

Computer Organization and Design

Arithmetic for Computers

Part III Floating-Point Number

Jiang Zhong
zhongjiang@cqu.edu.cn

浮点数的表示与计算

- 浮点数的表示
- IEEE754标准
- 浮点数的加减法
- 浮点数的乘除法

几个物理学常数

电子质量 $9.10938215 \times 10^{-31} \text{ kg}$

普朗克常数 $6.62606896 \times 10^{-34}$

阿伏加德罗常数 (摩尔常数) 6.023×10^{23}

.....

**请大家思考计算机中32位或者64位的二进制定点小数或者
定点整数是否能够表示上述物理常数**

假设一个科学计算中包含了电子质量、普朗克常数，那么采用32位的整数运算的计算机，以下哪种说法正确：

- ☐ A 寄存器可以精确地表示这些数字
- ☒ B 不能使用寄存器精确表示和计算
- ☒ C 可以利用多条整数指令编写程序完成上述计算
- ☐ D 利用该计算机无法处理此类计算问题

提交

浮点数的表示方法（10进制推广到任意进制）

$$9 \times 10^{-28} = 0.9 \times 10^{-27}$$

$$2 \times 10^{33} = 0.2 \times 10^{34}$$

$$\text{—— } N = 10^E \cdot M \quad (\text{十进制表示})$$

任意一个十进制数 N 可以写成

$$N = 10^E \times M$$

计算机中一个**任意进制数** N 可以写成

$$N = R^e \times m$$

m : **尾数**，是一个纯小数。

e : 浮点的**指数**，是一个整数。

R : **基数**，对于二进计数值的机器是一个常数，一般规定
 R 为2，8或16。

一个机器浮点数由阶码和尾数及其符号位组成：

尾数：用定点小数表示，给出有效数字的位数，
决定了浮点数的表示精度；

阶码：用定点整数形式表示，指明小数点在数据中的位置，
决定了浮点数的表示范围。

E_s	$E_1 E_2 \dots E_m$	M_s	$M_1 M_2 \dots M_n$
阶符	阶码	数符	尾数

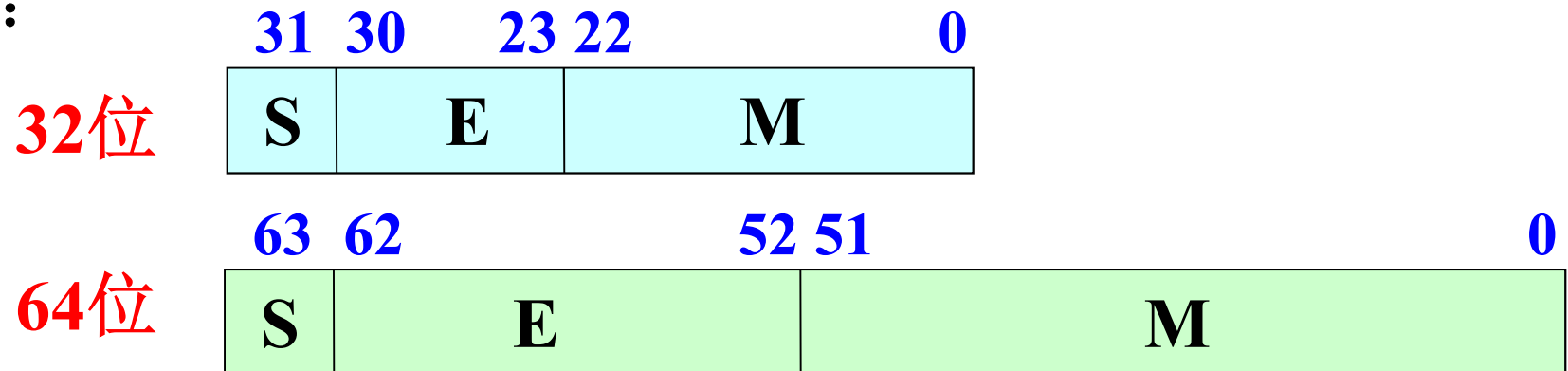
(2) 浮点数的标准格式

$$(N = R^{e.m})$$

为便于软件移植，使用 IEEE（电气和电子工程师协会）标准

IEEE754 标准：尾数用原码；阶码用变形移码；基为2

- 按照 IEEE754 的标准，32位浮点数和64位浮点数的标准格式为：



S——尾数符号，0正1负；

M——尾数，纯小数表示，小数点放在尾数域的最前面。
采用原码表示。

E——阶码，采用“移码”表示（+127，+1023）；

阶符采用隐含方式，即采用“移码”方法来表示正负指数。

(3) 浮点数的规格化表示

浮点数是数学中实数的子集合，由一个纯小数乘上一个指数值来组成。

一个浮点数有不同的表示：

0.5; 0.05×10^1 ; 0.005×10^2 ; 50×10^{-2}

为提高数据的表示精度，需做规格化处理。

浮点数的规格化

规格化目的:

为了提高数据的表示精度

为了数据表示的唯一性

什么是规格化? 如果尾数为R进制的规格化:

绝对值大于或等于 $1/R$

二进制原码的规格化数的表现形式:

正数 0.1xxxxxx

负数 1.1xxxxxx

补码尾数的规格化的表现形式:

尾数的最高位与符号位相反

正数 0.1xxxxxx

负数 1.0xxxxxx

规格化处理

在计算机内，其纯小数部分被称为浮点数的尾数，对非 0 值的浮点数，要求尾数的绝对值必须 $\geq 1/2$ ，即尾数域的最高有效位应为 1，称满足这种表示要求的浮点数为规格化表示：

0.1000101010

把不满足这一表示要求的尾数，变成满足这一要求的尾数的操作过程，叫作浮点数的规格化处理，通过尾数移位和修改阶码实现。

隐藏位技术

既然非 0 值浮点数的尾数数值最高位必定为 1，则在保存浮点数到内存前，通过尾数左移，强行把该位去掉，用同样多的尾数位就能多存一位二进制数，有利于提高数据表示精度，称这种处理方案使用了隐藏位技术。

$$0.1100010 \rightarrow 1.100010$$

当然，在取回这样的浮点数到运算器执行运算时，必须先恢复该隐藏位。

(4) 规格化浮点数的真值

32位浮点数格式:

31	30	23	22	0
S	E	M		

移码定义: $[x]_{\text{移}} = x_0 x_1 x_2 \cdots x_n$
 $= 2^n + x$ $-2^n \leq x < 2^n$

一个规格化的32位浮点数 x 的真值为:

$$x = (-1)^s \times (1.M) \times 2^{E-127} \quad e = E - 127$$

一个规格化的64位浮点数 x 的真值为:

$$x = (-1)^s \times (1.M) \times 2^{E-1023}$$

这里 e 是真值, E 是机器数

例：若浮点数 x 的二进制存储格式为 $(41360000)_{16}$ ，求其32位浮点数的十进制值。

解： **0100,0001,0011,0110,0000,0000,0000,0000**

数符： **0**

阶码： **1000,0010**

尾数： **011,0110,0000,0000,0000,0000**

指数 $e = \text{阶码} - 127 = 10000010 - 01111111 = 00000011 = (3)_{10}$

包括隐藏位1的尾数：

$$1.M = 1.011\ 0110\ 0000\ 0000\ 0000\ 0000 = 1.011011$$

于是有 $x = (-1)^s \times 1.M \times 2^e$

$$= + (1.011011) \times 2^3 = + 1011.011 = (11.375)_{10}$$

例：将十进制数20.59375转换成32位浮点数的二进制格式来存储

解：首先分别将整数和分数部分转换成二进制数：

$$20.59375 = 10100.10011$$

然后移动小数点，使其在第1，2位之间

$$10100.10011 = 1.010010011 \times 2^4 \quad e=4$$

于是得到： $e = E - 127$

$$S=0, \quad E=4+127=131=1000,0011, \quad M=010010011$$

最后得到32位浮点数的二进制存储格式为

$$\mathbf{0100\ 0001\ 1010\ 0100\ 1100\ 0000\ 0000\ 0000} = (41A4C000)_{16}$$

例：将十进制数-0.75表示成单精度的IEEE 754标准代码

解： $-0.75 = -3/4 = -0.11_2 = -1.1 \times 2^{-1}$

$$=(-1)^1 \times (1 + 0.1000\ 0000\ 0000\ 0000\ 0000\ 000) \times 2^{-1}$$

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

$$s=1, E=126_{10} = 01111110_2, F = 1000 \dots 000。$$

1 011, 1111, 0 100, 0000, 0000, 0000, 0000, 0000
B F 4 0 0 0 0 0 H

例：求如下IEEE 754 单精度浮点数的十进制数值：

1 10000001 010000000000000000000000

解：S=1, $E=129$, $F = 1/4 = 0.25$,

$$(-1)^1 \times (1+0.25) \times 2^{129-127}$$

$$= -1 \times 1.25 \times 2^2$$

$$= -1.25 \times 4$$

$$= -5.0$$

例：对数据 123_{10} 作规格化浮点数的编码，假定1位符号位，基数为2，阶码5位，采用移码，尾数10位，采用补码。

解： $123_{10}=1111011_2=0.1111011000_2\times 2^7$

$$[7]_{\text{移}}=10000+00111=10111$$

$$[0.1111011000]_{\text{补}}=0.1111011000$$

$$[123]_{\text{浮}}=0\ 10111\ 1111011000$$

将十进制数-0.75表示成单精度的IEEE

754标准代码

正常使用主观题需2.0以上版本雨课堂

作答

IEEE754浮点数的范围

格式	最小值	最大值
单精度	$E=1, M=0,$ $1.0 \times 2^{1-127} = 2^{-126}$	$E=254, f=.1111 \dots,$ $1.111 \dots 1 \times 2^{254-127}$ $= 2^{127} \times (2-2^{-23})$
双精度	$E=1, M=0,$ $1.0 \times 2^{1-1023} = 2^{-1022}$	$E=2046, f=.1111 \dots,$ $1.111 \dots 1 \times 2^{2046-1023}$ $= 2^{1023} \times (2-2^{-52})$

浮点数标准 (IEEE754)

IEEE 单精度浮点数编码格式

符号位	阶码	尾数	表示
0/1	255	非 0	NaN
0/1	255	非 0	NaN
0	255	0	+INF
1	255	0	-INF
0/1	1-254	f	$(-1)^s \times 1.f \times 2^{e-127}$
0/1	0	$f(\text{非 } 0)$	$(-1)^s \times 0.f \times 2^{e-126}$
0/1	0	0	+0/-0

浮点运算方法和浮点运算器

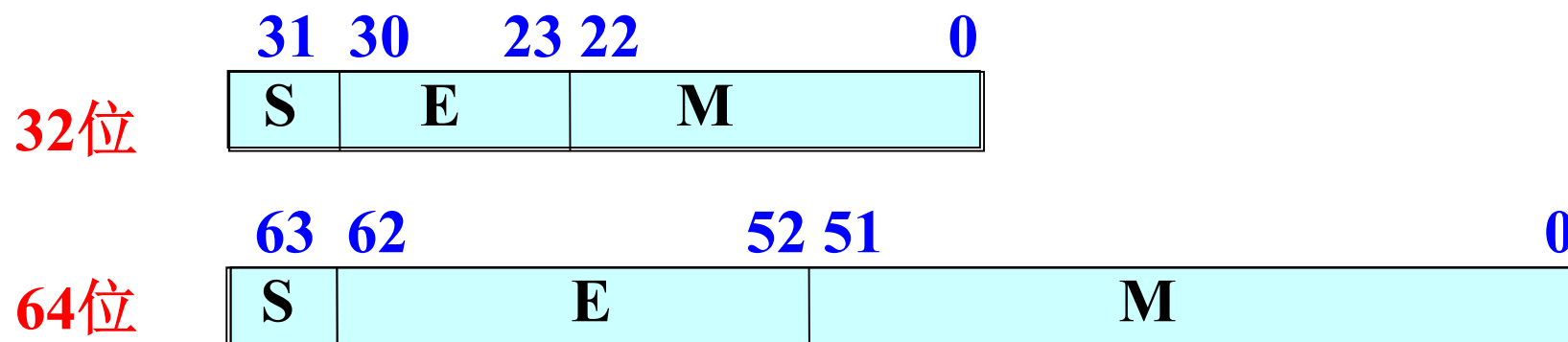
浮点加、减法运算

浮点乘、除法运算

浮点数的标准格式

为便于软件移植，使用 IEEE标准

按照 IEEE754 的标准，32位浮点数和64位浮点数的标准格式为：



IEEE 标准：尾数用原码； 阶码用“移码”； 基为2。

浮点加、减法运算

设有两个浮点数 x 和 y ，它们分别为：

$$x = 2^{E_x} \cdot M_x$$

$$y = 2^{E_y} \cdot M_y$$

其中 E_x 和 E_y 分别为数 x 和 y 的阶码，

M_x 和 M_y 为数 x 和 y 的尾数。

两浮点数进行加法和减法的运算规则是：

$$x \pm y = (M_x 2^{E_x - E_y} \pm M_y) 2^{E_y} \quad E_x \leq E_y$$

完成浮点加减运算的操作过程大体分为六步：

- (1) 0 操作数的检查；
- (2) 比较阶码大小并完成对阶；
- (3) 尾数进行加或减运算；
- (4) 结果规格化；
- (5) 舍入处理；
- (6) 溢出处理。

(1) 0 操作数检查

(2) 对阶

使二数阶码相同（即小数点位置对齐），这个过程叫作**对阶**。

- 先求两数阶码 E_x 和 E_y 之差，即 $\Delta E = E_x - E_y$

若 $\Delta E = 0$ ，表示 $E_x = E_y$

若 $\Delta E > 0$ ， $E_x > E_y$

若 $\Delta E < 0$ ， $E_x < E_y$

通过尾数的移动来改变 E_x 或 E_y ，使其相等。

- 对阶原则

阶码小的数向阶码大的数对齐；

小阶的尾数右移，每右移一位，其阶码加1（右规）。

例： $x=2^{01} \times 0.1101$, $y=2^{11} \times (-0.1010)$, 求 $x+y=?$

解： 为便于直观了解，两数均以**补码**表示，阶码、尾数均采用双符号位。

$$[x]_{\text{补}} = 00\ 01, \ 00.1101 \quad [y]_{\text{补}} = 00\ 11, \ 11.0110$$

$$[\Delta E]_{\text{补}} = [E_x]_{\text{补}} - [E_y]_{\text{补}} = 00\ 01 + 11\ 01 = 11\ 10$$

$\Delta E = -2$ ，表示 E_x 比 E_y 小 2，因此将 x 的尾数右移两位。

右移一位，得 $[x]_{\text{补}} = 00\ 10, \ 00.0110$

再右移一位，得 $[x]_{\text{补}} = 00\ 11, \ 00.0011$

至此， $\Delta E = 0$ ，对阶完毕。

(3) 尾数求和运算

尾数求和方法与定点加减法运算完全一样。

对阶完毕可得： $[x]_{\text{补}} = 00\ 11, 00.0011$

$[y]_{\text{补}} = 00\ 11, 11.0110$

对尾数求和： 00.0011

$+ 11.0110$

11.1001

即得： $[x+y]_{\text{补}} = 00\ 11, 11.1001$

(4) 结果规格化

求和之后得到的数可能不是规格化了的数，为了增加有效数字的位数，提高运算精度，必须将求和的结果规格化。

①规格化的定义： $\frac{1}{2} \leq |S| < 1$ (二进制)

