 25	
c.lt.s	R	17	16	4	2	0	60	c.lt.s \$f2, \$f4	
c.lt.d	R	17	17	4	2	0	60	c.lt.d \$f2, \$f4	
域宽		6 位	5 位	5 位	5 位	5 位	6 位	所有 MIPS 指令都是 32 位宽	

图 3-17 (续)

op(31:26):								
28 ~ 26 31 ~ 29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	<u>Rfmt</u>	<u>Bltz/gez</u>	j	jal	beq	bne	blez	bgtz
1(001)	addi	addiu	slti	sltiu	ANDi	ORi	xORi	lui
2(010)	<u>TLB</u>	<u>F1Pt</u>						
3(011)								
4(100)	lb	lh	lw	lw	lbu	lhu	lwr	
5(101)	sb	sh	swl	sw			swr	
6(110)	lwcl							
7(111)	swcl							

图 3-18 MIPS 浮点指令编码。标记是按照行和列给出域值。例如，在图的顶端部分，在第 4 行（指令的 31 ~ 29 位为 100<sub>2</sub>）和第 3 列（指令的 28 ~ 26 位为 011<sub>2</sub>）可以发现 lw，所以 op 域（31 ~ 26 位）相应的值为 100011<sub>2</sub>。带下划线意味着该域用于其他地方。例如，在第 2 行第 1 列的 F1Pt (op = 010001<sub>2</sub>) 定义在图的底端。因此，在图底部第 0 行第 1 列的 sub.f 意味着 funct 域（指令的 5 ~ 0 位）为 000001<sub>2</sub> 且 op 域（31 ~ 26 位）是 010001<sub>2</sub>。注意在图的中间给出的 5 位的 rs 域，决定了操作是单精度 (f = s, 所以 rs = 10000) 还是双精度 (f = d, 所以 rs = 10001)。类似地，指令的 16 位决定了指令 bclt.c 是测试为真 (16 位 = 1 => bclt.t) 还是为假 (16 位 = 0 => bclt.f)。加粗的指令是在第 2 章或者本章描述过的，附录 A 给出了全部指令。这个信息也可以在 MIPS 参考数据卡的第 2 列里找到。

op(31:26) = 010001(FIpt), (rt(16:16) = 0 => c=f, rt(16:16) = 1 => c=t), rs(25:21) :								
23 ~ 21 25 ~ 24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfcl		cfcl		mtcl		ctcl	
1(01)	bcl.c							
2(10)	f=单精度	f=双精度						
3(11)								

op(31:26) = 010001(FIpt), (上面的f: 10000 = 0 => f=s; 10001 = 0 => f=d), funct(5:0) :								
2~0 5~3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	add.f	sub.f	mul.f	div.f		abs.f	mov.f	neg.f
1(001)								
2(010)								
3(011)								
4(100)	cvt.s.f	cvt.d.f			cvt.w.f			
5(101)								
6(110)	c.f.f	c.un.f	c.eq.f	c.ueq.f	c.olt.f	c.ult.f	c.ole.f	c.ule.f
7(111)	c.sf.f	c.ngle.f	c.seq.f	c.ngl.f	c.lt.f	c.nge.f	c.le.f	c.ngt.f

图 3-18 (续)

**01 硬件/软件接口** 在支持浮点算术方面，体系结构设计者面临着一个问题：是否和整数指令使用相同的寄存器，或者为浮点增加一组专用的寄存器。因为程序通常对不同的数据执行整数和浮点操作，独立的寄存器会略微增加程序中要执行的指令数目。主要的影响是需要建立一组独立的指令用于浮点寄存器和内存之间数据的传输。

单独的浮点寄存器的好处是倍增了寄存器数目而不需要在指令格式中增加更多的位数，同时因为使用了相互独立的整数和浮点寄存器堆而倍增了寄存器带宽，另外，还可以量身定做针对浮点的寄存器；例如，一些计算机将寄存器中各种大小的源操作数转化为一种单一的内部格式。

### 01 例题·将浮点 C 程序编译为 MIPS 汇编代码

将华氏温度转为摄氏温度：

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

假设浮点变量 fahr 存放在 \$f12 中，结果存放在 \$f0 中。（不像整数寄存器，浮点寄存器也可以存储数据。）那么 MIPS 汇编代码是什么？

### 01 答案

我们假设编译器将三个浮点常数放置在内存中，并且可以用全局指针 \$gp 很容易地获得。首先前两个取数指令将常数 5.0 和 9.0 载入浮点寄存器：

```
f2c:
    lwcl $f16,const5($gp) # $f16 = 5.0 (5.0 in memory)
    lwcl $f18,const9($gp) # $f18 = 9.0 (9.0 in memory)
```

然后相除得到分数 5.0/9.0：

```
div.s $f16, $f16, $f18 # $f16 = 5.0 / 9.0
```

(许多编译器在编译的时候就做了 5.0 除以 9.0 的操作，并将单精度常数 5.0/9.0 存入内存，从而在运行的时候避免做除法。) 下面，我们将常数 32.0 载入，然后将其从 `fahr($f12)` 中减去：

```
lwc1 $f18, const32($gp) # $f18 = 32.0
sub.s $f18, $f12, $f18 # $f18 = fahr - 32.0
```

最后，我们将两个中间结果相乘，积作为返回结果放在 `$f0` 中，然后程序返回

```
mul.s $f0, $f16, $f18 # $f0 = (5/9)*(fahr - 32.0)
jr $ra                 # return
```

现在，让我们做浮点矩阵操作，其代码在科学计算程序中是常见的。 □

### 01 例题·将二维矩阵的浮点 C 程序编译为 MIPS

许多浮点计算都采用双精度。现在做矩阵乘法  $C = C + A * B$ ，这通常称为双精度通用矩阵乘法（简称 DGEMM）。我们将在 3.8 节以及后面的第 4、5、6 章中都会看到该版本。假定  $A$ 、 $B$ 、 $C$  都是  $32 \times 32$  的矩阵。

```
void mm (double c[][][], double a[][][], double b[][][])
{
    int 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];
}
```

数组的开始地址都是参数，存在 `$a0`、`$a1`、`$a2` 中。假设整数变量分别存在 `$s0`、`$s1`、`$s2` 中。这段程序的 MIPS 汇编代码是什么？

### 01 答案

注意到  $c[i][j]$  处于上面循环的最里面。因为循环变量是  $k$ ，不影响  $c[i][j]$ ，所以我们可以避免在每次迭代时载入和存储  $c[i][j]$ 。相反，编译器每次在循环外将  $c[i][j]$  载入一个寄存器，然后将  $a[i][k]$  和  $b[k][j]$  的积累加到这个寄存器里，在最里层的循环结束后将和存入  $c[i][j]$ 。

为了保持代码简洁，我们使用汇编语言的伪指令 `li`（其将一个常数载入一个寄存器）、`l.d` 和 `s.d`（汇编器将其变为一对数据传送指令、`lwc1` 和 `swc1`，向一对浮点寄存器传送数据）。

程序段首先将循环结束值 32 存入一个临时寄存器中，然后初始化循环变量：

```
mm:...
    li      $t1, 32  # $t1 = 32 (row size/loop end)
    li      $s0, 0   # i = 0; initialize 1st for loop
L1:   li      $s1, 0   # j = 0; restart 2nd for loop
L2:   li      $s2, 0   # k = 0; restart 3rd for loop
```

要计算  $c[i][j]$  的地址，我们首先要知道一个  $32 \times 32$  的二维矩阵是如何在内存中存储的。正如你所料，它的排布如同 32 个有 32 个元素的一维矩阵。所以获得我们需要的元素的第一步是跳过第  $i$  个“一维矩阵”或者第  $i$  行。因此，我们用首维的索引乘以行的大小，32。因为 32 是以 2 为底的指数值，所以我们可以用移位来替代：

```
sll  $t2, $s0, 5      # $t2 = i * 2^5 (size of row of c)
```

现在我们加上第二维的索引来获得我们想要的那行的第  $j$  个元素：

```
addu $t2, $t2, $s1  # $t2 = i * size(row) + j
```

为了将这个和转化为按字节的索引，我们给它乘上矩阵元素所占的字节大小。因为每个元素都是双精度的，所以占了 8 字节，我们用左移 3 位来代替：

```
sll  $t2, $t2, 3      # $t2 = byte offset of [i][j]
```

下面我们将这个和加上  $c$  的基地址，得到  $c[i][j]$  的地址，然后将双精度数  $c[i][j]$  载入 `$f4` 寄存器中：

```
addu $t2, $a0, $t2    # $t2 = byte address of c[i][j]
1.d $f4, 0($t2)      # $f4 = 8 bytes of c[i][j]
```

接着的 5 条指令类似于刚才的 5 条：计算双精度数  $b[k][j]$  的地址，然后将其载入。

```
L3: sll $t0, $s2, 5    # $t0 = k *  $2^5$  (size of row of b)
    addu $t0, $t0, $s1 # $t0 = k * size(row) + j
    sll $t0, $t0, 3    # $t0 = byte offset of [k][j]
    addu $t0, $a2, $t0 # $t0 = byte address of b[k][j]
    1.d $f16, 0($t0)   # $f16 = 8 bytes of b[k][j]
```

