指令；或者如果一个操作数是有符号的而另一个是无符号的，则使用 mulhsu 指令。

3.3.5 总结

就像在小学学习到的纸笔计算方法一样，乘法硬件只是进行简单的移位和相加。编译器甚至会直接使用移位指令来做 2 的幂的乘法。通过使用更多硬件，我们可以做并行加法，并且做得更快。

硬件/软件接口 软件可以使用乘法高位指令来检查 64 位乘法溢出。如果 mulhu 的结果为 0，则 64 位乘法不会溢出。如果 mulh 结果的所有位都是 mul 结果的符号位的复制，则 64 位有符号乘法没有溢出。

3.4 除法

乘法的逆操作是除法，除法操作使用相对较少且很诡异。它甚至会出现无效的计算操作：除以 0。

Divide et impera.
拉丁语，意为“分而治之”，引自 Machiavelli 的一句政治箴言，1532

首先通过一个十进制数的长除法来回忆一下操作数的命名以及小学时学习的除法算法。由于与前一节类似的原因，我们仅使用十进制数字的 0 或 1。以下示例将计算 1001010_{10} 除以 1000_{10}：

```
                1001₁₀   商
除数  1000₁₀ ⌈1001010₁₀  被除数
             -1000
             ------
                  10
                 101
                1010
               -1000
               -----
                  10₁₀   余数
```

除法的两个源操作数分别叫作**被除数**和**除数**，除法的结果叫作**商**，随之产生的附带结果叫作**余数**。下面是表示各部分间关系的另一种方式：

被除数：被除的数。

除数：被被除数除的数。

$$被除数 = 商 \times 除数 + 余数$$

其中余数小于除数。少数情况下会有程序使用除法指令来获得余数而忽略商。

商：除法的主要结果；该数乘以除数再加上余数得到被除数。

基本除法算法试图查看可以减去一个多大的数字，从而每次得到商的一位数。我们精心挑选的十进制例子只使用数字 0 和 1，因此很容易计算出除数从被除数中减去的次数：要么是 0 次，要么是 1 次。二进制数只包含 0 或 1，因此二进制除法仅限于这两种选择，这就简化了二进制除法运算。

余数：除法的附带结果；该数加上商和除数的乘积得到被除数。

假设被除数和除数都是正数，那么商和余数也都是非负的。除法的源操作数和两个结果都是 64 位数，以下内容忽略符号位。

3.4.1 除法算法及其硬件实现

图 3-8 展示了模拟基本除法算法的硬件。在开始时将 64 位的商寄存器置 0。算法的每次

迭代都需要将除数右移一位，因此开始需要将除数放置到 128 位的除数寄存器的左半部分，并且每运算一步并将其右移 1 位，使之与被除数对齐。余数寄存器初始化为被除数。

图 3-9 展示了第一个除法算法的三个步骤。与人不同，计算机没有聪明到能预先知道除数是否小于被除数。它必须先在步骤 1 中减去除数，这正是我们实现比较所使用的方式。如果结果是正数，则除数小于或等于被除数，所以在商中生成一位 1（步骤 2a）。如果结果为负，则下一步是通过将除数加回余数来恢复原始值，并在商中生成一位 0（步骤 2b）。除数右移，然后再次迭代。在迭代完成后，余数和商将存放在其同名的寄存器中。

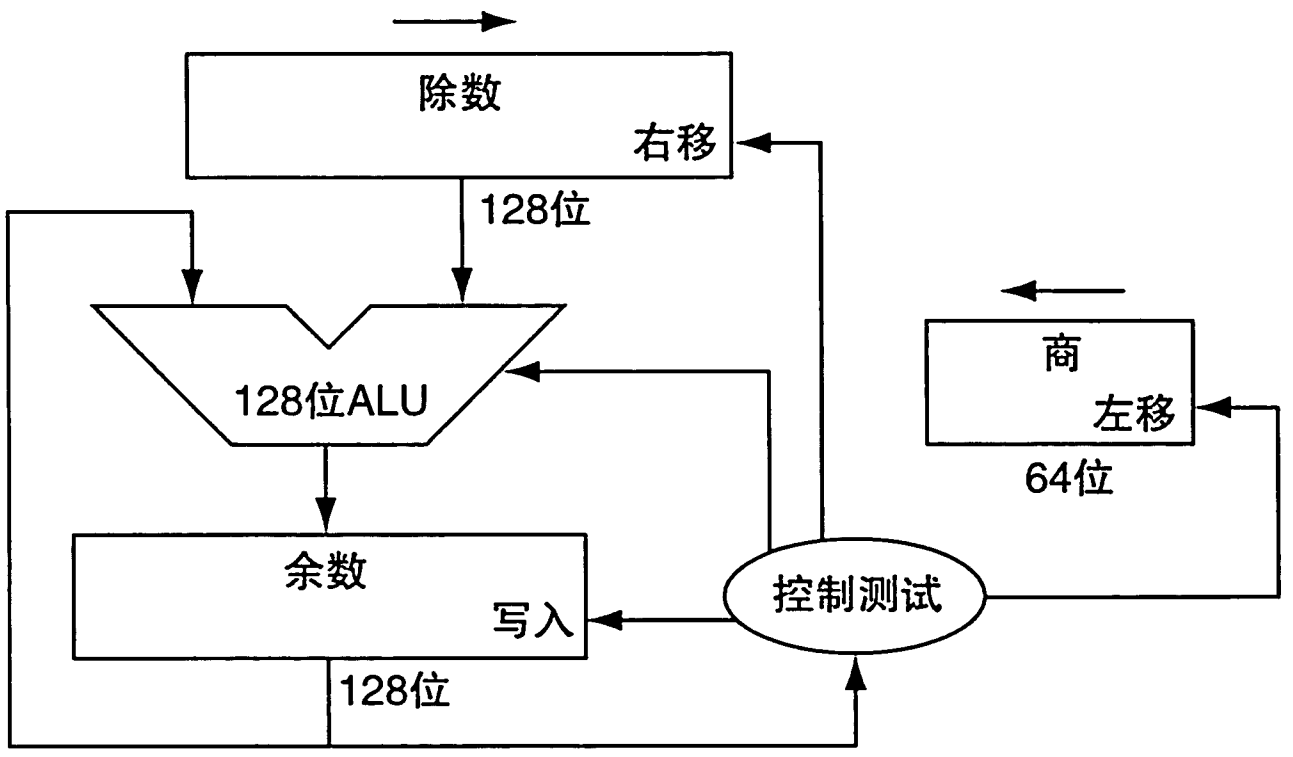


图 3-8 除法器硬件结构第一版。除数寄存器、ALU 和余数寄存器都是 128 位宽，只有商寄存器是 64 位宽。64 位的除数开始位于除数寄存器的左半部分且每次迭代右移 1 位。余数被初始化为被除数。控制逻辑决定除数和商寄存器何时移位，以及何时将新值写入余数寄存器

