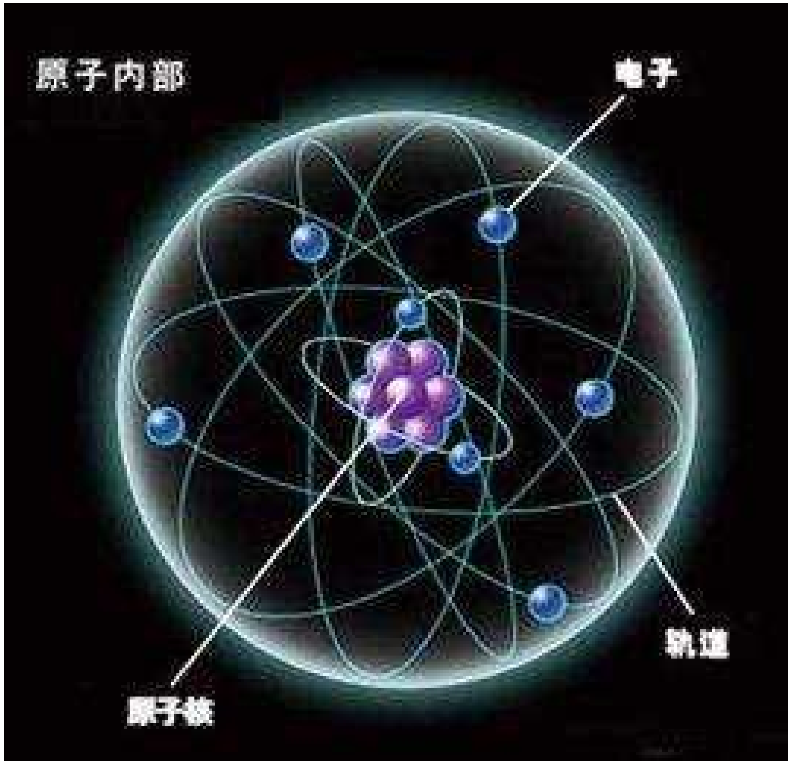
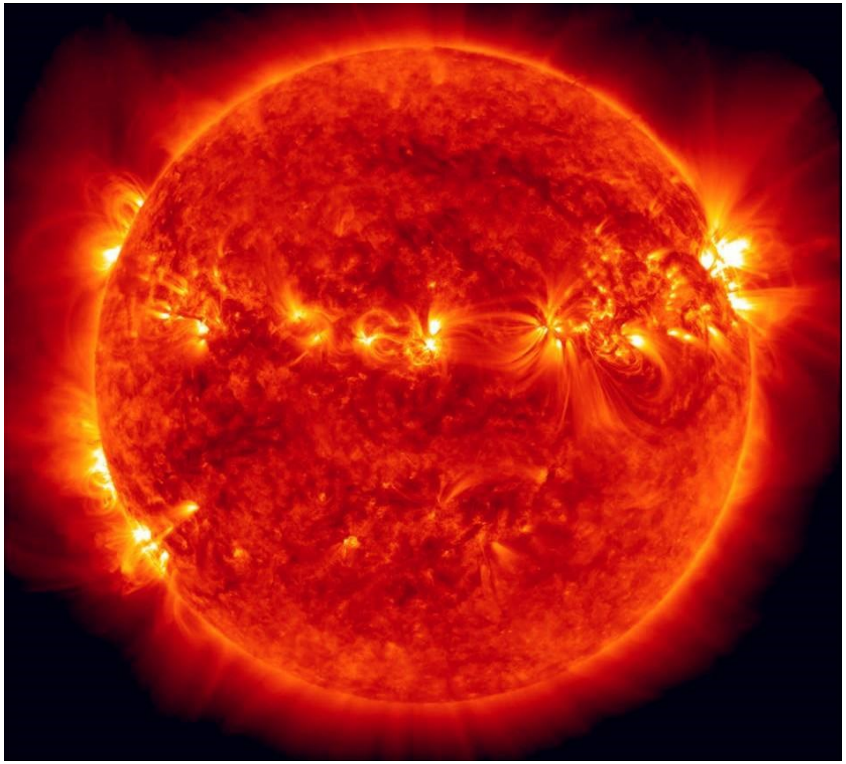
处理器体系结构

第三章 处理器中的数值运算B
--浮点数的表示与运算
(Micro-processor Architecture)

为什么需要浮点数?





电子：9.1e-31kg

太阳：1.99e30kg

目录

浮点数的表示与运算

1. 浮点数的表示

2. 浮点数运算

1. 浮点数的表示

如何表示非常小或非常大的非整数?