类似地，下面的 5 条指令像刚才的 5 条一样：计算双精度数  $a[i][k]$  的地址，然后将其

**216** 载入。

```
sll      $t0, $s0, 5    # $t0 = i *  $2^5$  (size of row of a)
addu    $t0, $t0, $s2 # $t0 = i * size(row) + k
sll      $t0, $t0, 3    # $t0 = byte offset of [i][k]
addu    $t0, $a1, $t0 # $t0 = byte address of a[i][k]
1.d     $f18, 0($t0)   # $f18 = 8 bytes of a[i][k]
```

现在已经载入了所有的数据，我们终于可以做一些浮点操作了！我们将分别存在  $\$f18$  和  $\$f16$  中的  $a$ 、 $b$  的元素相乘，然后累加到  $\$f4$  中。

```
mul.d $f16, $f18, $f16 # $f16 = a[i][k] * b[k][j]
add.d $f4, $f4, $f16   # f4 = c[i][j] + a[i][k] * b[k][j]
```

最后的部分将循环变量  $k$  加 1，如果索引值没到 32，则再次返回循环。如果到了 32，则结束最里层的循环，将放在  $\$f4$  中的累加和存入  $c[i][j]$ 。

```
addiu $s2, $s2, 1      # $k = k + 1
bne   $s2, $t1, L3     # if (k != 32) go to L3
s.d   $f4, 0($t2)      # c[i][j] = $f4
```

类似地，最后 4 条指令增加中间和最外层的循环变量，如果没有到 32 则返回循环，否则在到达 32 后退出循环。

```
addiu $s1, $s1, 1      # $j = j + 1
bne   $s1, $t1, L2     # if (j != 32) go to L2
addiu $s0, $s0, 1      # $i = i + 1
bne   $s0, $t1, L1     # if (i != 32) go to L1
...
```

后面的图 3-22 给出了 DGEMM 的 x86 汇编语言代码，该版本与图 3-21 中的 DGEMM 版本略有不同。□

**01 精解** 上面例子中的阵列排布，称为行主序列，用于许多 C 和其他编程语言中。但 Fortran 采用的是列主序列，即阵列是一列一列地存储。

**01 精解** 32 个 MIPS 浮点寄存器中，只有 16 个能用于双精度操作： $\$f0, \$f2, \$f4, \dots, \$f30$ 。计算中，双精度使用了成对的单精度寄存器。奇数编号的浮点寄存器只是载入和存储 64 位浮点数的右半部分。MIPS-32 给指令集增加了 1.d 和 s.d 指令。MIPS-32 也为所有浮点指令增加了“单精度配对”（paired single）版本，这里每个单指令能够对两个 32 位的源操作数并行执行浮点操作，这两个 32 位的源操作数存在 64 位的寄存器中（见 3.6 节）。例如，add.ps  $\$f0, \$f2, \$f4$  等价于 add.s  $\$f0, \$f2, \$f4$  和 add.s  $\$f1, \$f3, \$f5$ 。

**01 精解** 将整数和浮点寄存器分开的另外一个原因是在 20 世纪 80 年代的处理器还没有足够的晶体管将浮点单元和整数单元放在同一个芯片上。因此，浮点单元，包括浮点寄存器，只是一个备选的辅助芯片。这个可选的加速芯片称为协处理器。按首字母缩写的 MIPS 的浮点 load 指令 lwc1 的意思是载入一个字到协处理器 1，浮点单元。（协处理器 0 处理虚拟内存，

**217**

第 5 章对其进行描述。) 自 20 世纪 90 年代早期, 微处理器已经将浮点单元和其他功能单元集成在一个芯片上。因此, 由加速器和内置存储器组成的协处理器的术语已经过时了。

**01 精解** 正如 3.4 节提到的, 加速除法比乘法更有挑战性。除了 SRT, 还有一种利用快速乘法器的技术, 称为牛顿迭代, 它将除法变换为通过寻找函数的零点来求倒数  $1/c$ , 然后将其乘以另一源操作数。如果不计算更多的位, 迭代技术是无法进行正确舍入的。如 TI 的一款芯片通过计算倒数更多有效位的方法来解决这一问题。

**01 精解** Java 在定义浮点数据类型和操作时遵循 IEEE 754 标准。因此, 可以更好地生成第一个例子中的代码, 是一种经典的将华氏温度转换为摄氏温度的方法。

第二个例子里使用了多维矩阵, 不被 Java 显式支持。Java 允许在数组中嵌套数组, 但是不支持像 C 中的多维矩阵, 每个数组可能有自己的长度。像第 2 章中的那些例子, 第二个例子的 Java 版本需要大量的代码来进行数组的边界检查, 包括在行访问后对新的长度进行计算。它可能还需要检查对象引用是否非空。

### 3.5.5 算术精确性

与整数可以精确地表示在最大数和最小数之间的所有数不同, 浮点数通常是一个无法表示的数的近似。原因是, 假定在 0 和 1 之间, 实数就有无穷多个, 而双精度最多可以精确表示  $2^{53}$  个。我们能做到最好的就是给出最接近实际数的浮点表示。因此, IEEE 754 提供了几种舍入模式来供程序员选择他们想要的近似策略。

- **保护位:** 在浮点数中间计算中, 在右边多保留的两位中的首位; 用于提高舍入精度。
- **舍入位:** 在浮点数中间计算中, 在右边多保留的两位中的第二位; 使浮点中间结果满足浮点格式, 得到最接近的数。

舍入听起来很简单, 但它需要硬件支持在计算中产生更多的有效位。在前面的例子中, 我们对中间结果占有多少位未做介绍, 但很明显的是, 如果每个中间结果都截短成准确的位数, 就没法做舍入了。IEEE 754 因此在中间计算中, 右边总是多保留两位, 分别称为 **保护位** (guard) 和 **舍入位** (round)。我们用一个十进制的例子来说明它们的作用。

218

**01 例题·使用保护位来舍入**

将  $2.56_{10} \times 10^0$  和  $2.34_{10} \times 10^2$  相加, 假设我们有 3 位十进制尾数。首先使用保护位和舍入位将其舍入到只有三位尾数的最近数, 然后不用保护位和舍入位再做一次 (舍入)。

**01 答案**

首先我们右移较小数以对齐指数, 所以  $2.56_{10} \times 10^0$  变为  $0.0256_{10} \times 10^2$ 。因为有了保护位和舍入位, 所以当我们对齐指数时可以表示两个最低位。保护位为 5 而舍入位为 6。求和:

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

因此, 和为  $2.3656_{10} \times 10^2$ 。因为需要舍入掉两位, 所以我们需要以 50 为分水岭, 在其值为 0~49 之间时舍掉, 在 51~99 之间时向上舍入。向上舍入这个和, 变为  $2.37_{10} \times 10^2$ 。

在计算中, 在没有保护位和舍入位的情况下舍入掉两位。新的和为:

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

答案是  $2.36_{10} \times 10^2$ , 比上面的结果在最低位上少 1。 □

因为最糟糕的情况是实际的数在两个浮点表示的中间, 浮点的精确性通常是用尾数的最低位上有多少位的误差来衡量。这种衡量称为尾数最低位 (unit in the last place) 的数目, 即 ulp。如果一个数在最低位上少 2, 则称其少了 2 个 ulp。在没有上溢、下溢或无效操作异常的情况下, IEEE 754 保证了计算机使用的数的误差都在半个 ulp 以内。

- ② 尾数最低位: 在实际数和能表达的数之间的有效数最低位上的误差位数。

**01 精解** 尽管上面的例子实际只多需要一位, 但乘法需要两位。一个二进制乘积可能有一位前导 0; 因此, 规格化步骤必须将积左移一位。这个移位会将保护位移入积的最低位, 留下舍入位来精确地舍入乘积。

IEEE 754 有四种舍入模式: 总是向上舍入 (向  $+\infty$ ), 总是向下舍入 (向  $-\infty$ ), 截断舍入, 向最靠近的偶数舍入。最后一种模式给出了当数值在中间时如何去做。美国国税局 (IRS) 也许为了自身的利益, 总是将 0.50 美元向上舍入。一种更公平的办法是: 一半时间里使用向上舍入, 另一半时间里使用向下舍入。IEEE 754 处理这种中间情况的方法是如果最后一位是奇数, 就加 1; 如果是偶数, 则截去。这种方法总是在中间情况下将最低位设为 0, 正如舍入模式的名称。这种模式是用得最多的, 而且是唯一被 Java 支持的模式。

使用额外的舍入位的目的是让计算机获得相同的结果, 就如同是先以无穷的精度计算中间结果, 然后执行舍入那样。为了支持这个目标并向最靠近的偶数舍入, IEEE 754 标准在保护位和舍入位之后还有一位粘贴位 (sticky bit); 当舍入位右边的数非零时将它置 1。粘贴位可以让计算机在舍入时, 能够区分  $0.50\cdots 00_{10}$  和  $0.50\cdots 01_{10}$ 。