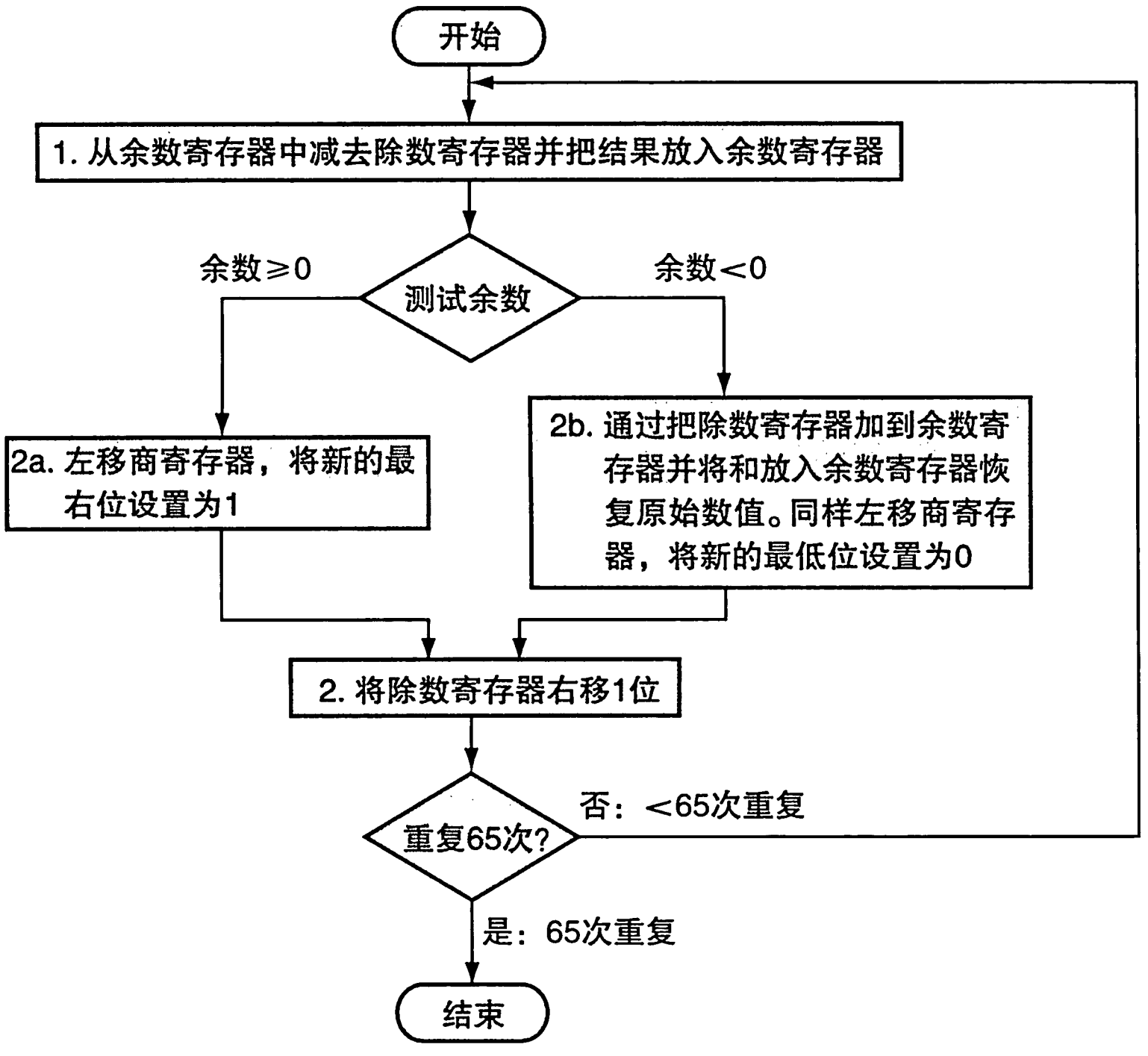


图 3-9 使用图 3-8 中硬件结构的除法算法。如果余数为正，则从被除数中减去除数，因此步骤 2a 产生商中的一位 1。步骤 1 后余数为负意味着不将除数从被除数中减去，所以步骤 2b 产生商中的一位 0，并把除数加到余数上，即为步骤 1 的逆操作。在步骤 3 中，最后一次移位根据下一次迭代的被除数将除数适当对齐。这些步骤重复 65 次

例题 | 除法算法

为节省篇幅，算法使用4位的版本，尝试用7_{10}除以2_{10}，即00000111_2除以0010_2。

答案 | 图3-10展示了每个步骤中各个寄存器的值，最后商为3_{10}，余数为1_{10}。注意，在步骤2中检测余数的正负时，仅简单检查余数寄存器的符号位是0还是1即可。这种算法令人惊讶之处在于需要n+1个步骤才能得到正确的商和余数。

步	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，左移商，商的第0位=0	0000	0010 0000	0000 0111
	3 ：右移除数	0000	0001 0000	0000 0111
2	1 ：余数=余数−除数	0000	0001 0000	①111 0111
	2b：余数<0 =>+除数，左移商，商的第0位=0	0000	0001 0000	0000 0111
	3 ：右移除数	0000	0000 1000	0000 0111
3	1 ：余数=余数−除数	0000	0000 1000	①111 1111
	2b：余数<0 =>+除数，左移商，商的第0位=0	0000	0000 1000	0000 0111
	3 ：右移除数	0000	0000 0100	0000 0111
4	1 ：余数=余数−除数	0000	0000 0100	⓪000 0011
	2a：余数≥0 =>左移商，商的第0位=1	0001	0000 0100	0000 0011
	3 ：右移除数	0001	0000 0010	0000 0011
5	1 ：余数=余数−除数	0001	0000 0010	⓪000 0001
	2a：余数≥0 =>左移商，商的第0位=1	0011	0000 0010	0000 0001
	3 ：右移除数	0011	0000 0001	0000 0001

图3-10　使用图3-9中算法的除法示例。用灰色圈出的位用于检测以决定下一步骤

这个算法及其硬件结构可以被改进得更快且更便宜。通过操作数移位和商与减法同时进行来加速。该细化包括注意哪里有未使用的寄存器和将加法器和寄存器宽度减半。图3-11展示了修改后的硬件。