科学计数法:

-2.34×10^{56} ← 规格化形式

$+0.002 \times 10^{-4}$ ← 非规格化形式

$+987.02 \times 10^{9}$ ← 非规格化形式

二进制形式:

$\pm 1.xxxxxxx_2 \times 2^{yyyy}$

->对应C语言中的float和double数据类型

1.浮点数的表示--754标准

符合IEEE 754-1985标准

- 解决了表示方法的分歧问题
- 增加了代码的可移植性
- 现在已被普遍接受

两种表示方法

- 单精度（32位）
- 双精度（64位）

1.浮点数的表示--IEEE浮点数格式

S	指数	尾数
单精度：	8位	23位
双精度：	11位	52位

有效数

$$x = (-1)^S \times \boxed{(1 + \text{尾数})} \times 2^{(\text{指数}-\text{偏量})}$$

S：符号位(0为正，1为负)

简化数据交换、简化浮点算法、提高数据精度

规格化的有效数：二进制小数点左边有且仅有一位1的形式 1.0≤|有效数|＜2.0

小数点前面一位始终为1，所以不需要显式表示

问题：指数可能是正数或负数，然而补码形式的负数看起来更大，如何解决？

解决方案：指数采用移码形式　　实际表示的指数=原码形式的指数-偏量

单精度：偏量=127　　双精度：偏量=1023

1.浮点数的表示--浮点数表示范围

NOTE：对于浮点数的指数，全0和全1是被保留的

最小值：

	单精度	双精度
指数	0000 0001	000 0000 0001
尾数	全零	全零
表示的浮点数	$\pm 1.0\times 2^{-126}\approx \pm 1.2\times 10^{-38}$	$\pm 1.0\times 2^{-1022}\approx \pm 2.2\times 10^{-308}$

最大值：

	单精度	双精度
指数	1111 1110	111 1111 1110
尾数	全1	全1
表示的浮点数	$\approx \pm 2.0\times 2^{+127}\approx \pm 3.4\times 10^{+38}$	$\approx \pm 2.0\times 2^{+1023}\approx \pm 1.8\times 10^{+308}$

1.浮点数的表示--浮点数精度

单精度数的尾数为23位，双精度数的尾数为52位

单精度数的精度：2^{-23}

等效于$23\times\log_{10}2\approx23\times0.3\approx6$位十进制精度

双精度数的精度：2^{-52}

等效于$52\times\log_{10}2\approx52\times0.3\approx16$位十进制精度

1.浮点数的表示

例1：分别用单精度和双精度浮点数表示-0.75

$-0.75=(-1)^1\times 1.1_2\times 2^{-1}$

S=1

Fraction为$1000\ldots00_2$

指数为-1+Bias

单精度：偏量为127，所以为 $-1+127=126=0111\ 1110_2$

双精度：偏量为1023，所以为 $-1+1023=1022=011\ 1111\ 1110_2$

单精度：1011111101000...00

双精度：10111111111101000...00

1.浮点数的表示

例2：单精度浮点数11000000101000...00表示多少?

S = 1

Fraction = $01000...00_2$

Exponent = $10000001_2 = 129$

$$x = (-1)^1 \times (1 + 0.01_2) \times 2^{(129-127)}$$
$$= (-1) \times 1.25 \times 2^2$$
$$= -5.0$$

1.浮点数的表示--非规格化数和零

指数为全0时，隐藏位也变为0，表示非规格化数：

1-偏量

$$x=(-1)^S\times(0+尾数)\times 2^{?}$$

比规格化数要小：允许逐渐的下溢出，精度也逐渐降低

当指数和尾数全为零时，表示零：

$$x=(-1)^S\times(0+0)\times 2^{-偏量}=\pm 0.0$$

零有两种表示形式

1.浮点数的表示--无穷数和非数(NaN)

当指数为全1，且尾数为全0时，表示±无穷：

可以用于后续的计算，免去溢出检查

当指数为全1，且尾数不全为0时，表示非数：

NaN：Not a Number

表示非法或未定义的结果，比如0.0/0.0

可以被用于后续的计算

1.浮点数的表示--小结

符合IEEE 754-1985标准的浮点数：

符号位	指数	尾数
1位	8位	23位
1位	11位	52位

单精度： 1位 8位 23位

双精度： 1位 11位 52位

127、1023

$$x=(-1)^{\text{符号位}}\times(1+\text{尾数})\times 2^{\text{指数(原码)}-\text{偏量}}$$

指数	全零	全零	00...01≤指数≤11...10	全一	全一
尾数	全零	0<尾数<1	0≤尾数<1	全零	非零
表示	±0	非规格化数	规格化数	±∞	NaN
单精度范围	\	$\pm\{2^{-23}\sim(1-2^{-23})\}\times2^{-126}$	$\pm1\times2^{-126}\sim\pm(2-2^{-23})\times2^{+127}$	\	\
双精度范围	\	$\pm\{2^{-52}\sim(1-2^{-52})\}\times2^{-1022}$	$\pm1\times2^{-1022}\sim\pm(2-2^{-52})\times2^{+1023}$	\	\

课后作业

1. 以下单精度浮点数分别表示多少?