采用原码：

正数： $S=0.1 \times \times \times \dots \times$

负数： $S=1.1 \times \times \times \dots \times$

采用双符号位的补码：

对正数： $S=00.1 \times \times \times \dots \times$

对负数： $S=11.0 \times \times \times \dots \times$

②向左规格化

若不是规格化的数, 需要尾数向左移位, 以实现规格化的过程, 我们称其为向左规格化。

前例中, 00 11, 11. 1001不是规格化数, 因而需要左规, 即左移一位, 阶码减1, 得:

$$[x+y]_{\text{补}} = 00 \ 10, \ 11. 0010$$

③向右规格化

浮点加减运算时, 尾数求和的结果也可能得到:

$$01. \times \times \times \dots \times \text{ 或 } 10. \times \times \times \dots \times,$$

即两符号位不等, 即结果的绝对值大于1。向左破坏了规格化。

此时, 将尾数运算的结果右移一位, 阶码加1, 称为向右规格化。

例：两浮点数 $x=0.1101 \times 2^{10}$, $y=(0.1011) \times 2^{01}$, 求 $x+y$ 。

解： $[x]_{\text{补}}=00\ 10, 00.1101$ $[y]_{\text{补}}=00\ 01, 00.1011$

对阶： $[\Delta E]_{\text{补}} = [E_x]_{\text{补}} - [E_y]_{\text{补}} = 00\ 10 + 11\ 11 = 00\ 01$

y 向 x 对齐，将 y 的尾数右移一位，阶码加1。

$[y]_{\text{补}}=00\ 10, 00.0101$

求和： 00.1101

$+ 00.0101$
 \hline
 01.0010

$[x+y]_{\text{补}}=00\ 10, 01.0010$

右归：运算结果两符号位不同，其绝对值大于1，右归。

$[x+y]_{\text{补}}=00\ 11, 00.1001$

(5) 舍入处理

在对阶或向右规格化时，尾数要向右移位，这样，被右移的尾数的低位部分会被丢掉，从而造成一定误差，因此要进行舍入处理。

- 简单的舍入方法有两种：

- ① “0舍1入”法

- 即如果右移时被丢掉数位的最高位为0则舍去，反之则将尾数的末位加“1”。

- ② “恒置1”法

- 即只要数位被移掉，就在尾数的末位恒置“1”。从概率上来说，丢掉的0和1各为1/2。

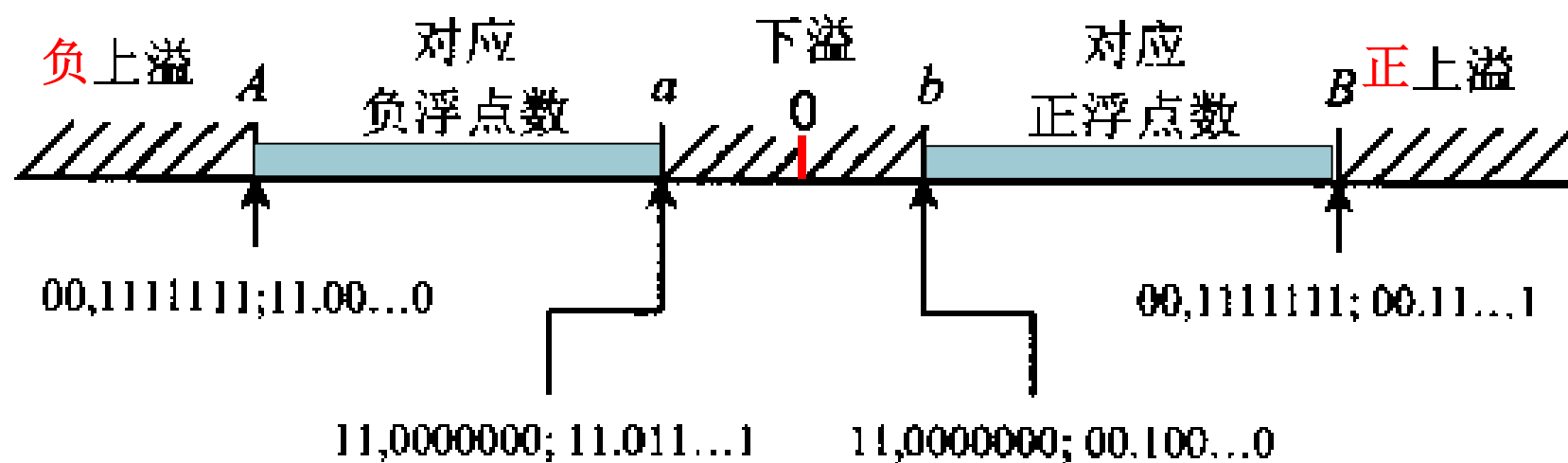
- IEEE754标准中，舍入处理提供了四种可选方法：

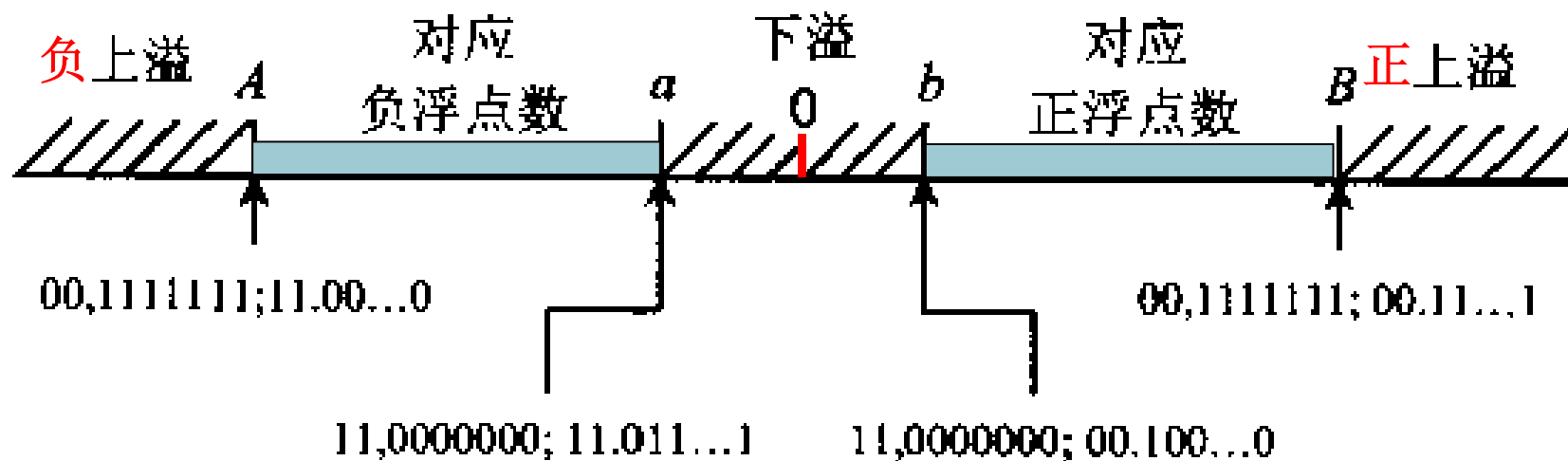
- 就近舍入、向+无穷，向-无穷或者零舍入，具体内容参见附加内容

(6) 溢出处理

与定点加减法一样，浮点加减运算最后一步也需判溢出。在浮点规格化中已指出，当尾数之和(差)出现 $01.\times\times\dots\times$ 或 $10.\times\times\dots\times$ 时，并不表示溢出，只有将此数右规后，再根据阶码来判断浮点运算结果是否溢出。