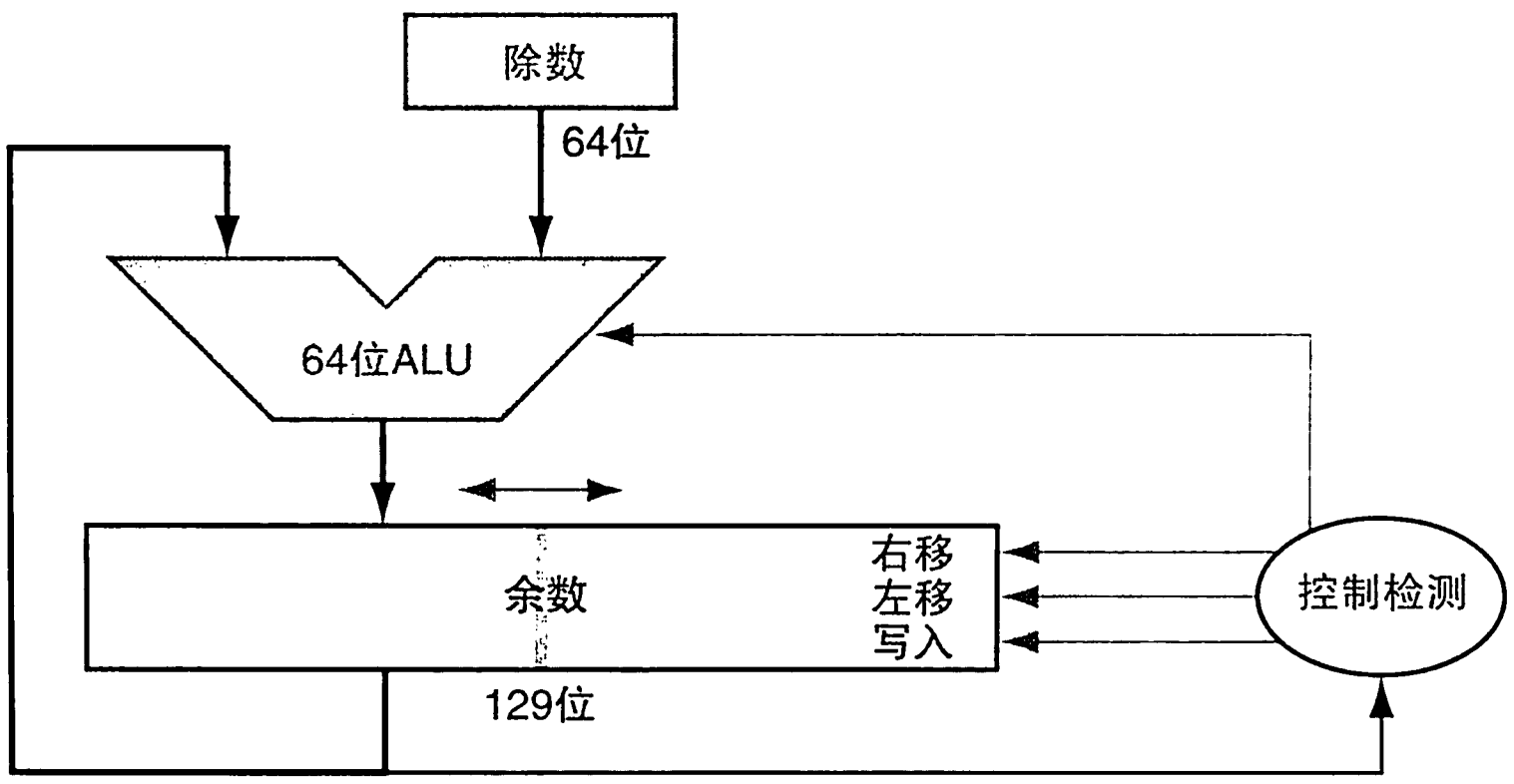


图3-11　改进版本的除法硬件。除数寄存器、ALU和商寄存器都是64位宽。和图3-8相比，ALU和除数寄存器位宽减半且余数被左移。这个版本把商寄存器和余数寄存器的右半部分拼接在了一起。如图3-5所示，余数寄存器增加到了129位以确保加法器进位不会被丢失

3.4.2 有符号除法

到目前为止，我们忽略了有符号数的除法。最简单的解决办法是记住除数和被除数的符号，如果符号相异，则商为负。

详细阐述 有符号除法之所以复杂的原因之一是必须设置余数部分的符号。记住以下等式必须始终保持：

$$被除数 = 商 \times 除数 + 余数$$

要理解如何设置余数的符号，我们来看看 $\pm 7_{10}$ 除以 $\pm 2_{10}$ 的所有组合的示例。第一种情况很简单：

$$+7 \div +2：商 = +3, + 余数 = +1$$

检查结果：

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

如果改变被除数的符号，商也一定会随之改变：

$$-7 \div +2：商 = -3$$

重写基本公式来计算余数：

$$余数 = (被除数 - 商 \times 除数) = -7 - (-3 \times +2) = -7 - (-6) = -1$$

因此，

$$-7 \div +2：商 = -3，余数 = -1$$

再次检查结果：

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

答案不是商为 -4 以及余数为 +1（这也适应于这个公式）的原因是商的绝对值会根据被除数和除数的符号而改变！显然，如果

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

编程将面临一个更大的挑战。这种异常情况通过让被除数和余数保持相同符号来避免，而不管除数和商的符号如何。

通过遵循相同规则来计算其他组合：

$$+7 \div -2：商 = -3，余数 = +1$$

$$-7 \div -2：商 = +3，余数 = -1$$

因此，如果源操作数的符号相反，那么正确的有符号除法算法的商为负，并让非零余数的符号与被除数的符号相匹配。

3.4.3 快速除法

摩尔定律适用于除法硬件以及乘法运算，所以希望能够通过其硬件来加速除法。通过使用许多加法器来加速乘法，但不能对除法使用相同的方法。因为在执行下一步运算之前，需要先知道减法结果的符号，而乘法运算可以立即计算 64 个部分积。

有些技术每步可以产生多于一位的商。SRT 除法技术试图根据被除数和余数的高位来查找表，以预测每步的多个商的位数。它依靠后续步骤纠正错误预测。今天的典型值是 4 位。关键在于猜测要减去的值。对于二进制除法，只有一个选择。这些算法使用余数的 6 位和除

数的4位来索引查找表，以确定每个步骤的猜测。

这种快速方法的准确性取决于查找表中的值是否合适。3.8节中的谬误展示了如果表不正确将会发生什么情况。

3.4.4 RISC-V中的除法

你可能已经观察到图3-5和图3-11中的乘法和除法都可以使用相同的顺序执行硬件。唯一需要的是一个可以左右移位的128位寄存器和一个实现加法或减法的64位ALU。

为了处理有符号整数和无符号整数，RISC-V有两条除法指令和两条余数指令：除（div），无符号除（divu），余数（rem）和无符号余数（remu）。

3.4.5 总结

支持乘法和除法的通用硬件允许RISC-V提供一个单独用于乘法和除法的64位寄存器。通过预测多位商再纠正错误的预测方法来加速除法。图3-12总结了前面两节RISC-V体系结构的优化处理。