10111111111000...0

00000000001000...0

2. 十进制数-33.75的单精度浮点数是多少?

目录

浮点数的表示与运算

1. 浮点数的表示

2. 浮点数运算

2. 浮点数运算--加法

考虑一个4位十进制数加法：

$$9.999 \times 10^{1} + 1.610 \times 10^{-1}$$

1.对齐

$$9.999 \times 10^{1} + 0.016 \times 10^{1}$$

2.尾数相加

$$= 10.015 \times 10^{1}$$

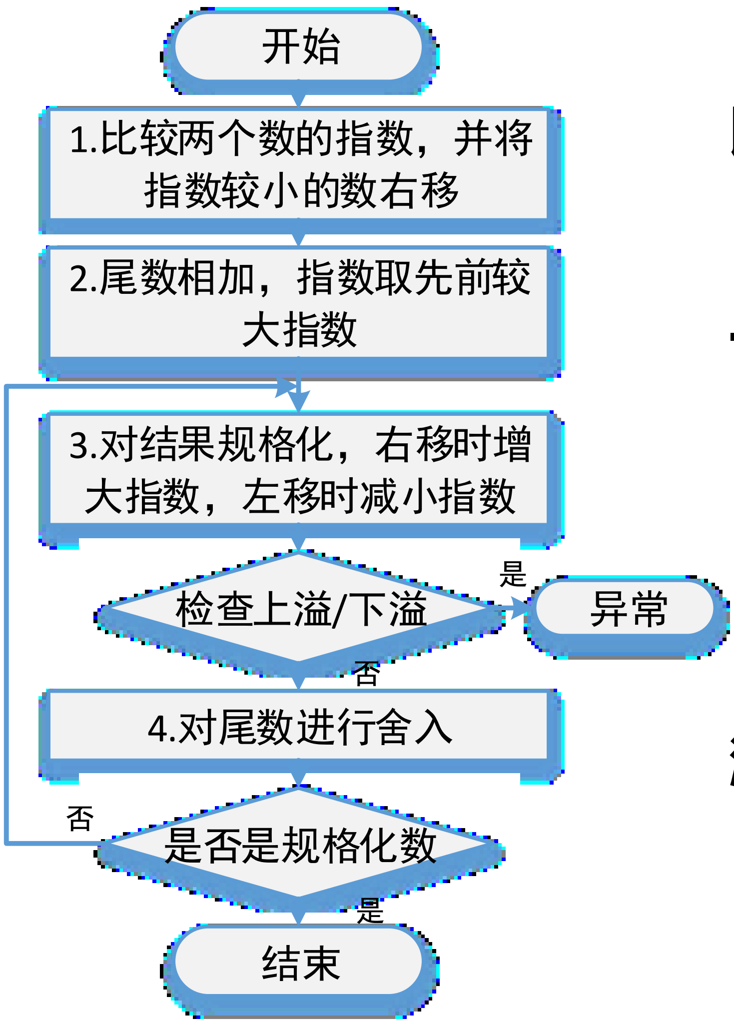
3.将结果规格化，并检查上溢/下溢

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

4.四舍五入，并按需求再次规格化

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

2. 浮点数运算--加法器的硬件实现



比整数加法器复杂很多

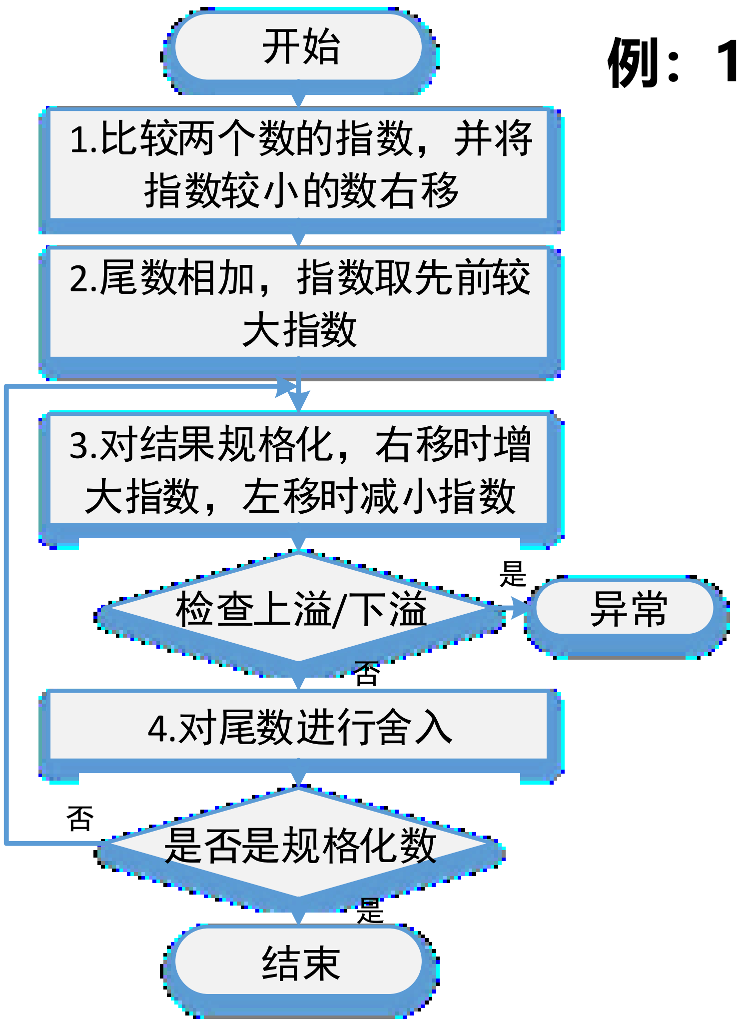
一个时钟周期内完成所有操作会不利于整体设计

- **浮点数加法的时间比整数加法长很多**
- **增加时钟周期长度会降低所有指令的执行速度**

浮点数加法往往需要数个时钟周期来完成操作

- **更便于流水**

2. 浮点数运算--加法器的硬件实现



例：$1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)