粘贴位可能被置 1, 例如, 在加法中, 当较小数右移时就可能这样。假设在前面的例子里我们将  $5.01_{10} \times 10^{-1}$  和  $2.34_{10} \times 10^2$  相加。即使有保护位和舍入位, 我们将 0.0050 和 2.34 相加, 得到 2.3450。粘贴位会被置 1, 因为右边是非零的。假设没有粘贴位来记住是否有 1 被移走, 我们会假设这个数等于 2.345000…00, 然后向最靠近的偶数舍入得到 2.34。使用粘贴位记住这个数是大于 2.345000…00 的, 我们舍入后会得到 2.35。

- ② 粘贴位: 同保护位和舍入位一样用于舍入的位, 当舍入位右边有非零的数据时将其置 1。

**01 精解** PowerPC、SPARC64、AMD SSE5 和 Intel AVX 体系结构提供了一个单独的指令来对三个寄存器执行乘法和加法操作:  $a = a + (b \times c)$ 。很明显, 因为这个操作常用, 所以这条指令潜在地允许更高的浮点性能。同样重要的是替换掉了两次舍入——在乘法后和在加法后——其可能在分开的指令中出现, 乘加指令只是在加法后执行一次舍入。一次舍入步骤增加了乘加的精度。这样的一次舍入的操作称为混合乘加 (fused multiply add)。它已被加入修订的 IEEE 754-2008 标准里 (见 3.11 节)。

- ② 混合乘加: 一条浮点指令, 其执行一次乘法和一次加法, 但只在加法后执行一次舍入。

### 3.5.6 小结

下面的重点再次强调了第 2 章中存储程序的概念; 不能仅仅看看数据位就决定信息的含义, 因为即使是相同的位也代表了不同的目标。这一节给出的计算机算术是有限精度的, 因此

和自然的算术不同。例如，IEEE 754 的标准浮点表示为

$$(-1)^s \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

几乎总是一个实数的近似。计算机系统必须小心弥合计算机算术和真实世界的算术之间的差距，而程序员有时也需要小心这种近似值的含义。

**01 重点** 位模式并没有内在的含义，它们可能表示有符号整数、无符号整数、浮点数和指令等。具体代表什么意思要看指令对该字的哪些位进行操作。

计算机数和真实世界里的数的主要不同是计算机数的大小是有限制的，因此限制了其精度；计算的数字有可能太大或太小而无法在一个字中表示。程序员必须记住这些限制并相应地编程。

220

C 类型	Java 类型	数据传送	操作
int	int	lw, sw, lui	addu, addiu, subu, mult, div, AND, ANDi, OR, ORi, NOR, slt, slti
Unsigned int	—	lw, sw, lui	addu, addiu, subu, multu, divu, AND, ANDi, OR, ORi, NOR, sltu, sltiu
char	—	lb, sb, lui	add, addi, sub, mult, div, AND, ANDi, OR, ORi, NOR, slt, slti
—	char	lh, sh, lui	addu, addiu, subu, multu, divu, AND, ANDi, OR, ORi, NOR, sltu, sltiu
float	float	lwcl, swcl	add.s, sub.s, mult.s, div.s, c.eq.s, c.lt.s, c.le.s
double	double	l.d, s.d	add.d, sub.d, mult.d, div.d, c.eq.d, c.lt.d, c.le.d

**01 硬件/软件接口** 在上一章，我们提出了编程语言 C 的存储分类（见 2.7 节的硬件/软件接口部分）。上表给出了一些 C 和 Java 的数据类型、MIPS 数据传送指令，以及对出现在第 2 章和本章的那些数据类型的操作指令。注意 Java 省略了无符号整数。

### 01 小测验

假设有一个 16 位的 IEEE 754-2008 浮点格式，其中有 5 位指数位。那么它可能表示的数的范围是多少？

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

**01 精解** 为了进行可能包含 NaN 的比较，IEEE 754 标准包含了有序和无序作为比较的选项。因此，完整的 MIPS 指令集有许多用于比较的指令来支持 NaN。（Java 不支持无序比较。）

为了从一次浮点操作中最大限度地获得精度，标准允许一些数以非规格化的形式出现。标准允许有非规格化数（也称为非规格化或者亚规格化），目的是使 0 和最小规格化数之间的间隙更小。在指数为零而有效数非零时，允许一个有效数逐步变小直到 0，称为逐步下溢（gradual underflow）。例如，最小的正的单精度规格化数为

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

而最小的单精度非规格化数为

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

对于双精度，非规格化间隙为从  $1.0_2 \times 2^{-1022}$  到  $1.0_2 \times 2^{-1074}$ 。

221

对于想建立一个快速浮点单元的设计者来说，可能偶尔出现的非规格化源操作数是一件令人头疼的事情。因此，许多计算机在源操作数为非规格化数时产生异常，让软件来处理相应的操作。尽管软件执行可以完美地处理，但它们低效的表现降低了非规格化数在可移植的浮点软件中的受欢迎程度。再者，如果程序员并不期望得到非规格化数，他们所写程序的执行效率之低也许会令他们感到惊讶。

### 3.6 并行性和计算机算术：子字并行

由于每台桌面计算机都有自己的图形显示器，因此，随着处理器中晶体管数量的增加，用于桌面计算机的微处理器不可避免地要增加支持图形操作的功能。

许多图形系统最初都是用 8 位数据来表示三种基本颜色中的一种，外加 8 位数据来表示像素的位置。电话会议中和视频游戏中使用的扬声器和麦克风要求对声音进行支持。音频采样需要 8 位以上的精度，但是 16 位精度已经足够。

每种处理器对于字节或半字都有特殊的支持，从而使得在存储器中占据较少的空间（见 2.9 节）。然而，在典型的整数程序中，对这些数据类型的算术操作非常少，因此几乎不支持除数据传送之外的操作。设计师们发现许多视频和音频应用中通常对这类数据的向量做相同的操作。通过在 128 位内对进位链进行分割，处理器可以同时对 16 个 8 位、8 个 16 位、4 个 32 位或 2 个 64 位的运算同时进行并行操作。对加法器进行这样的分割的开销非常小。

将这种在一个宽字内部进行的并行操作称为子字并行，也可将其称为更加通用的数据级并行。它们也被称为向量或 SIMD（单指令多数据，见 6.6 节）。多媒体应用的日益广泛促使支持易于并行实现窄位宽操作的算术运算指令的出现。

例如，ARM 在 NEON 多媒体指令集中增加了 100 多条指令来支持子字并行，这些扩展的指令可以在 ARMv7 或 ARMv8 中实现。NEON 中增加了宽度为 256 字节的寄存器，它们可以当作 32 个 8 字节宽度的寄存器或者 16 个 16 字节宽度的寄存器使用。除了 64 位浮点数之外，NEON 支持你能够想到的任何子字数据类型。

- 8 位、16 位、32 位和 64 位无符号整数和带符号整数。
- 32 位浮点数。

图 3-19 给出了 NEON 基本指令的总结。

数据传送	算术运算	逻辑/比较
VLDR. F32	VADD. F32, VADD{L,W}{S8,U8,S16,U16,S32,U32}	VAND. 64, VAND. 128
VSTR. F32	VSUB. F32, VSUB{L,W}{S8,U8,S16,U16,S32,U32}	VORR. 64, VORR. 128
VLD {1, 2, 3, 4}, {I8, I16, I32}	VMUL. F32, VMULL{S8,U8,S16,U16,S32,U32}	VEOR. 64, VEOR. 128

图 3-19 子字并行的 ARM NEON 指令总结。使用大括号 {} 表示基本操作的可选对象：{S8, U8, 8} 表示 8 位有符号和 8 位无符号整数或者与类型无关的 8 位数据，16 个这些类型的数据可映射为一个 128 位寄存器；{S16, U16, 16} 表示 16 位有符号和 16 位无符号整数或者与类型无关的 16 位数据，8 个这些类型的数据可映射为一个 128 位寄存器；{S32, U32, 32} 表示 32 位有符号和 32 位无符号整数或者与类型无关的 32 位数据，4 个这些类型的数据可映射为一个 128 位寄存器；{S64, U64, 64} 表示 64 位有符号和 64 位无符号整数或者与类型无关的 64 位数据，2 个这些类型的数据可映射为一个 128 位寄存器；{F32} 表示 32 位有符号或无符号浮点数，4 个这种类型的数据可映射为一个 128 位寄存器；向量装载（vector load）把一个  $n$  元的结构从存储器读入 1 个、2 个、3 个或 4 个 NEON 寄存器中。它把一个  $n$  元（element）的结构装载到一个线性结构中（见 6.6 节），寄存器中没有进行装载的部分保持不变。向量存储（vector store）将 1 个、2 个、3 个或 4 个 NEON 寄存器中的内容写入存储器中的一个结构中去。