RISC-V汇编语言

类别	指令	示例	含义	注解
算术运算	加	add x5, x6, x7	x5 = x6 + x7	三寄存器操作数
	减	sub x5, x6, x7	x5 = x6 - x7	三寄存器操作数
	立即数加	addi x5, x6, 20	x5 = x6 + 20	用于加常数
	小于置位	slt x5, x6, x7	x5 = 1 if x5 < x6, else 0	三寄存器操作数
	小于置位（无符号数）	sltu x5, x6, x7	x5 = 1 if x5 < x6, else 0	三寄存器操作数
	小于置位（立即数）	slti x5, x6, x7	x5 = 1 if x5 < x6, else 0	与立即数比较
	小于置位（无符号立即数）	sltiu x5, x6, x7	x5 = 1 if x5 < x6, else 0	与立即数比较
	乘	mul x5, x6, x7	x5 = x6 × x7	128位乘积的低64位
	高位乘	mulh x5, x6, x7	x5 = (x6 × x7) >> 64	128位有符号乘积的高64位
	高位乘（无符号数）	mulhu x5, x6, x7	x5 = (x6 × x7) >> 64	128位无符号乘积的高64位
	高位乘（有–无符号数）	mulhsu x5, x6, x7	x5 = (x6 × x7) >> 64	128位有–无符号乘积的高64位
	除	div x5, x6, x7	x5 = x6 / x7	除有符号64位数字
	无符号除	divu x5, x6, x7	x5 = x6 / x7	除无符号64位数字
	取余	rem x5, x6, x7	x5 = x6 % x7	对有符号64位除法取余
	无符号取余	remu x5, x6, x7	x5 = x6 % x7	对无符号64位除法取余
数据传输	取双字	ld x5, 40(x6)	x5 = Memory[x6 + 40]	从存储器取双字到寄存器
	存双字	sd x5, 40(x6)	Memory[x6 + 40] = x5	从寄存器存双字到存储器
	取字	lw x5, 40(x6)	x5 = Memory[x6 + 40]	从存储器取字到寄存器
	取字（无符号数）	lwu x5, 40(x6)	x5 = Memory[x6 + 40]	从存储器取无符号字到寄存器
	存字	sw x5, 40(x6)	Memory[x6 + 40] = x5	从寄存器存字到存储器
	取半字	lh x5, 40(x6)	x5 = Memory[x6 + 40]	从存储器取半字到寄存器
	取半字（无符号数）	lhu x5, 40(x6)	x5 = Memory[x6 + 40]	从存储器取无符号半字到寄存器
	存半字	sh x5, 40(x6)	Memory[x6 + 40] = x5	从寄存器存半字到存储器
	取字节	lb x5, 40(x6)	x5 = Memory[x6 + 40]	从存储器取字节到寄存器
	取字节（无符号数）	lbu x5, 40(x6)	x5 = Memory[x6 + 40]	从存储器取无符号字节到寄存器
	存字节	sb x5, 40(x6)	Memory[x6 + 40] = x5	从寄存器存字节到存储器
	取保留字	lr.d x5, (x6)	x5 = Memory[x6]	取；原子交换的前半部分
	存条件字	sc.d x7, x5, (x6)	Memory[x6] = x5; x7 = 0/1	存；原子交换的后半部分
	取立即数高位	lui x5, 0x12345	x5 = 0x12345000	取左移12位后的20位立即数
	加立即数高位到PC	auipc x5, 0x12345	x5 = PC + 0x12345000	用作程序计数器相对寻址
逻辑运算	与	and x5, x6, x7	x5 = x6 & x7	三寄存器操作数；按位与
	或	or x5, x6, x8	x5 = x6 | x8	三寄存器操作数；按位或
	异或	xor x5, x6, x9	x5 = x6 ^ x9	三寄存器操作数；按位异或
	与立即数	andi x5, x6, 20	x5 = x6 & 20	寄存器与常数按位与
	或立即数	ori x5, x6, 20	x5 = x6 | 20	寄存器与常数按位或
	异或立即数	xori x5, x6, 20	x5 = x6 ^ 20	寄存器与常数按位异或

图3-12　RISC-V核心指令系统。RISC-V机器语言列在本书的RISC-V参考数据卡中

类别	指令	示例	含义	注解
移位操作	逻辑左移	sll x5, x6, x7	x5 = x6 << x7	按寄存器给定位数左移
	逻辑右移	srl x5, x6, x7	x5 = x6 >> x7	按寄存器给定位数右移
	算术右移	sra x5, x6, x7	x5 = x6 >> x7	按寄存器给定位数算术右移
	逻辑左移立即数	slli x5, x6, 3	x5 = x6 << 3	根据立即数给定位数左移
	逻辑右移立即数	srli x5, x6, 3	x5 = x6 >> 3	根据立即数给定位数右移
	算术右移立即数	srai x5, x6, 3	x5 = x6 >> 3	根据立即数给定位数算术右移
条件分支	相等则跳转	beq x5, x6, 100	if (x5 == x6) go to PC+100	如果寄存器比较相等则发生PC相对跳转
	不等则跳转	bne x5, x6, 100	if (x5 != x6) go to PC+100	如果寄存器比较不等则发生PC相对跳转
	小于则跳转	blt x5, x6, 100	if (x5 < x6) go to PC+100	如果寄存器比较小于则发生PC相对跳转
	大于等于则跳转	bge x5, x6, 100	if (x5 >= x6) go to PC+100	如果寄存器比较大于等于则发生PC相对跳转
	小于则跳转（无符号数）	bltu x5, x6, 100	if (x5 < x6) go to PC+100	如果寄存器比较小于则发生PC相对跳转
	大于等于则跳转（无符号数）	bgeu x5, x6, 100	if (x5 >= x6) go to PC+100	如果寄存器比较大于等于则发生PC相对跳转
无条件跳转	跳转并链接	jal x1, 100	x1 = PC+4; go to PC+100	PC相对过程调用
	跳转并链接到寄存器所指位置	jalr x1, 100(x5)	x1 = PC+4; go to x5+100	过程返回；间接调用

图 3-12 （续）

硬件/软件接口 RISC-V 除法指令忽略溢出，因此软件必须判断商是否过大。除溢出外，除法也可能产生不适当的计算：除数为 0。一些计算机区分这两种异常事件。RISC-V 软件必须检查除数是否为 0 以及是否溢出。

详细阐述 如果余数为负，则算法不会立即将除数加回去。由于 $(r+d)\times 2-d = r-2+d\times 2-d = r\times 2+d$，它会在下一步中简单地将被除数加到移位后的余数上。这种每步花费一个时钟周期的非恢复（nonrestoring）除法算法将在练习中进一步探讨；前面的算法被称为恢复（restoring）除法。第三种算法称作不执行（nonperforming）除法算法，即如果减法结果为负，则不保存。它能使算术操作平均减少三分之一。

3.5 浮点运算

如果方向错了，再快也没用。
美国谚语

除了有符号和无符号整数外，编程语言还支持小数，在数学中被称作实数。下面是一些实数的例子：

$3.14159265\cdots_{10}$（pi）

$2.71828\cdots_{10}$（e）

0.000000001_{10} 或 $1.0_{10}\times 10^{-9}$（以秒为单位表示 1 纳秒）

$3\ 155\ 760\ 000_{10}$ 或 $3.15576_{10}\times 10^{9}$（以秒为单位表示 1 世纪）

科学记数法：小数点左边只有一位数字的表示数的方法。

请注意，在最后的例子中，这个数字（$3.15576_{10}\times 10^{9}$）并不表示小数，它比我们用 32 位有符号整数所能表示的还要大。后两个例子中的第二种表示方法称为**科学记数法**，该记数法在小数点左边只有一个数字。科学记数法中整数部分没有前导 0 的数字称为**规格化数**，这是一种常用的表示方法。例如，1.0×10^{-9} 是规格化的科学记数法表示，但是 $0.1_{10}\times 10^{-8}$ 和 $10.0_{10}\times 10^{-10}$ 就不是。