$$1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$$

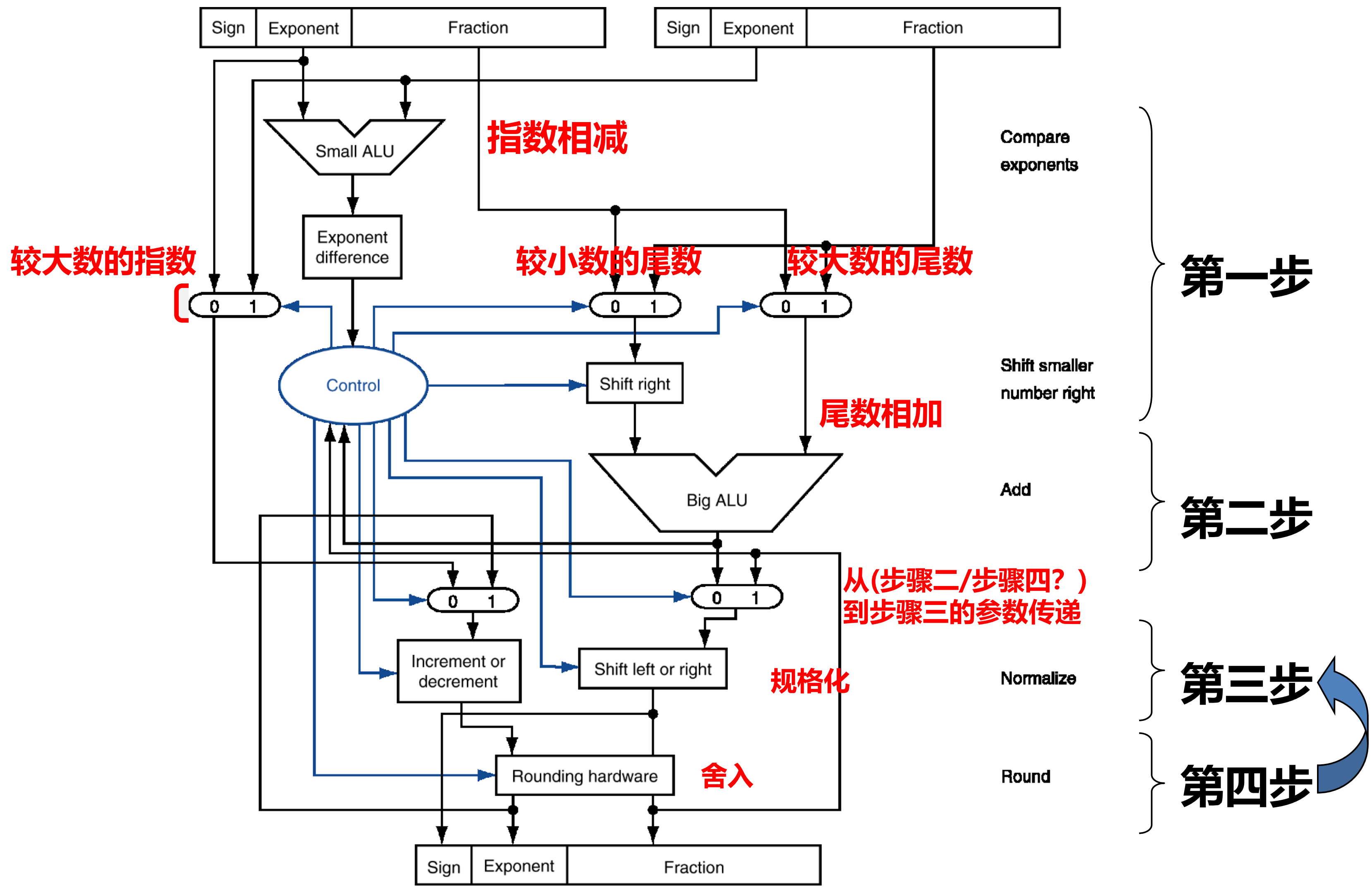
$$=0.001_2 \times 2^{-1}$$

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

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

$$=0.0625$$

2. 浮点数运算--加法器的硬件实现



2. 浮点数运算--乘法

4位十进制数乘法：

$$1.110 \times 10^{10} \times 9.200 \times 10^{-5}$$

1.指数相加

10+-5=5

2.尾数相乘

$= 10.212 \times 10^{5}$

3.将结果归一化，并检查上溢/下溢

$= 1.0212 \times 10^{6}$

4.四舍五入，并重新归一化

$= 1.021 \times 10^{6}$

5.操作数的符号决定了结果的符号

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

2. 浮点数运算--乘法

4位二进制数乘法：(对应十进制乘法0.5 × –0.4375)