若机器数为补码，尾数为规格化形式，并假设阶符取2位，阶码取7位、数符取2位，尾数取n位，则它们能表示的补码在数轴上的表示范围如图所示。





图中A，B，a，b分别对应最小负数、最大正数、最大负数和最小正数。它们所对应的真值分别是：

- A最小负数 $2^{+127} \times (-1)$
- B最大正数 $2^{+127} \times (1-2^{-n})$
- a最大负数 $2^{-128} \times (-2^{-1}-2^{-n})$
- b最小正数 $2^{-128} \times 2^{-1}$

图中a,b之间的阴影部分，对应阶码小于128的情况，叫做浮点数的下溢。下溢时，浮点数值趋于零，故机器不做溢出处理，仅把它作为机器零。

图中的A、B两侧阴影部分，对应阶码大于127的情况，叫做浮点数的上溢。此刻，浮点数真正溢出，机器需停止运算，作溢出中断处理。一般说浮点溢出，均是指上溢。

可见，浮点机的溢出与否可由阶码的符号决定：

阶码 $[j]_{\text{补}}=01, \times \times \times \times \times$ 为上溢，机器停止运算，做中断处理；

阶码 $[j]_{\text{补}}=10, \times \times \times \times \times$ 为下溢，按机器零处理。

例：若某次加法操作的结果为 $[X+Y]_{\text{补}} = \mathbf{00.111}, 10.1011100111$

则应对其进行向右规格化操作：

尾数为： 11.0101110011 ， 阶码加1： 01.000

阶码超出了它所能表示的最大正数（+7），表明本次浮点运算产生了溢出。

例：若某次加法操作的结果为 $[X+Y]_{\text{补}} = \mathbf{11.010}, 00.0000110111$

则应对其进行向左规格化操作：

尾数为： 00.1101110000 ， 阶码减4：
$$\begin{array}{r} 11.010 \\ + 11.100 \\ \hline 10.110 \end{array} [-4]_{\text{补}}$$

阶码超出了它所能表示的最小负数（-8），表明本次浮点运算产生了溢出。

溢出处理小结

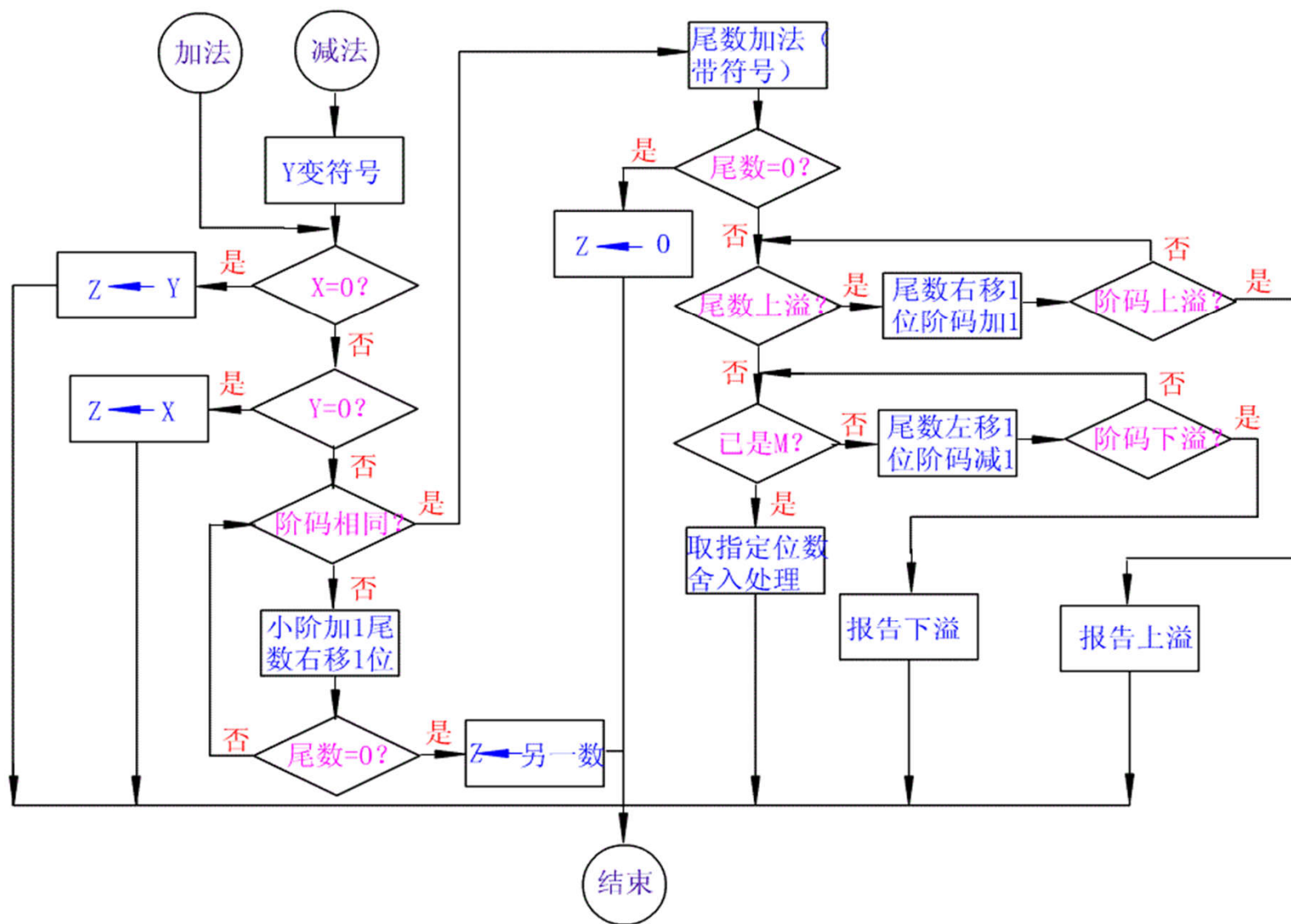
- 浮点数的溢出是以其阶码溢出表现出来的

在加、减运算过程中要检查是否产生了溢出：若阶码正常，加减运算正常结束；若阶码溢出，则要进行相应的处理。

阶码上溢——超过了阶码可能表示的最大值的正指数值,一般将其认为是 $+\infty$ 和 $-\infty$ 。

阶码下溢——超过了阶码可能表示的最小值的负指数值,一般将其认为是0。

- 对尾数的溢出也需要处理(上溢—右归, 下溢—舍入)。



例 设 $x = 2^{010} \times 0.11011011$, $y = 2^{100} \times (-0.10101100)$, 求 $x + y$ 。

阶码采用双符号位, 尾数采用单符号位补码形式

解: 浮点表示分别为

$$\begin{aligned}[x]_{\text{浮}} &= 00\ 010, & 0.11011011 \\[y]_{\text{浮}} &= 00\ 100, & 1.01010100\end{aligned}$$

(1) 求阶差并对阶

$$\Delta E = E_x - E_y = [E_x]_{\text{补}} + [-E_y]_{\text{补}} = 00\ 010 + 11\ 100 = 11\ 110$$

即 ΔE 为 -2, x 的阶码小, 应使 M_x 右移两位, E_x 加 2,

$$[x]_{\text{浮}} = 00\ 100, 0.00110110(11)$$

其中(11)表示 M_x 右移 2 位后移出的最低两位数。

(2) 尾数求和

$$\begin{array}{r} 0.00110110(11) \\ + 1.01010100 \\ \hline 1.10001010(11) \end{array}$$

(3) 规格化处理

尾数运算结果的符号位与最高数值位为同值，应执行左规处理，结果为1.00010101(10)，阶码为00 011。

(4) 舍入处理

采用0舍1入法处理，则有：

$$\begin{array}{r} 1.00010101 \\ + \quad \quad \quad 1 \\ \hline 1.00010110 \end{array}$$