数据传送	算术运算	逻辑/比较
VST {1, 2, 3, 4}, {I8, I16, I32}	VMLA. F32, VMLAL{S8, U8, S16, U16, S32, U32}	VBIC. 64, VBIC. 128
VMOV. {I8, I16, I32, F32}, # imm	VMLS. F32, VMLSL{S8, U8, S16, U16, S32, U32}	VORN. 64, VORN. 128
VMVN. {I8, I16, I32, F32}, # imm	VMAX. {S8, U8, S16, U16, S32, U32, F32}	VCEQ. {I8, I16, I32, F32}
VMOV. {I64, I128}	VMIN. {S8, U8, S16, U16, S32, U32, F32}	VCGE. {S8, U8, S16, U16, S32, U32, F32}
VMVN. {I64, I128}	VABS. {S8, S16, S32, F32}	VCGT. {S8, U8, S16, U16, S32, U32, F32}
	VNEG. {S8, S16, S32, F32}	VCLE. {S8, U8, S16, U16, S32, U32, F32}
	VSHL. {S8, U8, S16, U16, S32, U32, S64, U64}	VCLT. {S8, U8, S16, U16, S32, U32, F32}
	VSHR. {S8, U8, S16, U16, S32, U32, S64, U64}	VTST. {I8, I16, I32}

图 3-19 (续)

**01 精解** 除了有符号和无符号整数外, ARM 还包含 4 种大小的“定点”格式, 分别是 I8、I16、I32 和 I64, 16 个 I8、8 个 I16、4 个 I32 或 2 个 I64 可以映射到 1 个 128 位的寄存器。定点数的一部分是尾数(二进制小数点的右边), 另一部分是整数(二进制小数点的左边)。二进制小数点的位置在软件层面上可见。许多 ARM 处理器没有浮点硬件, 因此浮点操作必须使用库例程来实现。定点算术运算要比软件实现的库例程快得多, 但是程序员需要做更多的工作。

223

### 3.7 实例: x86 中流处理 SIMD 扩展和高级向量扩展

最初, x86 中的 MMX (MultiMedia eXtension, 多媒体扩展) 指令和 SSE (Streaming SIMD Extension, 流处理 SIMD 扩展) 指令与 ARM NEON 中的操作类似。第 2 章中提到在 2001 年 Intel 在其体系结构中增加了 144 条指令作为 SSE2 的一部分, 包括了双精度浮点寄存器和操作。它包含了可用作浮点操作数的 8 个 64 位寄存器。AMD 将寄存器数量扩展到 16 个, 作为 AMD64 的一部分, 称之为 XMM, 这些寄存器被 Intel 重新称为 EM64T。图 3-20 总结了 SSE 和 SSE2 指令。

数据传送	算术	比较
MOV {A/U} {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 {H/L} {PS/PD} xmm, mem/xmm	MUL {SS/PS/SD/PD} xmm, mem/xmm	
	DIV {SS/PS/SD/PD} xmm, mem/xmm	
	SORT {SS/PS/SD/PD} mem/xmm	
	MAX {SS/PS/SD/PD} mem/xmm	
	MIN {SS/PS/SD/PD} mem/xmm	

图 3-20 x86 的 SSE/SSE2 浮点指令。xmm 是指一个 128 位 SSE2 寄存器操作数; mem/xmm 是指该操作数要么是一个存储器操作数, 要么是一个 SSE2 寄存器操作数。用大括号 {} 表示基本操作可选择的类型: {SS} 表示标量的单精度浮点数, 或 128 位 SSE2 寄存器中的 1 个 32 位操作数; {PS} 表示组合的单精度浮点数, 或者 128 位 SSE2 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 位寄存器中的一个 64 位操作数; {PD} 表示组合的双精度浮点数, 或者 128 位 SSE2 寄存器中的 2 个 64 位操作数; {A} 表示存储器中对齐的 128 位操作数; {U} 表示存储器中不对齐的 128 位操作数; {H} 表示传送 128 位操作数的高半部分; {L} 表示传送 128 位操作数的低半部分

224

在一个寄存器中，除了能够存放一个单精度或双精度数之外，Intel 允许将多个操作数组合在一起放在一个 128 位 SSE2 寄存器中：4 个单精度或 2 个双精度数。因此，SSE2 的 16 个浮点寄存器实际上是 128 位宽。如果操作数能够在存储器中组织为 128 位对齐的数据，则每条 128 位的数据传送指令可以装载（load）或保存（store）多个操作数。这种组合的浮点数格式由可以并行进行 4 个单精度（PS）或 2 个双精度（PD）数运算的算术操作支持。

2011 年，Intel 使用高级向量扩展（advanced vector extension，AVX）将寄存器宽度再次加倍，现在称之为 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。现在，SSE 和 SSE2 指令可以对 YMM 寄存器的低 128 位进行操作。因此，为了使用 128 位和 256 位操作，在 SSE2 汇编指令操作码前加上前缀字母 v（表示向量），然后使用 YMM 寄存器名字替代 XMM 寄存器名字。例如，进行 2 个 64 位浮点加法<sup>①</sup>操作的 SSE2 指令。

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

变为

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

该指令进行 4 个 64 位浮点加法<sup>②</sup>。

**01 精解** AVX 也对 x86 增加了 3 地址指令。例如，vaddpd 可以有如下形式：

```
vaddpd %ymm0, %ymm1, %ymm4 # %ymm4 = %ymm1 + %ymm2
```

而标准的 2 地址指令为：

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

与 MIPS 不同，x86 的目标操作数位于右边。3 地址可以减少计算所需的寄存器数量和指令数量。

### 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，我们将在 Intel Core i7 上在有无 AVX 的情况下运行相同的代码。图 3-21 给出了一个用 C 语言写的矩阵乘法的代码，它还没有进行优化。就像我们在 3.5 节看到的，该程序通常称为 DGEMM，表示双精度通用矩阵乘法。从该版本出发，我们增加一个新的称为“加速”的小节来说明在底层硬件的基础上使用适应的软件的性能提升。这里的底层硬件是 Intel Core i7 的 Sandy Bridge。在第 3、4、5、6 章中，该小节将逐步使用每章介绍的思想来提高 DGEMM 的性能。

图 3-22 给出了图 3-21 中内循环的 x86 汇编代码。以 v 开头的 5 条浮点指令与 AVX 指令类似，但是需要注意的是它们使用的是 XMM 寄存器，而不是 YMM 寄存器，另外它们在指令名字里包含了 sd，代表着向量双精度。我们将对子字并行指令作简要定义。

由于编译程序员最终能够使用 x86 的 AVX 指令生成高质量代码，因此现在我们必须使用 C 循环体的属性，通过“欺骗”的方式，告诉编译器如何生成高质量的代码。图 3-23 是图 3-21 的加强版，给出了 Gnu C 编译器产生的 AVX 代码。图 3-24 给出了带注释的 x86 代码，它是在编译时使用 gcc-O3 级优化选项的输出。

图 3-23 第 6 行的声明使用了 \_m256d 的数据类型，用来告诉编译器变量将保存 4 个双精度浮点值。第 6 行的内蕴函数 \_mm256\_load\_pd() 使用 AVX 指令从矩阵 c 中并行的(\_pd)取出 4 个双精度浮点数到 c0。第 6 行的地址计算 c + i + j \* n 表示元素 c[i + j \* n]。与之相应的

<sup>①</sup> 原文为“乘法”。——译者注

<sup>②</sup> 原文为“乘法”。——译者注

是在第 11 行中的最后一步，使用内蕴函数 `_mm256_store_pd()` 将 `c0` 中的 4 个双精度浮点数保存到矩阵 `C` 中。由于在每次迭代时处理 4 个元素，第 4 行中的外层 for 循环的循环变量 `i` 做加 4 操作，而不像图 3-21 中第 3 行的加 1 操作。

```

1. void dgemm (int n, double* A, double* B, double* C)
2. {
3.     for (int i = 0; i < n; ++i)
4.         for (int j = 0; j < n; ++j)
5.         {
6.             double cij = C[i+j*n]; /* cij = C[i][j] */
7.             for( int 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 3-21 未经优化的双精度矩阵乘法的 C 语言版本，称之为双精度通用矩阵乘法 (GEMM)。因为我们通过参数 `n` 传递矩阵的维数，该版本的 DGEMM 使用的是矩阵 `C`、`A`、`B` 的一维版本，并且强调使用算术运算来获得更好的性能，而不像 3.5 节中使用直观的二维阵列。注释提醒我们使用这种更加直观的符号

```

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,%r11d            # register %r11 = %r11 + 1
12. vmovsd %xmm0,(%r10)          # Store %xmm0 into C element

```

图 3-22 图 3-21 中嵌套循环体使用优化编译后对应的 x86 汇编语言代码。虽然只是 64 位数据，编译器使用了 AVX 版本指令，而不是 SSE2，因此每条指令可以使用 3 个地址而不是 2 个地址（见 3.7 节的精解）

在循环体内部，首先在第 9 行又一次使用 `_mm256_load_pd()` 取入 `A` 的 4 个元素。为了将这些元素与 `B` 的一个元素相乘，第 10 行使用了内蕴函数 `_mm256_broadcast_sd()`，它将标量双精度数复制为相同的 4 份——在这种情况下，`B` 的一个元素在一个 YMM 寄存器中。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 个和上。

图 3-24 给出了编译器生成的内循环体的 x86 代码。可以看到 5 条 AVX 指令——它们全部

以 v 开头，并且其中 4 条使用了 pd 表示并行的双精度——与前面提到的 C 内在属性一致。代码与图 3-22 中所示代码非常类似：都使用 12 条指令，整数指令几乎相同（但使用不同的寄存器），浮点指令的不同之处仅仅在于使用 XMM 寄存器的标量双精度（sd）和使用 YMM 寄存器的并行双精度（pd）。不同之处在于图 3-24 的第 4 行，A 中每个元素必须与 B 中每个元素相乘。一种解决方法是将 64 位 B 元素的 4 个相同的备份依次放入 256 位的 YMM 寄存器中，正如 vbroadcastsd 指令所做的工作。

对于 32 乘 32 的矩阵，图 3-21 中未优化的 DGEMM 在一个 2.6GHz 的 Intel Core i7（Sandy Bridge）的一个核上运行时性能为 1.7GFLOPS（每秒浮点操作次数）。图 3-23 中的优化代码的性能为 6.4GFLOPS。由于使用了子字并行，在许多操作中可以获得 4 倍的加速，因此 AVX 版本要比原始版本快 3.85 倍，性能增加了接近 4 倍。