规格化：没有前导 0 的浮点表示法。

正如可以用科学记数法表示十进制数一样，我们同样可以用其来表示二进制数：

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

为了保证二进制数的规格化形式，我们需要一个基数，使得这个二进制数在移位后（相当于增大或减小基数的指数），小数点左侧必须只剩一位非零数。只有以 2 为基数才符合我

们的要求。由于基数不是10，我们还需要一个新的小数点名称：二进制小数点。

浮点：二进制小数点不固定的数的计算机表示。

支持这种数的计算机运算称为**浮点**运算，称作“浮点”是因为它表示二进制小数点不固定的数字，这与整数表示法不太相同。C语言中使用float来表示这些数字。正如在科学记数法中一样，数被表示为在二进制小数点左边只有一个非零数字的形式。在二进制中，其格式为：

$$1.xxxxxxxxx_2 \times 2^{yyyy}$$

（尽管在计算机中指数部分和其余部分都是用二进制表示的，但为了简化表达，我们用十进制来表示指数。）

利用规格化形式的标准科学记数法表示实数有三个优点：简化了包含浮点数的数据交换；由于都使用同一表示法，简化了浮点运算算法；提高了可存储在字中的数据的精度，因为无用的前导零占用的位被二进制小数点右边的实数位替代了。

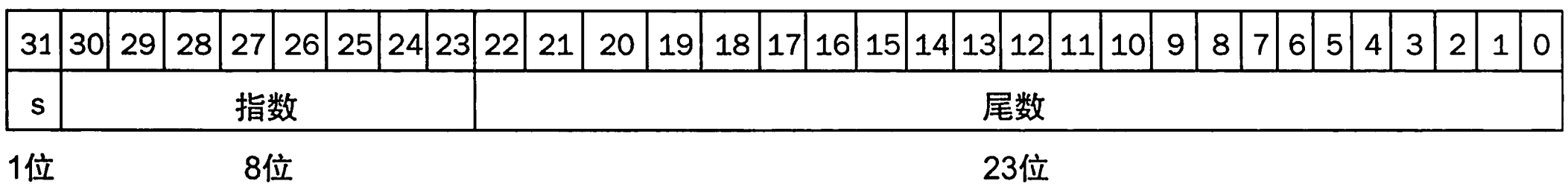
3.5.1　浮点表示

浮点表示的设计者必须在**尾数**的位数大小和**指数**的位数大小之间找到一个平衡，因为固定的字大小意味着若一部分增加一位，则另一部分就得减少一位。即要做精度和范围之间的权衡：增加尾数位数的大小可以提高小数精度，而增加指数位数的大小则可以增加数的表示范围。正如在第2章的设计原则中讲的那样，好的设计需要好的权衡。

尾数：该值通常在0和1之间，放置在尾数字段中。

指数：在浮点运算的数值表示系统中，放置在指数字段中的值。

浮点数通常占用多个字的长度。下图是RISC-V浮点数的表示方法，其中s是浮点数的符号（1表示负数），指数由8位指数字段（包括指数的符号）表示，尾数由23位数表示。正如第2章提过的那样，这种表示称为符号和数值，符号与数值的位是相互分离的。



通常来讲，浮点数可以这样表示：

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

F是尾数字段中表示的值，而E是指数字段表示的值；之后会对这两个字段之间的确切关系做详细说明。（我们很快就会看到稍微复杂一点的RISC-V。）

这些指定的指数和尾数位长使RISC-V计算机具有很大的运算范围。小到$2.0_{10} \times 10^{-38}$，大到$2.0_{10} \times 10^{38}$，计算机都能表示出来。但是它和无穷大不同，所以仍然可能存在数太大而表示不出来的情况。因此，和整点运算一样，浮点运算中也会发生**溢出**中断。注意这里的溢出表示因指数太大而无法在指数字段中表示出来。

浮点运算还会导致出现一种新的例外情况。正如程序员想知道他们什么时候计算了一个难以表示的太大的数一样，他们还想知道他们正在计算的非零小数是否变得小到无法表示；这两个事件都可能导致程序给出不正确的答案。为了和上溢区分开来，我们把这种情况称为**下溢**。当负指数太大而指数字段无法表示时，就会出现这

上溢：正指数太大而无法用指数字段表示的情况。

下溢：负指数太大而无法用指数字段表示的情况。

种情况。

减少下溢或上溢发生概率的一种方法是提供另一种具有更大指数范围的格式。在 C 语言中，这个数据类型称为双精度（double），基于双精度的运算称为**双精度**浮点运算，而**单精度浮点**就是前面介绍的格式。

双精度：以 64 位双字表示的浮点值。

单精度：以 32 位字表示的浮点值。

双精度浮点数需要一个 RISC-V 双字才能表示，如下所示，其中 s 仍然是数的符号位，指数字段为 11 位，尾数字段为 52 位。

RISC-V 双精度可以表示的实数范围小到 $2.0_{10} \times 10^{-308}$，大到 $2.0_{10} \times 10^{308}$。尽管双精度确实增加了指数字段能表示的范围，但其最主要的优点是由于有更大的有效位数而具有更高的精度。

3.5.2 例外和中断

在上溢或下溢时应该让计算机发生什么以让用户知道出现了问题？有些计算机会通过引发**例外**（有时也称作**中断**）来告知问题的出现。例外或中断在本质上是一种非预期的过程调用。造成溢出的指令的地址保存在寄存器中，并且计算机会跳转到预定义的地址以调用相应的例外处理程序。中断的地址被保存下来，以便在某些情况下可以在执行纠正代码之后继续执行原程序。（4.9 节介绍了有关例外的更多细节；第 5 章描述了例外异常和中断发生的其他情况）。RISC-V 计算机不会在上溢或下溢时引发例外，不过，软件可以读取浮点控制和状态寄存器（fcsr）来检测是否发生上溢或下溢。

例外：也称为中断。打扰程序执行的意外事件；用于检测溢出。

中断：来自处理器之外的例外（有些体系结构用术语中断表示所有的例外）。

3.5.3 IEEE 754 浮点数标准

这些格式并非 RISC-V 所独有。它们是 IEEE 754 浮点数标准的一部分，1980 年以后的几乎所有计算机都遵循该标准。该标准极大地提高了移植浮点程序的简易程度和计算机的运算质量。

为了将更多的位打包到数中，IEEE 754 让规格化二进制数的前导位 1 是隐含的。因此，在单精度下，该数实际上是 24 位长（隐含 1 和 23 位尾数），在双精度下则为 53 位长（1 + 52）。为了更精确，我们使用术语*有效位数*来表示隐含的 1 加上尾数，当*尾数*是 23 或 52 位数时，有效位数是 24 或 53 位。由于 0 没有前导 1，因此它被赋予保留的阶码 0，以便硬件不会给它附加一个前导位 1。

因此 $00...00_2$ 代表 0，其他的数就是用前面的形式，加上隐含的 1：

$$(-1)^S \times (1 + \text{尾数}) \times 2^E$$

其中，尾数位表示大小在 0 到 1 之间的小数，E 表示指数字段的值，稍后将对这部分做详细

分析。如果将尾数的各位从左到右依次用 s1，s2，s3，…来表示的话，则数的值为：

$$(-1)^S \times (1+(s1\times 2^{-1})+(s2\times 2^{-2})+(s3\times 2^{-3})+(s4\times 2^{-4})+\cdots)\times 2^E$$