(5) 判断溢出

阶码符号位为00，不溢出，故得最终结果为

$$x + y = 2^{011} \times (-0.11101010)$$

例 两浮点数 $x = 2^{01} \times 0.1101$, $y = 2^{11} \times (-0.1010)$ 。假设尾数在计算机中以补码表示, 可存储4位尾数, 2位保护位, 阶码以原码表示, 求 $x+y$ 。

解: 将 x, y 转换成浮点数据格式

$$[x]_{\text{浮}} = 00\ 01, 00.1101$$

$$[y]_{\text{浮}} = 00\ 11, 11.0110$$

步骤1: 对阶, 阶差为 $11-01=10$, 即2, 因此将 x 的尾数右移两位, 得

$$[x]_{\text{浮}} = 00\ 11, 00.001101$$

步骤2: 对尾数求和, 得:

$$[x+y]_{\text{浮}} = 00\ 11, 11.100101$$

步骤3: 由于符号位和第一位数相等, 不是规格化数, 向左规格化, 得

$$[x+y]_{\text{浮}} = 00\ 10, 11.001010$$

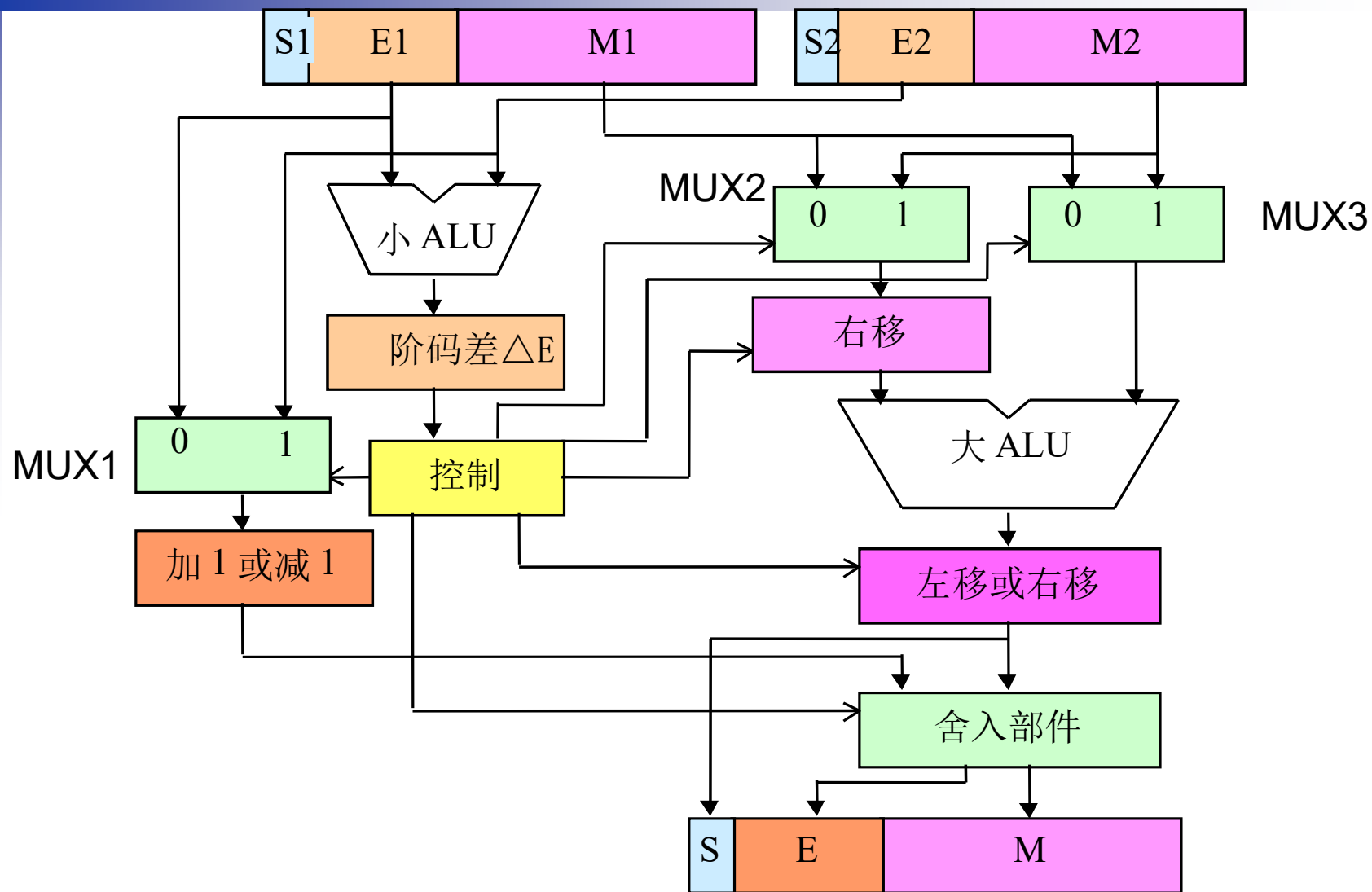
步骤4: 截去。

$$[x+y]_{\text{浮}} = 00\ 10, 11.0010$$

步骤5: 数据无溢出, 因此结果为

$$x+y = 2^{10} \times (-0.1110)$$

浮点运算电路



浮点加法器原理框图

浮点乘、除法运算

1.浮点乘法、除法运算规则

设有两个浮点数 x 和 y :

$$x = 2^{E_x} \cdot M_x$$
$$y = 2^{E_y} \cdot M_y$$

浮点乘法运算的规则是: $x \times y = 2^{(E_x + E_y)} \cdot (M_x \times M_y)$

即: 乘积的尾数是相乘两数的尾数之积;
乘积的阶码是相乘两数的阶码之和。

浮点除法运算的规则是: $x \div y = 2^{(E_x - E_y)} \cdot (M_x \div M_y)$

即: 商的尾数是相除两数的尾数之商;
商的阶码是相除两数的阶码之差。

2. 浮点乘、除法运算步骤

浮点数的乘除运算大体分为四步：

- (1) 0 操作数检查；
- (2) 阶码加/减操作；
- (3) 尾数乘/除操作；
- (4) 结果规格化及舍入处理。

(2) 浮点数的阶码运算

- 对阶码的运算有+1、-1、两阶码求和、两阶码求差四种，运算时还必须检查结果是否溢出。
- 在计算机中，阶码通常用补码或移码形式表示。

①移码的运算规则和判定溢出的方法

移码的定义为 $[x]_{\text{移}} = 2^n + x \quad -2^n \leq x < 2^n$

按此定义，则有：

$$\begin{aligned} [x]_{\text{移}} + [y]_{\text{移}} &= 2^n + x + 2^n + y \\ &= 2^n + (2^n + (x + y)) \\ &= 2^n + [x + y]_{\text{移}} \end{aligned}$$

$$[x + y]_{\text{移}} = -2^n + [x]_{\text{移}} + [y]_{\text{移}}$$

②混合使用移码和补码

考虑到移码和补码的关系：

对同一个数值，其数值位完全相同，而符号位正好完全相反。

$[y]_{\text{补}}$ 的定义为 $[y]_{\text{补}} = 2^{n+1} + y$

则求阶码和用如下方式完成：

$$[x]_{\text{移}} + [y]_{\text{补}} = 2^n + x + 2^{n+1} + y$$

$$= 2^{n+1} + (2^n + (x + y))$$

即：

$$[x + y]_{\text{移}} = [x]_{\text{移}} + [y]_{\text{补}} \quad (\text{mod } 2^{n+1})$$

同理：

$$[x - y]_{\text{移}} = [x]_{\text{移}} + [-y]_{\text{补}} \quad (\text{mod } 2^{n+1})$$