```

1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B, double* C)
3. {
4.     for ( int i = 0; i < n; i+=4 )
5.         for ( int j = 0; j < n; j++ ) {
6.             _mm256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
7.             for( int 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 3-23 DGEMM 的优化 C 版本，使用 C 的循环体属性为 x86 生成 AVX 子字并行指令。  
图 3-24 显示了内循环编译后的汇编语言

```

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 3-24 编译图 3-23 中优化的嵌套循环体的 x86 汇编语言。注意与图 3-22 相同，区别仅在于 5 个浮点操作现在使用 YMM 寄存器和 pd 版本的指令来进行并行双精度操作，而不是 sd 版本的标量双精度

**01 精解** 同 1.6 节中的精解相同, Intel 提供了 Turbo 模式, 暂时运行在较高时钟频率下, 直到芯片过热。Intel Core i7 (Sandy Bridge) 在 Turbo 模式可将频率从 2.6GHz 增加到 3.3GHz。以上的结果是在关闭 Turbo 模式下获得的。如果将 Turbo 模式打开, 由于时钟频率提高了  $3.3/2.6 = 1.27$  倍, 未优化的 DGEMM 性能将提升为 2.1GFLOPS, AVX 性能将提升为 8.1GFLOPS。当一个八核芯片中只使用一个核时, Turbo 模式会很好的工作, 因为在这种情况下, 单个核可以使用比共享情况下更多的功耗。

228

### 3.9 谬误与陷阱

数学可以被定义为这样的学科, 我们绝不知道我们在谈论什么以及我们所谈论的是否正确。

——伯兰特·罗素, 《近来关于数学原理的发言》, 1901

算术中常见的谬误与陷阱通常是由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引起的。

**谬误:** 正如整数乘法中左移指令可以代替与 2 的幂次方数相乘一样, 右移指令也可以代替与 2 的幂次方数相除。

回忆一下二进制数  $x$ , 其中  $xi$  代表第  $i$  位, 有

$$\cdots + (x_3 \times 2^3) + (x_2 \times 2^2) + (x_1 \times 2^1) + (x_0 \times 2^0)$$

将  $x$  右移  $n$  位看起来似乎是被  $2^n$  相除相同。<sup>②</sup>事实上, 对于无符号整数确实如此。问题出在有符号整数上。例如, 假设我们用  $-5_{10}$  除以  $4_{10}$ , 商就是  $-1_{10}$ 。 $-5_{10}$  的补码形式是

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_2$$

根据这个谬误, 右移 2 位就是被  $4_{10}$  除 ( $2^2$ ):

$$0011\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2$$

由于符号位上是 0, 所以结果很明显是错的。右移后的值实际是  $1\ 073\ 741\ 822_{10}$  而不是  $-1_{10}$ 。

一种解决办法是算术右移时, 进行符号位扩展而不是移入 0。 $-5_{10}$  算术右移 2 位得到

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2$$

结果是  $-2_{10}$  而不是  $-1_{10}$ , 虽然很接近, 但依然不正确。

**陷阱:** 浮点加法是不能使用结合律的。

结合律适用于一系列整型的二进制补码加法, 即使在计算过程中发生溢出。然而, 因为浮点数是实数的近似表示, 且计算机算术精度有限, 结合律不能适用于浮点数。假定浮点数可以表示一个很大的数的范围, 当两个不同符号的大数与一个小数相加时就会发生问题。例如, 对于  $c + (a + b) = (c + a) + b$ , 假设  $c = -1.5_{10} \times 10^{38}$ ,  $a = 1.5_{10} \times 10^{38}$ ,  $b = 1.0$ , 它们都是单精度数。

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

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

由于浮点数精度有限且结果是实数结果的近似值,  $1.5_{10} \times 10^{38}$  远远大于  $1.0_{10}$ , 因此  $1.5_{10} \times 10^{38} + 1.0$  仍然是  $1.5_{10} \times 10^{38}$ , 这就是为什么  $c$ 、 $a$ 、 $b$  的和根据浮点加法的计算顺序不同有 0.0

229

② 此处原书中  $x$  为  $C$ , 我们认为此处  $x$  更合适, 因为后边表示用  $xi$ 。原书中  $x^3$ 、 $x^2$  有误, 3、2 不应为上角。 $2^n$  也有误,  $n$  也不应为上角。——译者注

和 1.0 两种结果，所以  $c + (a + b) \neq (c + a) + b$ ，因此浮点加法不能使用结合律。

**谬误：**并行执行策略不但适用于整型数据类型，同样也适用于浮点数据类型。

一般情况下，首先编写串行运行的程序，然后再编写并行运行的程序，这就自然产生一个问题：“两个版本的程序能否得到相同的结果？”如果是否定的答案，那么你就得推断并行程序中有一个需要消除的 bug。

该方法假定将串行结构转化为并行结构时，计算机算术不会影响计算结果。这就是说，如果要将 100 万个数相加，无论使用 1 个处理器还是使用 100 个处理器应该得到相同的结果。该假定适用于二进制补码整数，因为整数加法可结合。然而，因为浮点加法不能结合，所以该假定不适用。

这个谬误有一个更加令人烦恼的情况在并行机上可能发生。并行机上操作系统调度器会根据并行程序的运行情况来使用不同数目的处理器。对并行无意识的程序员可能会因为程序每次运行结果总有些不同而苦恼，即使是相同的代码和输入，这是因为每次运行使用不同数目的处理器可能导致浮点求和以不同的顺序进行。

在这个困境下，写并行代码并使用了浮点数的程序员需要验证结果是否可信，即便结果可能与顺序执行的结果不一致。处理这个问题的领域称为数值分析，关于该问题本身就可以写一本教科书。这也是像 LAPACK 和 SCALAPAK 这样的数学库流行的一个原因。这些数学库在顺序和并行执行下都被验证是有效的。

230

**陷阱：**MIPS 指令 addiu（无符号立即数加）会对 16 位立即数域进行符号扩展。

当我们不关心上溢时，addiu 经常用于将常数和有符号整数相加。由于 MIPS 没有立即数减的指令，所以 MIPS 体系结构设计者决定对该指令的立即数域进行符号扩展，以支持立即数为负数时的需要。

**谬误：**只有理论数学家才会关心浮点精度。

1994 年 11 月的报纸新闻头条证明了这个观点是错误的（见图 3-25）。下面的故事在标题的后面。



图 3-25 1994 年 11 月的一些报刊文章，包括《纽约时报》《圣何塞信使报》《旧金山新闻》《信息世界》。Pentium 浮点 bug 甚至成为电视节目“David Letterman Late Show”的“十大新闻”。Intel 最后花了 3 亿美元来替换掉有 bug 的芯片

Pentium 用一种标准的浮点除算法每步生成多个商位，使用除数的最高几位和被除数猜测商的下面 2 位。这个猜测是利用一个含 -2、-1、0、+1、+2 的查找表。猜测结果和除数相乘，然后从余数中减去，获得新的余数。像不恢复除法一样，如果前面的一个猜测使得余数太大，部分余数将在下面的执行中进行调整。

很明显，Intel 认为有 5 个来自 80486 查找表的元素不会访问到，因此，他们在 Pentium 中优化了此查找表，在一些情况下返回 0 而不是 2。Intel 错了：前 11 位总是正确的，但错误会偶尔在 12 位和 52 位之间出现，或者说十进制下在第 4 位到第 15 位出现。231

弗吉尼亚州林奇伯格学院的数学家托马斯·内斯里（Thomas Nicely）在 1994 年 9 月发现了这个 bug。在拨打了 Intel 技术支持电话但没有获得官方回应后，他将他的发现公布在了因特网上。这引发了商业杂志上的一个故事，进一步引发了 Intel 发布了一条声明。Intel 称其为一个“小故障”，仅对理论数学家有影响，对于电子制表软件的用户来说，该缺陷只有 27 000 年才会发生一次。IBM 研究院很快提出反对，指出电子制表软件的用户平均每 24 天就能遇到一次这样的错误。很快，1994 年 12 月 21 日，Intel 发布了如下声明：