$$1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$$

1.指数相加

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

2.尾数相乘

$1.000_2 \times 1.110_2 = 1.110_2$ -> $1.110_2 \times 2^{-3}$

3.将结果归一化，并检查上溢/下溢

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

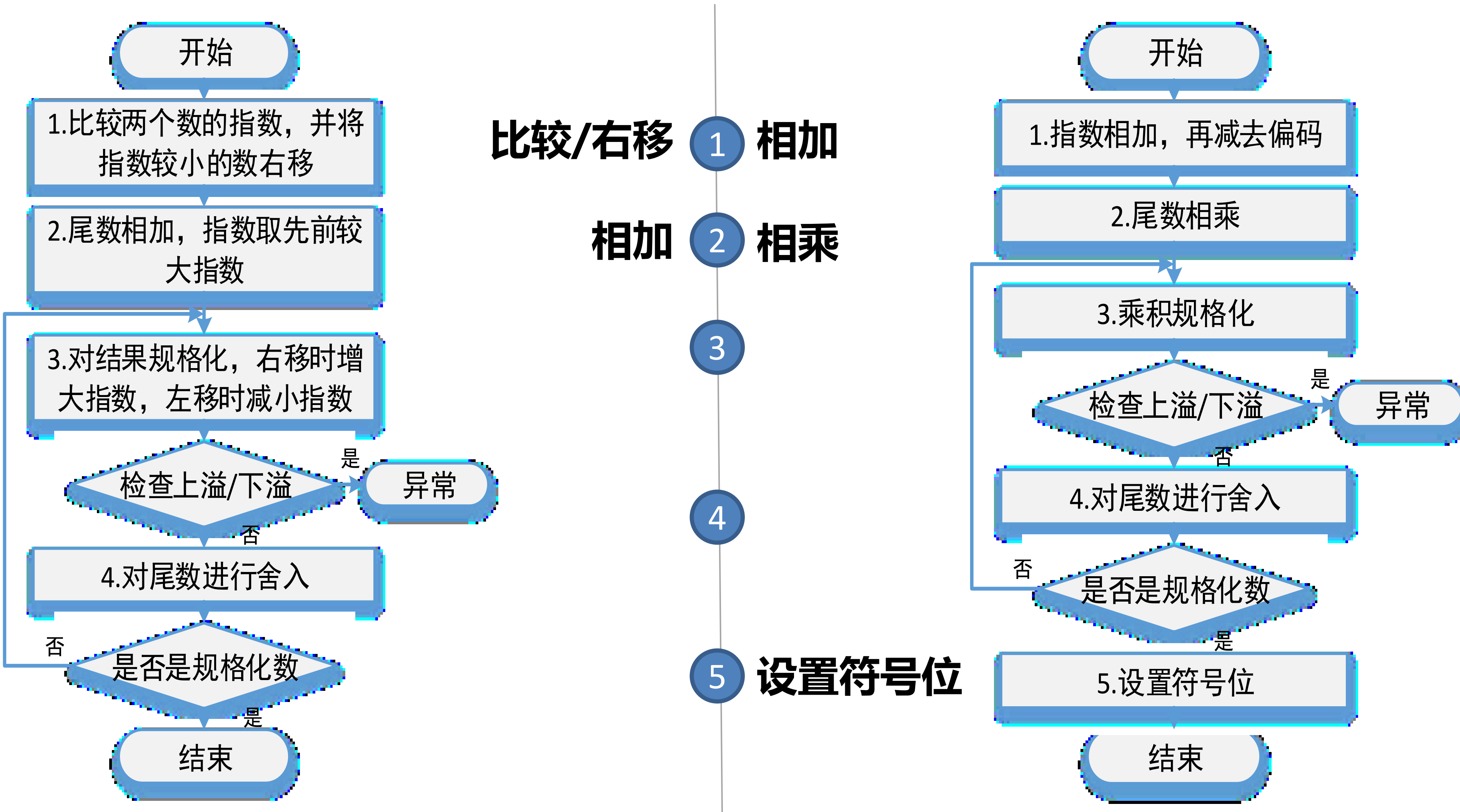
4.四舍五入，并重新归一化

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

5.操作数的符号决定了结果的符号

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

2. 浮点数运算--浮点加与浮点乘的对比



2. 浮点数运算--MIPS对浮点数运算的支持

MIPS对浮点数运算支持：加、减、乘、除、求倒数、开根…

*这些运算通常都需要好几个周期完成，可以被流水线

*通常，整数运算不使用浮点数硬件，浮点数运算也不使用整数硬件

浮点数采用专用的寄存器： $f0~$f31　　每个寄存器为32位(也有64位版本)

部分浮点数操作指令：（“c1”：表示协处理器1)

分类	指令	含义	举例
算术	add.s、sub.s、mul.s、div.s	单精度加、减、乘、除	add.s $f2, $f4,$f6
算术	add.d、sub.d、mul.d、div.d	双精度加、减、乘、除	add.d $f2, $f4, $f6
传输	lwc1、swc1、ldc1、sdc1	$f与内存存/取数据	swc1 $f1, 100($s2)
跳转	bc1t、bc1f	cond为1/0则跳转	bc1t 25
判定	c.xx.s、c.xx.d (xx: eq,lt,le)	成立，则将cond至1，否则至0	c.lt.s $f2, $f4

NOTE：双精度类指令只能使用$s双数寄存器(共16个)

2. 浮点数运算

例：完成华氏度到摄氏度的转换

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

Fahr存于$f12，结果存放在$f0，

其它常数在内存中

```
f2c: lwc1  $f16, const5($gp)
     lwc1  $f18, const9($gp)
     div.s $f16, $f16, $f18
     lwc1  $f18, const32($gp)
     sub.s $f18, $f12, $f18
     mul.s $f0,  $f16, $f18
     jr    $ra
```

2. 浮点数运算

例：对32*32的双精度浮点数矩阵进行运算　　X=X+Y×Z

```
void mm (double x[][],
  double y[][], double z[][]) {
  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)
        x[i][j] = x[i][j]
                + y[i][k] * z[k][j];
}
```