图 3-13 展示了 IEEE 754 浮点数的编码。IEEE 754 的其他特征是用特殊符号来表示异常事件。例如，软件可以将结果设置为代表 + ∞ 或 − ∞ 的位模式，而不是除零中断。最大的指数是为这些特殊符号保留的。当程序员打印结果时，程序将输出一个无穷大的符号。（对于有数学训练经验的人来说，无穷的目的是形成实数的拓扑闭集。）

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	正负非规格化数
1～254	任意数	1～2046	任意数	正负浮点数
255	0	2047	0	正负无穷
255	非0	2047	非0	NaN（非数）

图 3-13　IEEE 754 对浮点数的编码。一个单独的符号位决定了符号。非规格化数在 3.5.8 节的“详细阐述”中描述。这些信息也可以在本书 RISC-V 参考数据卡的第 4 列中找到

IEEE 754 甚至还有表示无效运算结果的符号，例如 0/0 或无穷减去无穷。这个符号是 NaN，表示不是一个数。符号 NaN 的目的是让程序员可以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 的设计者也考虑到，对于浮点表示，尤其是进行排序操作时，最好能直接利用已有的整数比较硬件来处理。这就是符号位处于最高位的原因，这样一来就可以快速判定是小于 0、大于 0、还是等于 0。（这比简单的整数排序稍微复杂一点，因为这个记数法本质上是符号和数值的形式[⊖]而不是用 2 的补码表示。）

把指数字段放在有效位之前也简化了利用整数比较指令对浮点数的排序。因为只要两个数的指数部分符号相同，那么具有更大指数的数就一定更大。

负指数对简化排序提出了挑战。如果我们使用 2 的补码或其他指数为负数时指数字段的最高有效位为 1 的表示法，负指数反而会显得比较大。例如，$1.0_2\times 2^{-1}$ 以单精度表示为：

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
●	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

（注意前导 1 隐含在有效位中。）以上面方法表示的 $1.0_2\times 2^{+1}$ 看起来倒像更小的二进制数

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
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

因此，最理想的表示法是将最小的负指数表示为 $00...00_2$，并将最大的正指数表示为 $11...11_2$。这种表示法称作*移码表示法*。从移码表示的数减去原数就可以得到相应的偏移值，从而由无符号的移码可得到真实的值。

IEEE 754 规定单精度的偏移值为 127，因此指数为 −1 表示为 $-1 + 127_{10}$，即 $126_{10}= 01111110_2$，+1 表示为 $1 + 127_{10}$，即 $128_{10}= 10000000_2$。双精度的指数偏移值为 1023。带偏

⊖ 原码。——译者注

移值的指数意味着一个由浮点数表示的值实际上是：

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

单精度数的表示的范围从：

$$\pm 1.00000000000000000000000_2 \times 2^{-126}$$

到

$$\pm 1.11111111111111111111111_2 \times 2^{+127}$$

让我们演示一下浮点表示。

例题

用二进制的形式，分别用 IEEE 754 的单精度格式和双精度格式表示浮点数 -0.75_{10}。

答案 -0.75_{10} 又可以表示为：

$$-3/4_{10}，\text{即 } -3/2^2{}_{10}$$

用二进制小数又可以表示为：

$$-11_2/2^2{}_{10}，\text{即 } -0.11_2$$

用科学记数法表示为：

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

用规格化的科学记数法表示为：

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

单精度数通常表示为：

$$(-1)^S \times (1+\text{有效位数}) \times 2^{(\text{指数}-127)}$$

从 $-1.1_2 \times 2^{-1}$ 的指数部分减去 127 可得：

$$(-1)^S \times (1+0.1000\ 0000\ 0000\ 0000\ 0000\ 000_2) \times 2^{(126-127)}$$

因此 -0.75_{10} 的单精度二进制表示为：

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
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1位　　8位　　23位

双精度表示为：

$$(-1)^1 \times (1+0.1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2) \times 2^{(1022-1023)}$$

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
1	0	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1位　　11位　　20位

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

32位

下面我们看一个反向的例子。

例题 | 将二进制浮点数转换为十进制浮点数

这个单精度浮点数表示的十进制数是多少？

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
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

答案 | 符号位为1，指数字段为129，尾数字段为 $1\times2^{-2}=1/4$，即0.25。使用基本公式：

$$(-1)^S \times (1+\text{有效位数}) \times 2^{(\text{指数}-\text{偏移值})} = (-1)^1 \times (1+0.25) \times 2^{(129-127)}$$
$$= -1 \times 1.25 \times 2^2$$
$$= -1.25 \times 4$$
$$= -5.0$$

在接下来的几小节中，我们将给出用于浮点加法和乘法的算法。它们的核心是对有效数位部分使用相应的整数运算，但需要额外的工作来处理指数和对结果进行规格化处理。我们首先给出一个十进制算法的直观推导，然后给出更详细的二进制版本，并给出相应图示。

详细阐述 遵循IEEE指导原则，IEEE 754委员会在标准制定20年后进行了改革，以查看应该制定哪些更改（如果有的话）。修订后的标准IEEE 754-2008几乎包含IEEE 754-1985标准的所有内容，并增加了16位格式（“半精度”）和128位格式（“四精度”）。修订后的标准还增加了十进制浮点运算。

详细阐述 为了在保留有效位的情况下增加表示范围，IEEE 754标准之前的一些计算机使用的基数不是2。例如，IBM 360和370大型计算机使用的是基数16。由于将IBM指数改变1相当于将有效数位移动4位，所以基数为16的规格化数的前导0多达3个！因此，这种十六进制数的表示方法意味着必须从有效位数中删除3位，这会对浮点运算的精度带来很大影响。最近的IBM大型机不仅支持早期的十六进制格式，还支持IEEE 754标准。

3.5.4 浮点加法

让我们用科学记数法表示的数的手算加法来说明一下浮点数的加法：$9.999_{10}\times10^{1}+1.610_{10}\times10^{-1}$。假设有效数位中只能保存4位十进制数字，而指数字段只能保存两位十进制数字。

第一步：为了能够对这些数做出正确的加法运算，我们必须将指数较小的数的小数点和指数较大的数的小数点对齐。因此，我们需要处理指数较小的数，即 $1.610_{10}\times10^{-1}$，让它与具有较大的指数的数的指数相同。我们发现一个非规格化的浮点数可以有多种科学记数法的表示形式，从而可以利用该特性完成指数对齐，即：

$$1.610_{10}\times10^{-1}=0.1610_{10}\times10^{0}=0.01610\times10^{1}$$

最右边的数是我们想要的版本，因为它的指数与较大数字的指数相等，即 $9.999_{10}\times10^{1}$。因此，第一步将较小数的有效数位进行右移，直到它的指数变得和较大数的指数一样。但是我们只能表示四位十进制数，所以在移位之后的数是：

$$0.016_{10}\times10^{1}$$

第二步：将两个数的有效数位相加：

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

和为 $10.015_{10} \times 10^{1}$。

第三步：这个和没有用规格化的科学记数法表示，因此要调整为：

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

因此，在加法之后，我们必须对和进行移位，适当地调整指数大小，把它变为规格化的形式。这个例子展示了将和右移一位的情况，但是如果一个数是正数而另一个数是负数，那么得到的和可能有许多前导 0，这时需要进行左移操作。每当指数增大或减小时，我们都必须检测上溢或下溢——也就是说，我们必须确保指数大小没有超过指数字段的表示范围。