我们 Intel 对最近发布的 Pentium 处理器的缺陷处理真诚地道歉。‘Intel Inside’ 标记的含义是指您的计算机拥有一颗在质量和性能上首屈一指的微处理器。有上千的 Intel 雇员为了实现这个目标而努力工作。但是，没有一款微处理器是完美的，Intel 会继续相信，从技术层面上来讲，任何一个微小的问题都有它的生命期。尽管 Intel 肯定会对 Pentium 处理器的这个版本负责到底，但我们也意识到了用户的关切。我们要解决这种关切。任何消费者在他们计算机生命周期的任何时刻，只要他们需要，Intel 会免费为其更换 Pentium 处理器，使浮点除缺陷得到纠正。

分析家估计这次召回花费了 Intel 5 亿美元，Intel 的工程师那年没有拿到圣诞节奖金。

这次事件对每个人来说，都有一些值得思考的地方。如果在 1994 年的 7 月修复了这个 bug 会少花多少钱？修复 Intel 的名声需要多大的代价？一个广泛应用的，像微处理器这样的产品出现 bug，其相关的责任有多么重大？

### 3.10 本章小结

在过去的几十年里，计算机算术在很大程度上被标准化，这极大地增强了程序的可移植性。在当今的每台计算机中，都有二进制补码整数算术，如果它支持浮点，则提供 IEEE 754 二进制浮点算术。

计算机算术和用纸和笔手算的算术不同的地方是受到有限精度的约束。这个限制可能会因为计算中数大于或者小于预先的设定而导致无效操作。这种异常称为“上溢”或“下溢”，可能导致异常、中断或类似于意外的子程序调用。第 4 章和第 5 章更详细地讨论了异常。

浮点算术因为是对实际的数字的近似而增加了挑战性，而且要小心确保所选的计算机数能最接近地表示实际数字。不精确和有限的表达带来的挑战是数值分析领域灵感的部分来源。最近转向并行性的趋势使得数值分析再次被关注起来，在顺序计算机上是完全安全的东西，在并行计算机里需要重新考虑，在寻找快速的算法的同时也要有正确的结果。

数据级并行，特称为子字并行，为算术操作密集型（无论是整数或者浮点数操作）性能的提高开辟了一条简单的路径。我们展示了可以使用同时进行 4 个浮点操作的指令来将矩阵乘法加速大约 4 倍。

在本章和第 2 章解释计算机算术时，更多地采用 MIPS 指令集进行描述。容易混淆的一点是这两章讲到的指令和 MIPS 芯片中执行的指令以及 MIPS 汇编器接受的指令之间的关系。图 3-26 和图 3-27 试图讲明白这点。

图 3-26 列出了本章和第 2 章中提到的 MIPS 指令。我们将图中左边的指令集称为 MIPS 核

心指令。在右边的指令称为 MIPS 算术核心。图 3-27 的左边是包含了 MIPS 处理器执行的但图 3-26 中没有的指令。我们将全部的硬件指令集称为 MIPS-32。在图 3-27 的右边是被编译器接受但不属于 MIPS-32 的指令。我们称为伪 MIPS 指令。

MIPS 核心指令	名称	格式	MIPS 算术核心	名称	格式
加法	add	R	乘	mult	R
立即数加法	addi	I	无符号乘	multu	R
无符号加法	addu	R	除	div	R
立即数无符号加法	addiu	I	无符号除	divu	R
减法	sub	R	取自 Hi 寄存器	mfhi	R
无符号减法	subu	R	取自 Lo 寄存器	mflo	R
与	AND	R	取自系统控制寄存器 (EPC)	mfc0	R
立即数与	ANDi	I	浮点单精度加	add.s	R
或	OR	R	浮点双精度加	add.d	R
立即数或	ORi	I	浮点单精度减	sub.s	R
或非	NOR	R	浮点双精度减	sub.d	R
逻辑左移	sll	R	浮点单精度乘	mul.s	R
逻辑右移	srl	R	浮点双精度乘	mul.d	R
取立即数的高位	lui	I	浮点单精度除	div.s	R
取字	lw	I	浮点双精度除	div.d	R
存字	sw	I	浮点单精度取字	lwcl	I
取无符号半字	lhu	I	浮点单精度存字	swcl	I
存半字	sh	I	浮点双精度取字	ldcl	I
取无符号字节	lbu	I	浮点双精度存字	sdc1	I
存字节	sb	I	浮点真则跳转	bclt	I
链接取 (原子更新)	ll	I	浮点假则跳转	bclf	I
条件存 (原子更新)	sc	I	浮点单精度比较	c.x.s	R
相等则跳转	beq	I	(x = eq, neq, lt, le, gt, ge)		
不相等则跳转	bne	I	浮点双精度比较	c.x.d	R
跳转	j	J	(x = eq, neq, lt, le, gt, ge)		
跳转并链接	jal	J			
寄存器跳转	jr	R			
小于则置位	slt	R			
小于立即数则置位	slti	I			
无符号比较, 小于则置位	sltu	R			
无符号比较, 小于立即数则置位	sltiu	I			

图 3-26 MIPS 指令集。本书集中介绍左列的指令。这个信息也可以在 MIPS 参考数据卡的第 1 列和第 2 列里找到

图 3-28 给出了 SPEC CPU2006 整数和浮点基准测试程序中 MIPS 指令的使用率。所有列出来的指令至少占执行指令的 0.2%。

注意, 尽管程序员和编译器编写人员可能为了更多的选项而使用 MIPS-32, MIPS 核心指令主宰了 SPEC CPU2006 整数程序, 而整数核心以及算术核心主宰了 SPEC CPU2006 浮点程序, 正如下表所列。

指令集	整数	浮点
MIPS 核心	98%	31%
MIPS 算术核心	2%	66%
剩余的 MIPS-32	0%	3%

保留的 MIPS-32	名称	格式	伪 MIPS	名称	格式
异或 (rs $\oplus$ rt)	xor	R	绝对值	abs	rd, rs
异或立即数	xori	I	变号 (signed 或者 unsigned)	negs	rd, rs
算术右移	sra	R	旋转左移	rol	rd, rs, rt
可变的逻辑左移	sllv	R	旋转右移	ror	rd, rs, rt
可变的逻辑右移	srlv	R	乘且不检查上溢 (signed 或者 uns.)	muls	rd, rs, rt
可变的算术右移	sraw	R	乘且检查上溢 (signed 或者 uns.)	mulos	rd, rs, rt
移至 Hi	mthi	R	除且检查上溢	div	rd, rs, rt
移至 Lo	mtlo	R	除且不检查上溢	divu	rd, rs, rt
取半字	lh	I	求余 (signed 或者 uns.)	rem	rd, rs, rt
取字节	lb	I	取立即数	li	rd, imm
取字的左边 (非对齐)	lwl	I	取地址	la	rd, addr
取字的右边 (非对齐)	lwr	I	取双字	ld	rd, addr
存字的左边 (非对齐)	swl	I	存双字	sd	rd, addr
存字的右边 (非对齐)	swr	I	非对齐取字	ulw	rd, addr
链接取 (原子更新)	ll	I	非对齐存字	usw	rd, addr
条件存 (原子更新)	sc	I	非对齐取半字 (signed 或者 unsigned)	ulhs	rd, addr
零则移	movz	R	非对齐存半字	ush	rd, addr
非零则移	movn	R	跳转	b	Label
乘和加 (S 或 uns.)	madds	R	等于零时跳转	beqz	rs, L
乘和减 (S 或 uns.)	msubs	I	比较并跳转 (signed 或者 unsigned)	bxs	rs, rt, L
大于等于零则跳转并链接	bgezal	I	(x = lt, le, gt, ge)		
小于零则跳转并链接	bltzal	I	相等则置位	seq	rd, rs, rt
跳转并链接寄存器	jalr	R	不相等则置位	sne	rd, rs, rt
和零比较并跳转	bxz	I	比较并置位 (signed 或者 unsigned)	sxs	rd, rs, rt
与零近似比较并跳转	bxzl	I	(x = lt, le, gt, ge)		
(x = lt, le, gt, ge)			取数给浮点 (s 或者 d)	lf	rd, addr
与寄存器值近似比较并跳转	bxl	I	浮点数存储 (s 或者 d)	sf	rd, addr
与寄存器值比较并自陷	tx	R			
与立即数比较并自陷	txi	I			
(x = eq, neq, lt, le, gt, ge)					
异常返回	rfe	R			
系统调用	syscall	I			
中断 (引起异常)	break	I			
从浮点移至整数	mfcl	R			
从整数移至浮点	mtcl	R			
浮点移 (s 或者 d)	movf	R			
如果零则浮点移 (s 或者 d)	movzf	R			
如果非零则浮点移 (s 或者 d)	movnf	R			
浮点平方根 (s 或者 d)	sqrtf	R			
浮点绝对值 (s 或者 d)	absf	R			
浮点变号 (s 或者 d)	negf	R			
浮点转换 (w、s 或者 d)	cvtwf	R			
浮点比较 (s 或者 d)	c.xnf	R			

图 3-27 保留的 MIPS-32 和伪 MIPS 指令集。*f* 代表单 (s) 或者 (d) 精度浮点指令，*s* 代表有符号和无符号 (u) 版本。MIPS-32 也有浮点指令，包括乘和加/减 (madd.f/msub.f)、向上舍入 (ceil.f)、截断 (trunc.f)、舍入 (round.f) 和倒数 (recip.f)。下划线表示这个字母表示数据类型

MIPS 核心指令	名称	整数	浮点	算术核心 + MIPS-32	名称	整数	浮点