③ 阶码运算结果溢出处理

使用双符号位的阶码加法器，并规定移码的第二个符号位，即最高符号位恒用 0 参加加减运算，则溢出条件是结果的最高符号位为1：

- 当低位符号位为 0 时，(10) 表明结果上溢，
- 当低位符号位为 1 时，(11) 表明结果下溢。
- 当最高符号位为 0 时，表明没有溢出：
 - 低位符号位为 1，(01) 表明结果为正；
 - 为 0，(00) 表明结果为负。

例： $x = +011, y = +110$, 求 $[x+y]_{\text{移}}$ 和 $[x-y]_{\text{移}}$, 并判断是否溢出。

解：阶码取3位（不含符号位），其对应的真值范围是 $-8 \sim +7$

$$[x]_{\text{移}} = 01\ 011, \quad [y]_{\text{补}} = 00\ 110, \quad [-y]_{\text{补}} = 11\ 010$$

$$[x+y]_{\text{移}} = [x]_{\text{移}} + [y]_{\text{补}} =$$

$$\begin{array}{r} 01\ 011 \\ + 00\ 110 \\ \hline 10\ 001 \end{array}$$

结果上溢。

$$[x-y]_{\text{移}} = [x]_{\text{移}} + [-y]_{\text{补}} =$$

$$\begin{array}{r} 01\ 011 \\ + 11\ 010 \\ \hline 00\ 101 \end{array}$$

结果正确，为-3。

(3) 尾数处理

浮点加减法对结果的规格化及舍入处理也适用于浮点乘法。

第一种方法是：

无条件地丢掉正常尾数最低位之后的全部数值。

这种方法被称为截断处理，好处是处理简单，缺点是影响结果的精度。

第二种办法是：

运算过程中保留右移中移出的若干高位的值，最后再按某种规则用这些位上的值修正尾数。

这种处理方法被称为舍入处理。

舍入处理

- 当尾数用原码表示时:

最简便的方法是，只要尾数的最低位为1，或移出的几位中有为1的数值位，就使最低位的值为1。

另一种是0舍1入法，即当丢失的最高位的值为1时，把这个1加到最低数值位上进行修正，否则舍去丢失的各位的值。这样处理时，舍入效果对正数负数相同，入将使数的绝对值变大，舍则使数的绝对值变小。

- 当尾数是用补码表示时:

采用0舍1入法时, 若丢失的位不全为0时:

对正数来说, 舍入的结果与原码分析相同;

对负数来说, 舍入的结果与原码分析相反, 即“舍”使绝对值变大, “入”使绝对值变小; 为使原、补码舍入处理后的结果相同, 对负数可采用如下规则进行舍入处理:

当丢失的各位均为0时, 不必舍入; 当丢失的最高位为0, 以下各位不全为0时, 或者丢失的最高位为1, 以下各位均为0时, 则舍去丢失位上的值; 当丢失的最高位为1, 以下各位不全为0时, 则执行在尾数最低位入1的修正操作。

例 设 $[x_1]_{\text{补}} = 11.01100000$, $[x_2]_{\text{补}} = 11.01100001$,

$[x_3]_{\text{补}} = 11.01101000$, $[x_4]_{\text{补}} = 11.01111001$, 求执行只保留
小数点后4位有效数字的舍入操作值。

解: 执行舍入操作后,其结果值分别为

$$[x_1]_{\text{补}} = 11.0110 \quad (\text{不舍不入})$$

$$[x_2]_{\text{补}} = 11.0110 \quad (\text{舍})$$

$$[x_3]_{\text{补}} = 11.0110 \quad (\text{舍})$$

$$[x_4]_{\text{补}} = 11.1000 \quad (\text{入})$$

例 设有浮点数 $x = 2^{-5} \times 0.0110011$, $y = 2^3 \times (-0.1110010)$, 阶码用4位移码表示, 尾数 (含符号位)用8位补码表示。求 $[x \times y]_{\text{浮}}$ 。要求用补码完成尾数乘法运算, 运算结果尾数保留高8位(含符号位), 并用尾数低位字长值处理舍入操作。

解: 移码采用双符号位, 尾数补码采用单符号位, 则有

$$[M_x]_{\text{补}} = 0.0110011, \quad [M_y]_{\text{补}} = 1.0001110,$$

$$[E_x]_{\text{移}} = 00 \ 011, \quad [E_y]_{\text{移}} = 01 \ 011, \quad [E_y]_{\text{补}} = 00 \ 011,$$

$$[x]_{\text{浮}} = 00 \ 011, 0.0110011, \quad [y]_{\text{浮}} = 01 \ 011, 1.0001110$$

(1) 求阶码和

$$[E_x + E_y]_{\text{移}} = [E_x]_{\text{移}} + [E_y]_{\text{补}} = 00\ 011 + 00\ 011 = 00\ 110,$$

值为移码形式-2。

(2) 尾数乘法运算

可采用补码阵列乘法器实现，即有

$$\begin{aligned} [M_x]_{\text{补}} \times [M_y]_{\text{补}} &= [0.0110011]_{\text{补}} \times [1.0001110]_{\text{补}} \\ &= [1.1010010, 1001010]_{\text{补}} \end{aligned}$$

(3) 规格化处理

乘积的尾数符号位与最高数值位符号相同，不是规格化的数，需要左规，阶码变为00 101(-3)，尾数变为 1.0100101,0010100。

(4) 舍入处理

尾数为负数，取尾数高位字长，按舍入规则，舍去低位字长，故尾数为1.0100101。

最终相乘结果为 $[x \times y]_{\text{浮}} = 00\ 101, 1.0100101$

其真值为 $x \times y = 2^{-3} \times (-0.1011011)$

浮点数表示与算法小结:

- 1) IEEE 754 浮点数的表示
- 2) 尾数的规格化表示
- 3) 浮点数计算流程
- 4) 浮点数计算的硬件实现

计算机的算术运算指令为什么只需要提供加、减、乘、除四类指令即可满足各种科学计算的需要？

是否需要提供开方、三角函数运算的指令？为什么？

FP Instructions in MIPS

- FP hardware is coprocessor 1
 - Adjunct processor that extends the ISA
- Separate FP registers
 - 32 single-precision: `$f0, $f1, ... $f31`
 - Paired for double-precision: `$f0/$f1, $f2/$f3, ...`
 - Release 2 of MIPS ISA supports 32×64 -bit FP reg's
- FP instructions operate only on FP registers
 - Programs generally don't do integer ops on FP data, or vice versa
 - More registers with minimal code-size impact
- FP load and store instructions
 - `lwc1, ldc1, swc1, sdc1`
 - e.g., `ldc1 $f8, 32($sp)`

FP Instructions in MIPS

- Single-precision arithmetic
 - `add.s, sub.s, mul.s, div.s`
 - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
 - `add.d, sub.d, mul.d, div.d`
 - e.g., `mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
 - `c.xx.s, c.xx.d` (`xx` is `eq, lt, le, ...`)
 - Sets or clears FP condition-code bit
 - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false
 - `bc1t, bc1f`
 - e.g., `bc1t TargetLabel`

FP Example: °F to °C

- C code:

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

- fahr in **\$f12**, result in **\$f0**, literals in global memory space

- Compiled MIPS code:

```
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
```

FP Example: Array Multiplication

- $X = X + Y \times Z$

- All 32×32 matrices, 64-bit double-precision elements

- C code:

```
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];  
}
```

- Addresses of x, y, z in $\$a0, \$a1, \$a2$, and
i, j, k in $\$s0, \$s1, \$s2$

FP Example: Array Multiplication

■ MIPS code:

	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
	sll	\$t2, \$s0, 5	# \$t2 = i * 32 (size of row of x)
	addu	\$t2, \$t2, \$s1	# \$t2 = i * size(row) + j
	sll	\$t2, \$t2, 3	# \$t2 = byte offset of [i][j]
	addu	\$t2, \$a0, \$t2	# \$t2 = byte address of x[i][j]
	l.d	\$f4, 0(\$t2)	# \$f4 = 8 bytes of x[i][j]
L3:	sll	\$t0, \$s2, 5	# \$t0 = k * 32 (size of row of z)
	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 z[k][j]
	l.d	\$f16, 0(\$t0)	# \$f16 = 8 bytes of z[k][j]

...