第四步：由于我们假定有效位数可能只有四位（不包括符号位），所以我们必须对最后结果进行舍入。在小学学过的算法中，如果右边多余的数在 0 和 4 之间，则直接舍去，如果右边的数在 5 和 9 之间，则舍去后前一位加 1。前面所得的和为：

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

因为小数点右边的第四位数字在 5 到 9 之间，所以舍入到四位有效数位的结果是：

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

请注意，如果我们在舍入的时候运气不好，例如遇到前面各位都是 9 的情况，那么加上 1 的和仍不是规格化的，需要再次执行第三步。

图 3-14 显示了遵循该十进制加法示例的二进制浮点加法的算法。第一步和第二步与刚刚讨论的示例类似：调整指数较小的数的有效数位，让它和另一个较大数的小数点对齐，然后加上两个数的有效数位。第三步对结果进行规格化，并强制检测上溢或下溢。第三步中的上溢和下溢检测取决于操作数的精度。回想一下，指数中全为 0 的表示被保留并用于 0 的浮点表示。而且，指数中全为 1 的表示仅用于标识指定的值和超出正常浮点数范围之外的情况（请参考 3.5.8 节的“详细阐述”）。对于以下示例，请记住，对于单精度，最大指数为 127，最小指数为 −126。

例题 二进制浮点加法

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

答案 让我们首先看一下这两个数用规格化的科学记数法的二进制表示，假设保持 4 位精度：

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}	$= -7/16_{10}$	$= -7/2^{4}{}_{10}$	
	$= -0.0111_{2}$	$= -0.0111_{2} \times 2^{0}$	$= -1.110_{2} \times 2^{-2}$

现在我们按照如下的算法执行。

第一步：将指数较小的数（$-1.11_{2} \times 2^{-2}$）的有效数位右移，直到其指数与较大的数相同：

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

第二步：将有效数位相加：

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

第三步：对和进行规格化，并检测上溢和下溢：

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

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

由于 $127 \geqslant -4 \geqslant -126$，没有上溢或下溢。（带偏移值的指数为 $-4+127$，即 123，它在最小指数 1 和未保留的带偏移值的最大指数 254 之间。）