加法	add	0.0%	0.0%	浮点双精度加	add.d	0.0%	10.6%
立即数加法	addi	0.0%	0.0%	浮点双精度减	sub.d	0.0%	4.9%
无符号加法	addu	5.2%	3.5%	浮点双精度乘	mul.d	0.0%	15.0%
立即数无符号加法	addiu	9.0%	7.2%	浮点双精度除	div.d	0.0%	0.2%
无符号减法	subu	2.2%	0.6%	浮点单精度加	add.s	0.0%	1.5%
与	AND	0.2%	0.1%	浮点单精度减	sub.s	0.0%	1.8%
立即数与	ANDi	0.7%	0.2%	浮点单精度乘	mul.s	0.0%	2.4%
或	OR	4.0%	1.2%	浮点单精度除	div.s	0.0%	0.2%
立即数或	ORi	1.0%	0.2%	浮点双精度取字	l.d	0.0%	17.5%
或非	NOR	0.4%	0.2%	浮点双精度存字	s.d	0.0%	4.9%
逻辑左移	sll	4.4%	1.9%	浮点单精度取字	l.s	0.0%	4.2%
逻辑右移	srl	1.1%	0.5%	浮点单精度存字	s.s	0.0%	1.1%
取立即数的高位	lui	3.3%	0.5%	浮点真则跳转	bclt	0.0%	0.2%
取字	lw	18.6%	5.8%	浮点假则跳转	bclf	0.0%	0.2%
存字	sw	7.6%	2.0%	浮点双精度比较	c.x.d	0.0%	0.6%
取字节	lbu	3.7%	0.1%	乘	mul	0.0%	0.2%
存字节	sb	0.6%	0.0%	算术右移	sra	0.5%	0.3%
相等则跳转 (o)	beq	8.6%	2.2%	取半字	lhu	1.3%	0.0%
不相等则跳转 (o)	bne	8.4%	1.4%	存半字	sh	0.1%	0.0%
跳转并链接	jal	0.7%	0.2%				
寄存器跳转	jr	1.1%	0.2%				
小于则置位	slt	9.9%	2.3%				
小于立即数则置位	slti	3.1%	0.3%				
无符号比较， 小于则置位	sltu	3.4%	0.8%				
无符号比较， 小于立即数则置位	sltiu	1.1%	0.1%				

图 3-28 在 SPEC2006 整数和浮点数中 MIPS 指令的使用频率。表中的所有指令要占到至少 1% 的份额。伪指令在执行前转化为 MIPS-32 指令，所以这里没有出现

本书的剩余部分，我们专注于 MIPS 核心指令——除了乘法、除法以外的整型指令集，以使计算机设计变得易于解释。正如你所看到的，MIPS 核心包含了绝大多数流行的 MIPS 指令；我们认为，理解运行 MIPS 核心的计算机将会给你足够的背景知识，去理解更为复杂的计算机。无论什么指令集或者其大小——MIPS、ARM、x86——永远不要忘记位模式没有内在的含义。同样，位模式可能表示一个带符号整数、无符号整数、浮点数、串、指令，等等。在存储程序计算机中，对位模式的操作决定其含义。

### 3.11 历史观点和拓展阅读

Gresham 法则（“劣币驱逐良币”），对于计算机则是“快的淘汰慢的，即使快的是错误的”。

——W. Kahan, 1992

绝不要放弃，绝不要，永远，永远，永远，不要放弃——任何事情，不论大小——绝不要放弃。

——温斯顿·丘吉尔，在 Harrow 学校的演讲，1941

本节回溯到冯·诺依曼来纵览浮点历史，包括有争议的 IEEE 标准的令人惊讶的成就，以及 x86 的 80 位浮点堆栈结构的基本原理。见配套网站上 3.11 节。

### 3.12 练习题

- 3.1 [5] <3.2> 5ED4 - 07A4 是无符号 16 位十六进制数时如何表示？结果必须使用 16 进制表示。
- 3.2 [5] <3.2> 5ED4 - 07A4 是带符号 16 位十六进制数且以符号 - 数值形式存放时如何表示？结果必须使用 16 进制表示。
- 3.3 [10] <3.2> 将 5ED4 转换成二进制数。使用十六进制表示计算机中的数值很具有吸引力的原因是什么？
- 3.4 [5] <3.2> 写出 4365 - 3412 使用无符号 12 位八进制数的表示形式。结果必须使用八进制表示。
- 3.5 [5] <3.2> 写出 4365 - 3412 使用带符号 12 位八进制数且以符号 - 数值形式存放的表示形式。结果必须使用八进制表示。
- 3.6 [5] <3.2> 假定 185 和 122 是无符号 8 位十进制整数。计算  $185 - 122$ 。是否有上溢或者下溢？
- 3.7 [5] <3.2> 假定 185 和 122 是带符号 8 位十进制整数且以符号 - 数值形式存放。计算  $185 + 122$ 。是否有上溢或者下溢？
- 3.8 [5] <3.2> 假定 185 和 122 是带符号 8 位十进制整数且以符号 - 数值形式存放。计算  $185 - 122$ 。是否有上溢或者下溢？
- 3.9 [10] <3.2> 假定 151 和 214 是带符号 8 位十进制整数且以补码形式存放。使用饱和算术计算  $151 + 214$ 。结果必须使用十进制。
- 3.10 [10] <3.2> 假定 151 和 214 是带符号 8 位十进制整数且以补码形式存放。使用饱和算术计算  $151 - 214$ 。结果必须使用十进制。
- 3.11 [10] <3.2> 假定 151 和 214 是无符号 8 位十进制数。使用饱和算术计算  $151 + 214$ 。结果必须使用十进制。
- 3.12 [20] <3.3> 使用图 3-3 所示的硬件描述计算八进制无符号 6 位整数 62 和 12 的乘积，并给出一个类似于图 3-6 中的表。必须给出每个步骤中每个寄存器的内容。
- 3.13 [20] <3.3> 使用与图 3-6 相同的一张表，使用图 3-5 所示的硬件描述计算十六进制无符号 8 位整数 62 和 12 的乘积。必须给出每个步骤中每个寄存器的内容。
- 3.14 [10] <3.3> 如果一个整数是 8 位宽，且每个步骤的操作需要 4 个时间单位，使用图 3-3 和图 3-4 的方法计算执行一次乘法必需的时间。假定在步骤 1a 中，无论是否加了被乘数还是加 0，加法都要执行。另外假设寄存器已经初始化（只需要计算执行乘法循环本身所需的时间）。如果是在硬件中执行，对被乘数和乘数的移位可以同时进行；如果是在软件中执行，则会一个做完再做下一个。对每种情况都给出解答。
- 3.15 [10] <3.3> 计算采用书中的方法（31 个垂直的加法堆栈）来执行乘法所需的时间。设整数位宽是 8，一个加法需 4 个单位时间。
- 3.16 [20] <3.3> 计算采用图 3-7 中的方法来执行乘法所需的时间。设整数位宽是 8，一个加法需 4 个单位时间。
- 3.17 [20] <3.3> 正如书中讨论的，一种增强性能的办法是做一次移位和加法来代替一次实际的乘法。例如，因为  $9 \times 6$  可以写成  $(2 \times 2 \times 2 + 1) \times 6$ ，所以我们可以通过将 6 左移 3 次再加上 6 来计算  $9 \times 6$ 。给出用移位和加/减法来计算  $0 \times 33 \times 0 \times 55$  的最好的方法。假设输入都是 8 位无符号整数。
- 3.18 [20] <3.4> 使用图 3-8 中的硬件结构计算 74 除以 21，并给出一个类似于图 3-10 中的表。你需要给出每一步中各个寄存器的值。假设输入都是 6 位无符号整数。
- 3.19 [30] <3.4> 用图 3-11 中的硬件结构计算 74 除以 21，并给出一个类似于图 3-10 中的表。你需要给出每一步中各个寄存器的值。假设 A 和 B 都是 6 位无符号整数。这个算法使用一个和图 3-9 中稍微不同的方法。这个算法你可能会认为很难，做一次或者两次试验，或者去网上寻找办法来让它正确工作。（提示：一种可能的解决方案是利用图 3-11 中暗示的那个余数寄存器既可右移也可左移的事实。）
- 3.20 [5] <3.5> 如果是补码整数，则这些位模式 0x0C000000 代表的十进制是多少？如果是无符号整数呢？

数呢?

- 238**
- 3.21 [10] <3.5> 如果位模式 0X0C000000 放在指令寄存器中, 那么将执行什么 MIPS 指令?
- 3.22 [10] <3.5> 如果是浮点数, 则位模式 0X0C000000 代表的十进制数是多少? 使用 IEEE 754 标准。
- 3.23 [10] <3.5> 写出十进制数 63.25 的二进制表达。设采用 IEEE 754 单精度格式。
- 3.24 [10] <3.5> 写出十进制数 63.25 的二进制表达。设采用 IEEE 754 双精度格式。
- 3.25 [10] <3.5> 写出十进制数 63.25 的二进制表达。设采用 IBM 单精度格式存储 (基数为 16 而不是 2, 有 7 位指数位)。
- 3.26 [20] <3.5> 写出  $-1.5625 \times 10^{-1}$  的二进制位模式。设采用一种类似 DEC PDP-8 使用的格式 (左边 12 位是以补码形式存储的指数, 而右 24 位是以补码形式存储的尾数。) 没有隐含 1。同 IEEE 754 标准的单精度和双精度比较, 评估这个 36 位位模式的范围和精确度。