FP Example: Array Multiplication

...		
sll	\$t0, \$s0, 5	# \$t0 = i*32 (size of row of y)
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 y[i][k]
l.d	\$f18, 0(\$t0)	# \$f18 = 8 bytes of y[i][k]
mul.d	\$f16, \$f18, \$f16	# \$f16 = y[i][k] * z[k][j]
add.d	\$f4, \$f4, \$f16	# f4=x[i][j] + y[i][k]*z[k][j]
addiu	\$s2, \$s2, 1	# \$k = k + 1
bne	\$s2, \$t1, L3	# if (k != 32) go to L3
s.d	\$f4, 0(\$t2)	# x[i][j] = \$f4
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

Accurate Arithmetic

- Infinite variety of real numbers between, say, 0 and 1
 - Only 2^{53} can be represented by double precision FP
- IEEE Std 754 specifies additional rounding control
 - Extra bits of precision (**guard, round, sticky**)
 - **guard** and **round** bits are 2 extra bits kept on the right during intermediate additions
 - **sticky** bit used in rounding in addition to **guard** and **round** bits; is set whenever a 1 bit shifts right of the round bit

F = 1 . xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx **G R S**

- Not all FP units implement all options
 - Most programming languages and FP libraries just use defaults
- Trade-off between hardware complexity, performance, and market requirements

Accurate Arithmetic



附加位和舍入

加多少附加位才合适？没有绝对的答案。

IEEE754规定：中间结果须在右边加**2个附加位**（**guard & round**）

Guard bit(保护位)：在尾数部分右边的位

Rounding bit(舍入位)：在保护位右边的位

附加位的作用：用以保护对阶时右移的位或运算的中间结果。

附加位的处理：①左规时被移到尾数中；② 作为舍入的依据。

举例：十进制数，最终有效位数为**3**，假定采用两位附加位。

问题：若没有舍入位，采用就近舍入到偶数，则结果是什么？

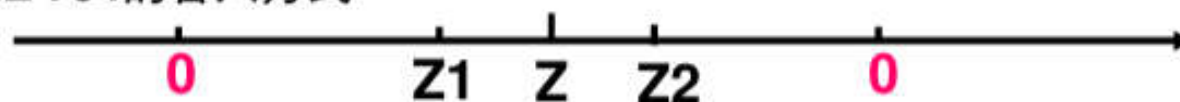
结果为**2.36**！精度没有**2.37**高！

$$\begin{array}{r} 2.34\textcolor{blue}{00} * 10^2 \\ 0.02\textcolor{blue}{53} * 10^2 \\ \hline 2.36\textcolor{blue}{53} * 10^2 \end{array}$$

文档在线

IEEE 754的舍入方式

IEEE 754的舍入方式



(Z1和Z2分别是结果Z的最近可表示的左、右数)

(1)就近舍入: 舍入为最近可表示的数

非中间值: 0舍1入;

中间值: 强迫结果为偶数-慢

如: 附加位为

01: 舍

11: 入

10: (强迫结果为偶数)

例: $1.110111 \rightarrow 1.1110$; $1.110101 \rightarrow 1.1101$;
 $1.110110 \rightarrow 1.1110$; $1.111110 \rightarrow 10.0000$;

(2)朝 $+\infty$ 方向舍入: 舍入为Z2(正向舍入)

(3)朝 $-\infty$ 方向舍入: 舍入为Z1(负向舍入)

(4)朝0方向舍入: 截去。正数: 取Z1; 负数: 取Z2

文档在线

IEEE 754的舍入方式的说明

IEEE 754通过在舍入位后再引入“粘位**sticky bit**”增强精度
加减运算对阶过程中，若阶码较小的数的尾数右移时，舍入位之后有非0数，则可设置**sticky bit**。

举例：

$1.110 \times 2^5 + 1.010 \times 2^1$ 分别采用二位、三位附加位时，结果各是多少？（就近舍入到偶数）

尾数精确结果为 1.110101 ，所以分别为：

1.110 ， 1.111 （误差较小）

Parallelism and Associativity

- Parallel programs may interleave operations in unexpected orders
 - Integer addition is associative
 - Assumptions of associativity for FP numbers may fail!

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

- Floating point numbers are approximations of real numbers – not associative!
- Need to validate parallel programs under varying degrees of parallelism

x86 FP Architecture

- Originally based on 8087 FP coprocessor
 - 8 × 80-bit extended-precision registers
 - Used as a push-down stack
 - Registers indexed from TOS: ST(0), ST(1), ...
- FP values are 32-bit or 64-bit in memory
 - Converted on load/store of memory operand
 - Integer operands can also be converted on load/store

x86 FP Instructions

Data transfer	Arithmetic	Compare	Transcendental
FILD mem/ST(i) FISTP mem/ST(i) FLDPI FLD1 FLDZ	FIADDP mem/ST(i) FISUBRP mem/ST(i) FIMULP mem/ST(i) FIDIVRP mem/ST(i) FSQRT FABS FRNDINT	FICOMP FIUCOMP FSTSW AX/mem	FPATAN F2XMI FCOS FPTAN FPREM FPSIN FYL2X

- No FP branch – FSTSW sends result of CMP to INT CPU
- Optional variations
 - **I**: integer operand
 - **P**: pop operand from stack
 - **R**: reverse operand order
 - But not all combinations allowed

Streaming SIMD Extension 2 (SSE2)

- Adds 4 × 128-bit registers
 - Extended to 8 registers in AMD64/EM64T
- Can be used for multiple FP operands
 - 2 × 64-bit double precision
 - 4 × 32-bit single precision
 - Instructions operate on them simultaneously
 - Single-Instruction Multiple-Data

Fallacy: Right Shift and Division

- Left shift by i places multiplies an integer by 2^i
- Right shift divides by 2^i ?
 - Only for unsigned integers!
- For signed integers
 - Logical right shift is clearly erroneous
 - e.g., $-5 / 4$
 - $11111011_2 \ggg 2 = 00111110_2 = +62$
 - Arithmetic right shift - replicate the sign bit
 - $11111011_2 \gg 2 = 11111110_2 = -2$
 - Result is -2 instead of -1; close, but no cigar

Who Cares About FP Accuracy?

- Important for scientific code
 - But for everyday consumer use?
 - “My bank balance is out by 0.0002¢!” ☹
- The Intel Pentium FDIV bug (~1994)
 - Bug in LUT used to guess multiple quotient bits per step; wrong values in some LUT locations
 - **Cost Intel \$300+ million**
 - The market expects accuracy
 - See Colwell, *The Pentium Chronicles*