第四步：对和进行舍入：

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

和已经完全符合 4 位精度，所以无须再做舍入。

所以和是

$$1.000_2 \times 2^{-4} = 0.0001000_2 = 0.0001_2$$

$$= 1/2^4{}_{10} = 1/16_{10} = 0.0625_{10}$$

这就是 0.5_{10} 和 -0.4375_{10} 的和。

许多计算机用专门硬件来尽可能快地运行浮点运算。图 3-15 描绘了浮点加法硬件的基本结构。

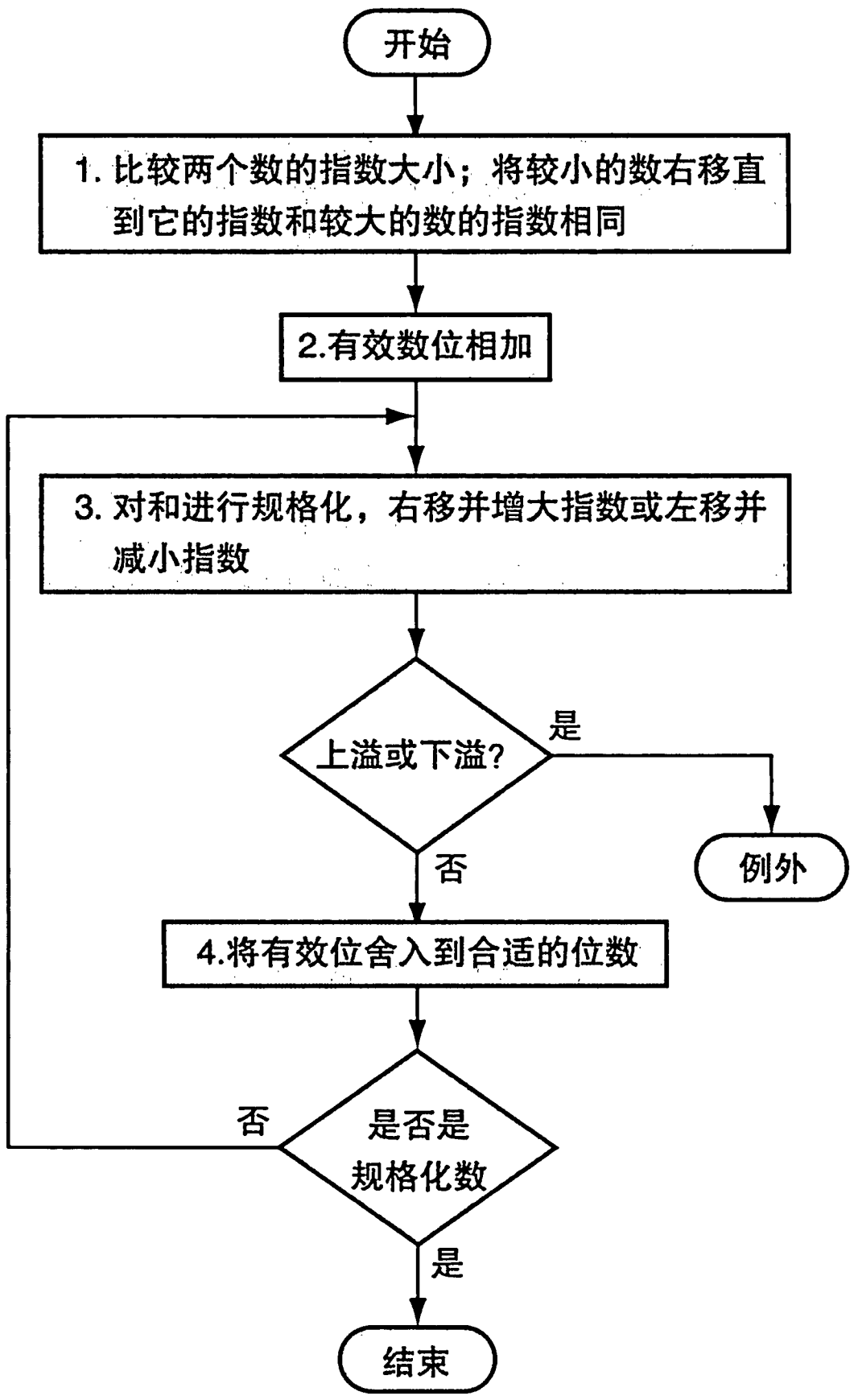


图 3-14　浮点加法。正常步骤是步骤 3 和 4 执行一次即可，但如果舍入未规格化的和，我们必须重复步骤 3

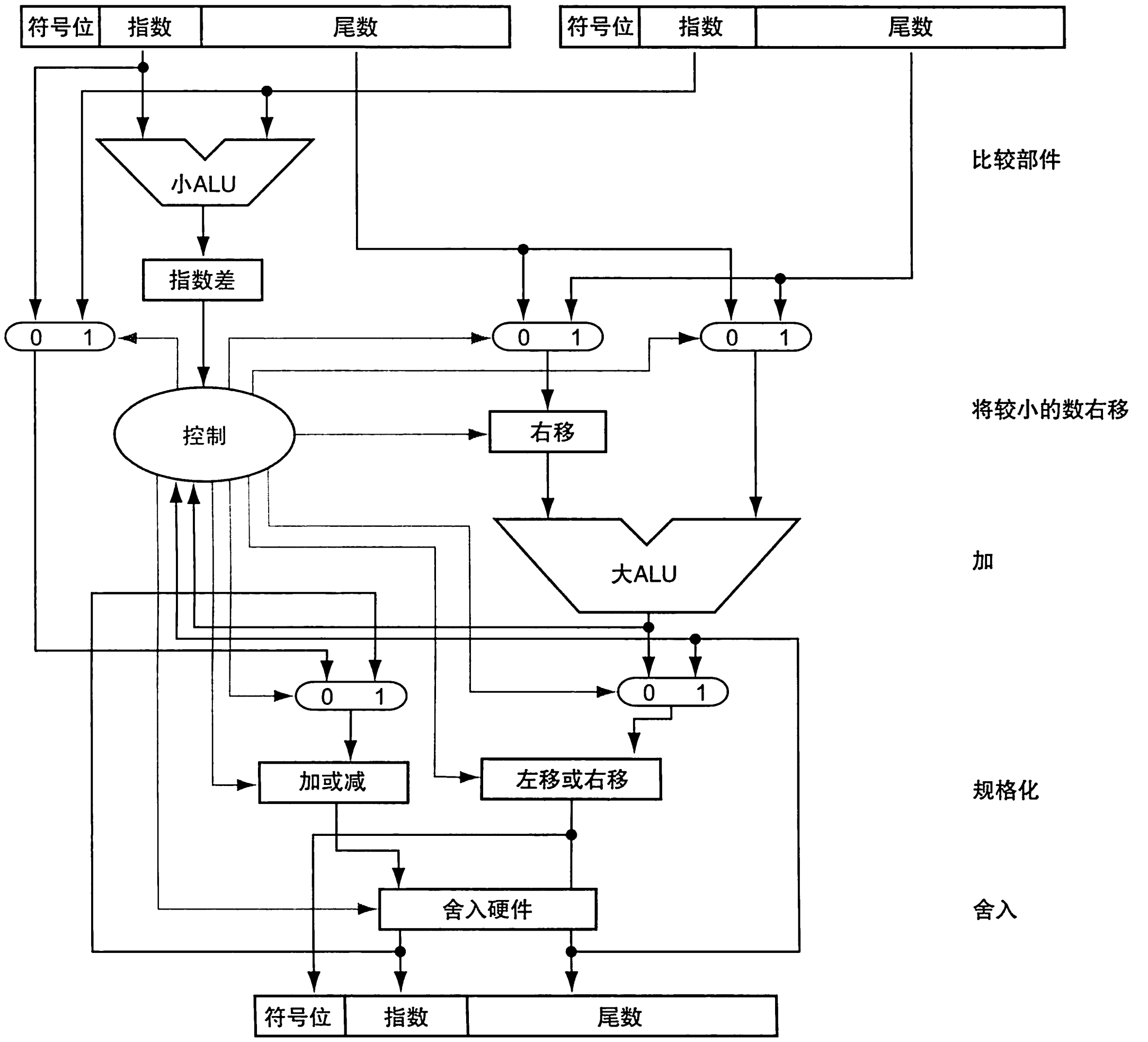


图 3-15　专用于浮点加法的算术单元结构图。图 3-14 的步骤从上到下对应于每个部分。首先，使用小的 ALU 将一个操作数的指数减去另一个操作数的指数，以确定哪一个更大以及大多少。这个差值控制着三个多选器；按从左到右的顺序，选择较大的指数、较小加数的有效数位以及较大加数的有效数位。较小加数的有效数位进行右移，然后使用大 ALU 将两数的有效数位相加。规格化步骤对总和进行左移或右移，并相应地减少或增加指数。舍入产生的结果，但有可能需要再次规格化以产生最终结果

3.5.5　浮点乘法

现在我们已经介绍完了浮点加法，下面开始介绍浮点乘法。我们从手工计算以十进制科学记数法表示的浮点乘法开始：$1.110_{10} \times 10^{10} \times 9.200_{10} \times 10^{-5}$。假设我们只能保存四位有效数位和两位指数字段。

第一步：和加法不同，我们通过简单地将操作数的指数相加来计算积的指数：

$$\text{新的指数} = 10 + (-5) = 5$$

让我们用移码来表示指数并确保获得相同的结果：$10 + 127 = 137$ 和 $-5 + 127 = 122$，所以

$$\text{新的指数} = 137 + 122 = 259$$

这个结果对于8位指数字段来说太大了，所以肯定有些地方出错了！问题在于偏移值，因为我们在进行指数相加的同时也进行了偏移值相加：

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

因此，当我们将带偏移值的数相加时，为了得到正确的带偏移值的总和，我们必须从总和中减去一个偏移值：

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

而5就是我们最初计算的指数。

第二步：下面开始有效数位部分的相乘：

$$\begin{array}{r} 1.110_{10} \\ \times\ 9.200_{10} \\ \hline 0\ 000 \\ 00\ 00 \\ 222\ 0 \\ 9990 \\ \hline 10212000_{10} \end{array}$$

每个操作数的小数点右侧有三位数字，因此乘积的小数点应该放在从右数第6位有效位前：

$$10.212000_{10}$$

如果我们只能保留小数点右侧三位数字，则积为 10.212×10^5。

第三步：因为得到的乘积是非规格化的，所以要将其规格化：

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

因此，在乘法之后，乘积需要右移一位得到规格化的结果，同时指数加1。此时，我们可以检测上溢和下溢。如果两个操作数都很小，也就是说，如果两者都具有较小的负指数时，则可能发生下溢。

第四步：我们假设有效数位只有四位数字（不包括符号位），所以必须将乘积舍入。乘积：

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

舍入到四位有效数位是：

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

第五步：积的符号取决于原始操作数的符号。如果它们符号相同，则积的符号为正；否则，积的符号为负。因此，积为：

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

在加法算法中，和的符号是由有效位数相加的结果来确定的，但是在乘法中，操作数的符号决定了乘积的符号。

同加法类似，如图3-16所示，二进制浮点数的乘法也与我们刚刚完成的十进制乘法的步骤非常相似。我们首先要将两数的带偏移值的指数相加并确保减去一个偏移值，以得到乘积的新的正确指数。接下来是有效数位的乘法，紧跟的是可选的规格化步骤。然后是检查指数的大小是否上溢或下溢，再对积进行舍入。如果舍入导致需要再次规格化，就要再次检查指数大小。最后，如果操作数的符号不同（积为负），则将积的符号位设置为1；如果它们相同（积为正），则将积的符号位设置为0。