X、Y、Z的首地址在$a0、$a1、$a2

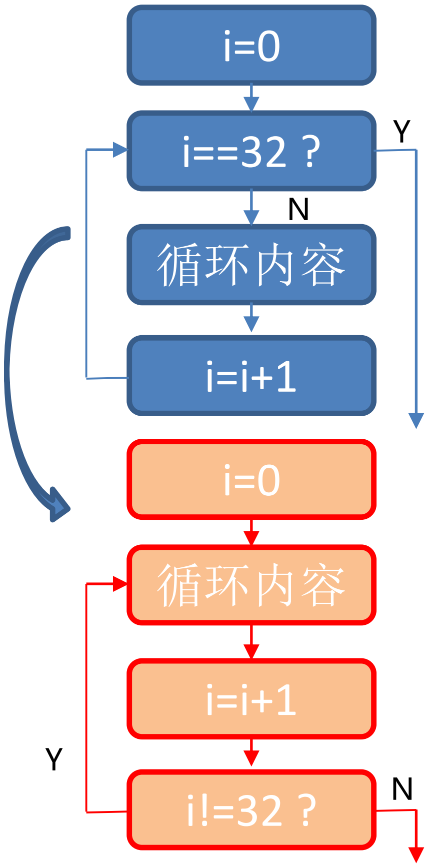
i、j、k存于$s0、$s1、$s2

```
    li    $t1, 32          #$t1 = 32
    li    $s0, 0           #i = 0
L1:beq    $s0, $t1, E1#如果i=32成立，则跳转E1
    li    $s1, 0           #j = 0
L2:beq    $s1, $t1, E2#如果j=32成立，则跳转E2
    li    $s2, 0           #k = 0
L3:beq    $s2, $t1, E3#如果k=32成立，则跳转E3
    sll   $t2, $s0, 5      #$t2=i*32
    addu  $t2, $t2, $s1    #$t2=i*32+j
    sll   $t2, $t2, 3      #地址乘8
    addu  $t2, $a0, $t2    #x[i][j]的地址
    l.d   $f4, 0($t2)      #读取双精度数x[i][j]
    sll   $t0, $s2, 5      #$t0=k*32
    addu  $t0, $t0, $s1    #$t0=k*32+j
    sll   $t0, $t0, 3      #地址乘8
    addu  $t0, $a2, $t0    #$z[k][j]的地址
    l.d   $f16, 0($t0)     #读取双精度数$z[k][j]
    sll   $t0, $s0, 5          #$t0=i*32
    addu   $t0, $t0, $s2       #$t0=i*32+k
    sll    $t0, $t0, 3         #地址乘8
    addu   $t0, $a1, $t0       #y[i][k]的地址
    l.d    $f18, 0($t0)        #读取双精度数y[i][k]
    mul.d $f16, $f18, $f16#$f16=y[i][k]*z[k][j]
    add.d $f4, $f4, $f16       #$f4=x[i][j]+$f16
    s.d    $f4, 0($t2)         #x[i][j]=$f4
    addiu $s2, $s2, 1          #k=k+1
    J  L3
E3:addiu $s1, $s1, 1           #j=j+1
    J  L2
E2:addiu $s0, $s0, 1           #i=i+1
    J  L1
E1:…
```

2. 浮点数运算

优化：

beq和J可以合并，从而减少指令数量



取x[i][j]和k没有关系，可以提前

```
    li    $t1, 32          #$t1 = 32
    li    $s0, 0           #i = 0
L1:beq    $s0, $t1, E1#如果i=32成立，则跳转E1
    li    $s1, 0           #j = 0
L2:beq    $s1, $t1, E2#如果j=32成立，则跳转E2
    li    $s2, 0           #k = 0
L3:beq    $s2, $t1, E3#如果k=32成立，则跳转E3
    sll   $t2, $s0, 5      #$t2=i*32
    addu  $t2, $t2, $s1    #$t2=i*32+j
    sll   $t2, $t2, 3      #地址乘8
    addu  $t2, $a0, $t2    #x[i][j]的地址
    l.d   $f4, 0($t2)      #读取双精度数x[i][j]
    sll   $t0, $s2, 5      #$t0=k*32
    addu  $t0, $t0, $s1    #$t0=k*32+j
    sll   $t0, $t0, 3      #地址乘8
    addu  $t0, $a2, $t0    #$z[k][j]的地址
    l.d   $f16, 0($t0)     #读取双精度数$z[k][j]
    sll   $t0, $s0, 5          #$t0=i*32
    addu   $t0, $t0, $s2       #$t0=i*32+k
    sll    $t0, $t0, 3         #地址乘8
    addu   $t0, $a1, $t0       #y[i][k]的地址
    l.d    $f18, 0($t0)        #读取双精度数y[i][k]
    mul.d $f16, $f18, $f16#$f16=y[i][k]*z[k][j]
    add.d $f4, $f4, $f16       #$f4=x[i][j]+$f16
    s.d    $f4, 0($t2)         #x[i][j]=$f4
    addiu $s2, $s2, 1          #k=k+1
     J   L3
E3:addiu $s1, $s1, 1           #j=j+1
     J   L2
E2:addiu $s0, $s0, 1           #i=i+1
     J   L1
E1:…
```

2. 浮点数运算

例：对32*32的双精度浮点数矩阵进行运算　　X=X+Y×Z

```
void mm (double x[][],
  double y[][], double z[][]) {
  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)
     x[i][j] = x[i][j]
          + y[i][k] * z[k][j];
}
```