Summary: MIPS Instruction Set

MIPS core instructions	Name	Format	MIPS arithmetic core	Name	Format
add	add	R	multiply	mult	R
add immediate	addi	I	multiply unsigned	multu	R
add unsigned	addu	R	divide	div	R
add immediate unsigned	addiu	I	divide unsigned	divu	R
subtract	sub	R	move from Hi	mfhi	R
subtract unsigned	subu	R	move from Lo	mflo	R
AND	AND	R	move from system control (EPC)	mfc0	R
AND immediate	ANDi	I	floating-point add single	add.s	R
OR	OR	R	floating-point add double	add.d	R
OR immediate	ORi	I	floating-point subtract single	sub.s	R
NOR	NOR	R	floating-point subtract double	sub.d	R
shift left logical	sll	R	floating-point multiply single	mul.s	R
shift right logical	srl	R	floating-point multiply double	mul.d	R
load upper immediate	lui	I	floating-point divide single	div.s	R
load word	lw	I	floating-point divide double	div.d	R
store word	sw	I	load word to floating-point single	lwc1	I
load halfword unsigned	lhu	I	store word to floating-point single	swc1	I
store halfword	sh	I	load word to floating-point double	ldc1	I
load byte unsigned	lbu	I	store word to floating-point double	sdcl	I
store byte	sb	I	branch on floating-point true	bclt	I
load linked (<i>atomic update</i>)	ll	I	branch on floating-point false	bclf	I
store cond. (<i>atomic update</i>)	sc	I	floating-point compare single	c.x.s	R
branch on equal	beq	I	(x = eq, neq, lt, le, gt, ge)		
branch on not equal	bne	I	floating-point compare double	c.x.d	R
jump	j	J	(x = eq, neq, lt, le, gt, ge)		
jump and link	jal	J			
jump register	jr	R			
set less than	slt	R			
set less than immediate	slti	I			
set less than unsigned	sltu	R			
set less than immediate unsigned	sltiu	I			

Summary: MIPS Instruction Set

Remaining MIPS-32	Name	Format	Pseudo MIPS	Name	Format
exclusive or ($rs \oplus rt$)	xor	R	absolute value	abs	rd,rs
exclusive or immediate	xori	I	negate (<i>signed or unsigned</i>)	negs	rd,rs
shift right arithmetic	sra	R	rotate left	rol	rd,rs,rt
shift left logical variable	sllv	R	rotate right	ror	rd,rs,rt
shift right logical variable	srlv	R	multiply and don't check oflw (<i>signed or uns.</i>)	mul _s	rd,rs,rt
shift right arithmetic variable	srav	R	multiply and check oflw (<i>signed or uns.</i>)	mul _{os}	rd,rs,rt
move to Hi	mthi	R	divide and check overflow	div	rd,rs,rt
move to Lo	mtlo	R	divide and don't check overflow	divu	rd,rs,rt
load halfword	lh	I	remainder (<i>signed or unsigned</i>)	rem _s	rd,rs,rt
load byte	lb	I	load immediate	li	rd,imm
load word left (<i>unaligned</i>)	lwl	I	load address	la	rd,addr
load word right (<i>unaligned</i>)	lwr	I	load double	ld	rd,addr
store word left (<i>unaligned</i>)	swl	I	store double	sd	rd,addr
store word right (<i>unaligned</i>)	swr	I	unaligned load word	ulw	rd,addr
load linked (<i>atomic update</i>)	ll	I	unaligned store word	usw	rd,addr
store cond. (<i>atomic update</i>)	sc	I	unaligned load halfword (<i>signed or uns.</i>)	ulh _s	rd,addr
move if zero	movz	R	unaligned store halfword	ush	rd,addr
move if not zero	movn	R	branch	b	Label
multiply and add (<i>S or uns.</i>)	madd _s	R	branch on equal zero	beqz	rs,L
multiply and subtract (<i>S or uns.</i>)	msub _s	I	branch on compare (<i>signed or unsigned</i>)	bx _s	rs,rt,L
branch on \geq zero and link	bgezal	I	($x = lt, le, gt, ge$)		
branch on $<$ zero and link	bltzal	I	set equal	seq	rd,rs,rt
jump and link register	jlr	R	set not equal	sne	rd,rs,rt
branch compare to zero	bxz	I	set on compare (<i>signed or unsigned</i>)	sx _s	rd,rs,rt
branch compare to zero likely	bxzl	I	($x = lt, le, gt, ge$)		
($x = lt, le, gt, ge$)			load to floating point (<u>s</u> or <u>d</u>)	l. _f	rd,addr
branch compare reg likely	bxl	I	store from floating point (<u>s</u> or <u>d</u>)	s. _f	rd,addr
trap if compare reg	tx	R			
trap if compare immediate	txi	I			
($x = eq, neq, lt, le, gt, ge$)					
return from exception	rfe	R			
system call	syscall	I			
break (<i>cause exception</i>)	break	I			
move from FP to integer	mfc1	R			
move to FP from integer	mtc1	R			
FP move (<u>s</u> or <u>d</u>)	mov. _f	R			
FP move if zero (<u>s</u> or <u>d</u>)	movz. _f	R			
FP move if not zero (<u>s</u> or <u>d</u>)	movn. _f	R			
FP square root (<u>s</u> or <u>d</u>)	sqr. _f	R			
FP absolute value (<u>s</u> or <u>d</u>)	abs. _f	R			
FP negate (<u>s</u> or <u>d</u>)	neg. _f	R			
FP convert (<u>w</u> , <u>s</u> , or <u>d</u>)	cvt. _{f,f}	R			
FP compare un (<u>s</u> or <u>d</u>)	c.xn. _f	R			

Frequency of Common MIPS Instructions

- Only included those with >3% (table 1) and >1% (table 2)

MIPS core	SPECint	SPECfp
addu	5.2%	3.5%
addiu	9.0%	7.2%
or	4.0%	1.2%
sll	4.4%	1.9%
lui	3.3%	0.5%
lw	18.6%	5.8%
sw	7.6%	2.0%
lbu	3.7%	0.1%
beq	8.6%	2.2%
bne	8.4%	1.4%
slt	9.9%	2.3%
slti	3.1%	0.3%
sltu	3.4%	0.8%

Arith core + MIPS-32	SPECint	SPECfp
add.d	0.0%	10.6%
sub.d	0.0%	4.9%
mul.d	0.0%	15.0%
add.s	0.0%	1.5%
sub.s	0.0%	1.8%
mul.s	0.0%	2.4%
l.d	0.0%	17.5%
s.d	0.0%	4.9%
l.s	0.0%	4.2%
s.s	0.0%	1.1%
lhu	1.3%	0.0%

Interpretation of Data

The BIG Picture

- Bits have no inherent meaning
 - Same bits can represent a variety of objects
 - Interpretation depends on the instructions applied
- Computer representations of numbers
 - Finite range and precision
 - Programmers, computer systems must **minimize gap** between computer arithmetic and real world arithmetic

Concluding Remarks

- ISAs support arithmetic
 - Signed and unsigned integers
 - Floating-point approximation for reals
- Bounded range and precision
 - Operations can overflow and underflow
- MIPS ISA
 - **MIPS core** and **arithmetic core** instructions: 54 most frequently used
 - 100% of SPECINT, 97% of SPECFP
 - Other instructions: less frequent

Assignment 3

- Homework assignment
3.1-3.3 , 3.29 ,3.30, 3.41-3.42
- To be submitted in the next week class
- Read Section 3.10 of Chapter 3
Historical Perspective and Further
Reading

Project

1. 项目目标

- (1) 深入掌握二进制数的表示方法以及不同进制数的转换;
- (2) 掌握二进制不同编码的表示方法,掌握IEEE 754 中单精度浮点数的表示和计算;
- (3) 能够应用MIPS汇编进行编程。

2. 具体要求

假设没有浮点表示和计算的硬件, 用软件方法采用仿真方式实现IEEE 754单精度浮点数的表示及运算功能, 具体要求如下:

- (1) 程序需要提供人机交互方式(字符界面)供用户选择相应的功能;
- (2) 可接受十进制实数形式的输入, 在内存中以IEEE 754单精度方式表示, 支持以二进制和十六进制的方式显示输出;
- (3) 可实现浮点数的加减(或者乘除)运算;
- (4) 使用MIPS汇编指令, 但是不能直接使用浮点指令, 只能利用整数运算指令来编写软件完成。
- (5) 设计报告中给出程序的需求分析和关键算法的流程图, 提交的代码注释比例>40%, 注释语义清晰。

3. 开发工具 采用Mars仿真器

4. 提交资料

按照学校实验报告的格式, 要求提交源程序、设计文档以及编译后可以执行程序电子版。

5. 提交时间

期末考试前一周。