- 3.27 [20] <3.5> IEEE 754-2008 包含一种“半精度”格式, 只有 16 位宽。最左边仍为符号位, 指数有 5 位宽且以余 -16 (excess -16) 的形式存储, 尾数有 10 位宽。具有隐含 1。写出  $-1.5625 \times 10^{-1}$  的这种格式的二进制位模式。同 IEEE 754 标准的单精度比较, 评估这个 16 位位模式的范围和精确度。
- 3.28 [20] <3.5> 惠普 2114、2115 和 2116 采用这样一种格式, 其最左边 16 位以补码形式存储着尾数, 紧跟着在另一个 16 位域里, 左边 8 位是尾数的扩展 (使尾数达到 24 位宽), 右边 8 位表示指数。然而, 作为一种有趣的交叉, 指数以符号 - 数值的形式存储且符号位在最右端! 写出  $-1.5625 \times 10^{-1}$  的这种格式的二进制位模式。没有隐含 1。同 IEEE 754 标准的单精度比较, 评估这个 32 位位模式的范围和精确度。
- 3.29 [20] <3.5> 手算  $2.6125 \times 10^1$  和  $4.150390625 \times 10^{-1}$  的和, 设 A 和 B 以练习题 3.27 中提到的 16 位半精度格式存储。假设有 1 位保护位、1 位舍入位和 1 位粘贴位, 并采用向最靠近的偶数舍入的模式。给出所有步骤。
- 3.30 [30] <3.5> 手算  $-8.0546875 \times 10^0$  和  $-1.79931640625 \times 10^{-1}$  的积, 设 A 和 B 以练习题 3.27 中提到的 16 位半精度格式存储。假设 1 位保护位、1 位舍入位和 1 位粘贴位, 并采用向最靠近的偶数舍入的模式。给出所有步骤; 然而, 作为书中已经做过的例子, 你可以以人们可读的格式来做一个乘法, 而不用练习题 3.12 到练习题 3.14 中描述的技术。注明是否上溢或者下溢。分别以练习题 3.27 中的 16 位浮点模式和十进制数写出你的答案。你的结果精确程度如何? 和你用计算器取得的结果相比呢?
- 239**
- 3.31 [30] <3.5> 手算  $8.625 \times 10^1$  除以  $-4.875 \times 10^0$ 。给出必要的步骤。假设有 1 个保护位、1 个舍入位和 1 个粘贴位, 并在必要时使用。以练习题 3.27 中的 16 位浮点格式和十进制格式给出最终的结果, 并比较十进制结果和用计算器得到的结果。
- 3.32 [20] <3.9> 手算  $(3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^3$ , 设每个数值都以练习题 3.27 中提到的 16 位半精度格式存储 (书中也有介绍)。假设有 1 位保护位、1 位舍入位和 1 位粘贴位, 并采用向最靠近的偶数舍入的模式。给出所有步骤, 并以 16 位浮点格式和十进制格式给出你的答案。
- 3.33 [20] <3.9> 手算  $3.984375 \times 10^{-1} + (3.4375 \times 10^{-1} + 1.771 \times 10^3)$ , 设每个数值都以练习题 3.27 中提到的 16 位半精度格式存储 (书中也有介绍)。假设有 1 位保护位、1 位舍入位和 1 位粘贴位, 并采用向最靠近的偶数舍入的模式。给出所有步骤, 并以 16 位浮点格式和十进制格式给出你的答案。
- 3.34 [10] <3.9> 根据练习题 3.32 和练习题 3.33 的结果, 计算  $(3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^3 = 3.984375 \times 10^{-1} + (3.4375 \times 10^{-1} + 1.771 \times 10^3)$  是否成立?
- 3.35 [30] <3.9> 手算  $(3.41796875 \times 10^{-3} \times 6.34765625 \times 10^{-3}) \times 1.05625 \times 10^2$ , 设每个值都以练习题 3.27 中提到的 16 位半精度格式存储 (书中也有介绍)。假设有 1 位保护位、1 位舍入位和 1 位粘贴位, 并采用向最靠近的偶数舍入的模式。给出所有步骤, 并以 16 位浮点格式和十进制格式给出你的答案。

- 3.36 [30] <3.9> 手算  $3.417\ 968\ 75 \times 10^{-3} \times (6.347\ 656\ 25 \times 10^{-3} \times 1.056\ 25 \times 10^2)$ ，设每个值都以练习题 3.27 中提到的 16 位半精度格式存储（书中也有介绍）。假设有 1 位保护位、1 位舍入位和 1 位粘贴位，并采用向最靠近的偶数舍入的模式。给出所有步骤，并以 16 位浮点格式和十进制格式给出你的答案。
- 3.37 [10] <3.9> 根据练习题 3.35 和练习题 3.36 的结果，计算  $(3.417\ 968\ 75 \times 10^{-3} \times 6.347\ 656\ 25 \times 10^{-3}) \times 1.056\ 25 \times 10^2 = 3.417\ 968\ 75 \times 10^{-3} \times (6.347\ 656\ 25 \times 10^{-3} \times 1.056\ 25 \times 10^2)$  是否成立？
- 3.38 [30] <3.9> 手算  $1.666\ 015\ 625 \times 10^0 \times (1.976\ 0 \times 10^4 + (-1.9744) \times 10^4)$ ，设每个值都以练习题 3.27 中提到的 16 位半精度格式存储（书中也有介绍）。假设有 1 位保护位、1 位舍入位和 1 位粘贴位，并采用向最靠近的偶数舍入的模式。给出所有步骤，并以 16 位浮点格式和十进制格式给出你的答案。
- 3.39 [30] <3.9> 手算  $(1.666\ 015\ 625 \times 10^0 \times 1.976\ 0 \times 10^4) + (1.666\ 015\ 625 \times 10^0 \times (-1.9744) \times 10^4)$ ，设每个值都以练习题 3.27 中提到的 16 位半精度格式存储（书中也有介绍）。假设有 1 位保护位、1 位舍入位和 1 位粘贴位，并采用向最靠近的偶数舍入的模式。给出所有步骤，并以 16 位浮点格式和十进制格式给出你的答案。
- 3.40 [10] <3.9> 根据练习题 3.38 和练习题 3.39 的结果，计算  $(1.666\ 015\ 625 \times 10^0 \times 1.976\ 0 \times 10^4) + (1.666\ 015\ 625 \times 10^0 \times (-1.9744) \times 10^4) = 1.666\ 015\ 625 \times 10^0 \times (1.976\ 0 \times 10^4 + (-1.9744) \times 10^4)$  是否成立？
- 3.41 [10] <3.5> 按照 IEEE 754 浮点格式，写出  $-1/4$  的位模式。你能精确表示  $-1/4$  吗？
- 3.42 [10] <3.5> 如果将  $-1/4$  自加 4 次得到多少？ $-1/4 \times 4$  是多少？它们相同吗？它们应该是多少？
- 3.43 [10] <3.5> 写出数值  $1/3$  的尾数的位模式，其浮点格式采用二进制编码的尾数。假设有 24 位，并且不需要进行规格化。这种表达精确吗？
- 3.44 [10] <3.5> 写出尾数的位模式，其浮点格式采用 BCD 编码（基 10）而不是基 2 的尾数。假设有 24 位，并且不需要进行规格化。这种表达精确吗？
- 3.45 [10] <3.5> 写出尾数的位模式，其浮点格式采用基 15 编码而不是基 2 的尾数。（基 16 编码使用符号 0~9 和 A~F。基 15 编码使用 0~9 和 A~E。）假设有 24 位，并且不需要进行规格化。这种表达精确吗？
- 3.46 [20] <3.5> 写出尾数的位模式，其浮点格式采用基 30 编码而不是基 2 的尾数。（基 16 编码使用符号 0~9 和 A~F。基 30 编码使用 0~9 和 A~T。）假设有 20 位，并且不需要进行规格化。这种表达精确吗？
- 3.47 [45] <3.6, 3.7> 下面的 C 代码实现了一个 4 阶 FIR 滤波器，其输入为数组 sig\_in。假设所有的数组元素为 16 位定点数。

```
for (i = 3; i < 128; i++)
    sig_out[i] = sig_in[i - 3] * f[0] + sig_in[i - 2] * f[1] +
        + sig_in[i - 1] * f[2] + sig_in[i] * f[3];
```

假设你要面向一个具有 SIMD 指令集且有 128 位寄存器的处理器，使用汇编语言对该代码进行优化。在不知道指令集细节的情况下，简要介绍一下你该怎样实现该代码，最大限度地使用子字并行操作，并且使寄存器和存储器间的数据传送量最少。指明你对使用的指令集的假设。

## 01 小测验答案

3.2 2

3.5 3