X、Y、Z的首地址在$a0、$a1、$a2

i、j、k存于$s0、$s1、$s2

```
    li    $t1, 32         #$t1 = 32
    li    $s0, 0          #i = 0
L1:li    $s1, 0          #j = 0
L2:li    $s2, 0          #k = 0
    sll   $t2, $s0, 5     #$t2=i*32
    addu  $t2, $t2, $s1   #$t2=i*32+j
    sll   $t2, $t2, 3     #地址乘8
    addu  $t2, $a0, $t2   #x[i][j]的地址
    l.d   $f4, 0($t2)     #读取双精度数x[i][j]
L3:sll   $t0, $s2, 5     #$t0=k*32
    addu  $t0, $t0, $s1   #$t0=k*32+j
    sll   $t0, $t0, 3     #地址乘8
    addu  $t0, $a2, $t0   #$z[k][j]的地址
    l.d   $f16, 0($t0)    #读取双精度数$z[k][j]
    sll   $t0, $s0, 5        #$t0=i*32
    addu  $t0, $t0, $s2      #$t0=i*32+k
    sll   $t0, $t0, 3        #地址乘8
    addu  $t0, $a1, $t0      #y[i][k]的地址
    l.d   $f18, 0($t0)       #读取双精度数y[i][k]
    mul.d $f16, $f18, $f16#$f16=y[i][k]*z[k][j]
    add.d $f4, $f4, $f16     #$f4=x[i][j]+$f16
    addiu $s2, $s2, 1        #k=k+1
    bne   $s2, $t1, L3       #如果k!=32成立，则跳转L3
    s.d   $f4, 0($t2)        #x[i][j]=$f4
    addiu $s1, $s1, 1        #j=j+1
    bne   $s1, $t1, L2       #如果j!=32成立，则跳转L2
    addiu $s0, $s0, 1        #i=i+1
    bne   $s0, $t1, L1       #如果i!=32成立，则跳转L1
```

循环内容得以缩短

2. 浮点数运算--关于舍入与精度问题

浮点数通常只是一个数的近似：只能表示有限个数，以单精度为例，可表示255*2^24+3个数：

±0、非规格化数：2^24

规格化数：254*2^24

±∞

NaN

用有限的浮点数表示无穷多的连续实数，必然会产生误差！

误差的衡量：尾数最低位ulp(unit in the last place)，表示目标数与浮点数之间的误差

2. 浮点数运算--关于舍入与精度问题

IEEE 754标准支持不同的舍入方式：向上舍入、向下舍入、截断舍入、向靠近的偶数舍入

1. 向上(round toward positive infinity)：1.23->2、-1.23->-1
2. 向下(round toward negative infinity)：1.23->1、-1.23->-2
3. 截断(round toward zero)：1.23->1、-1.23->-1

* 四舍五入?：1.xx：xx<=49则舍去，xx>=50则进位 (50总是向上进位会产生概率偏差！IEEE 754不支持)

4. 向最靠近的偶数舍入： 0.50->0、1.50->2、2.50->2

->Java唯一支持的舍入方式，从统计学来看，小数点前面是奇数或偶数的概率各为50%

翻译不准确，容易误解：0.9->0?、1.2->2? No!
(确切叫作round to nearest, ties to even)

round to nearest, ties to even的准确理解：

x<=0.5 → 0
0.5<x<1.5 → 1
1.5<=x<=2.5 → 2
2.5<x<3.5 → 3

2. 浮点数运算--关于舍入与精度问题

当处理器处理二进制数时，采用保护位(guard)、舍入位(round) 进行控制：

	10.00	10.01	10.10	10.11	-10.00	-10.01	-10.10	-10.11
round toward positive infinity	10	11	11	11	-10	-10	-10	-10
round toward negative infinity	10	10	10	10	-10	-11	-11	-11
round toward zero	10	10	10	10	-10	-10	-10	-10
round to nearest, ties to even	10	10	10 or 11?	11	-10	-10	-10 or -11?	-11

保护位、舍入位的后面有更多数据怎么处理?

粘贴位(sticky)：舍入位的右边非零，则置一

考虑：$1.01+1.10\times2^3 \quad =1.10\boxed{10}1\times2^3$ 分别放入保护位和舍入位

round to nearest, ties to even：

若“10”后面全为零，则”ties to even”，不进位

若“10”后面不全为零，则”round to nearest”，进位

$->1.11\times2^3$

2. 浮点数运算--移位与除法

关于移位：对于整数，左移一位等效于乘以2，那么右移一位呢?

对于无符号数等效于除以2

对于有符号数，需要复制符号位

例如： e.g., –5 / 4

$11111011_2 >> 2 = 11111110_2 = -2$

Rounds toward –∞

c.f. $11111011_2 >>> 2 = 00111110_2 = +62$

关于结合律：对于浮点数运算，结合律可能会失效

例如：

		(x+y)+z	x+(y+z)
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

原因：浮点数是实际数的近似

2. 浮点数运算—浮点数运算的精度问题

Intel：“小问题”，9个有效数以后才会出错、出错的概率为90亿分之一!

解决问题只需花费几十万美元

IBM：大问题！每天15分钟运算+每秒5000次除法=24天出现一次错误!

Intel召回Bug芯片，花费5亿美元!

本章小结

整数运算及其硬件电路实现

整数加减法、乘法、除法、MIPS对乘除法的支持

浮点数的表示和浮点数运算

IEEE 754标准、浮点数范围、硬件实现、舍入与精度、

MIPS浮点数运算指令