计算机组织结构

5 浮点数运算

刘博涵

2022年10月13日

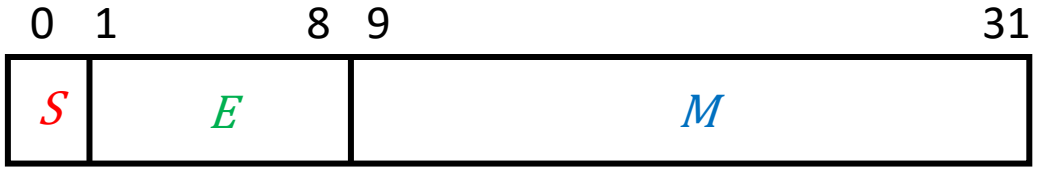
南京大学
NANJING UNIVERSITY

教材对应章节

第3章 运算方法和运算部件

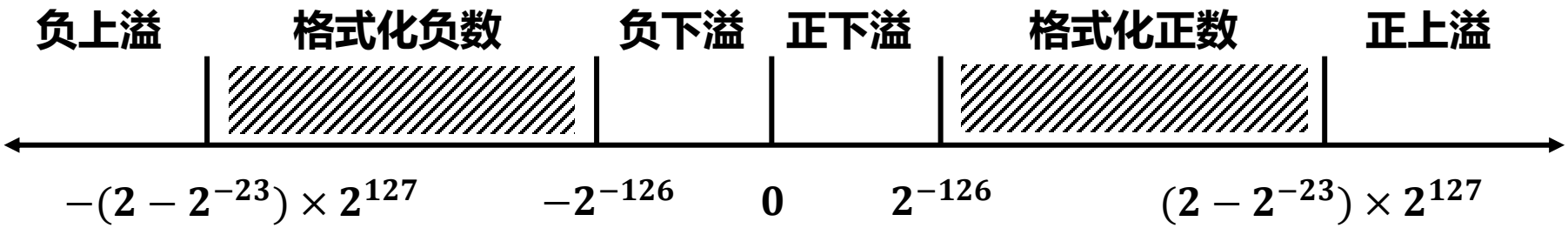
第10章 计算机算术运算

回顾：IEEE 754浮点数表示



1位 8位**移码** 23位**原码**表示**24位**尾数

$$X = (-1)^S \times 1.M \times 2^E$$



阶码的值	尾数的值	表示
0（全0）	0	+/- 0
0（全0）	非0	非规格化数
1~254	任意	规格化数
255（全1）	0	+/- ∞
255（全1）	非0	NaN

加法和减法

- 必须确保两个操作数具有相同的**指数值（阶）**

$$X + Y = (X_S \times B^{X_E - Y_E} + Y_S) \times B^{Y_E}$$
$$X - Y = (X_S \times B^{X_E - Y_E} - Y_S) \times B^{Y_E}$$
$$X_E \leq Y_E$$

- 计算步骤
  - 检查0：X是否等于0？Y是否等于0?
  - 对齐有效值：阶码向大值对齐 $X_S \times B^{X_E} = X_S \times B^{X_E - Y_E} \times B^{Y_E}$，$X_S$左移$(X_E - Y_E)$位
  - 加或减有效值： $X_S \pm Y_S$ （原码加减法）
  - 规格化结果：
    - 左规：0.$\underbrace{0\ldots0}_{k-1个0}$bb...bb则左移$k$位， $Y_E = Y_E - k$
    - 右规：$\underbrace{11}_{只可能2个1}$.bb...bb则右移1位， $Y_E = Y_E + 1$

乘法和除法

$$X \times Y = (X_S \times Y_S) \times B^{X_E + Y_E - bias}$$
$$X \div Y = (X_S \div Y_S) \times B^{X_E - Y_E + bias}$$

- 计算步骤
  - 检查0：X是否等于0？Y是否等于0?
  - 阶码相加减： $X_E \pm Y_E \pm bias$
  - 尾数相乘除： $X_S \times Y_S$ ， $X_S \div Y_S$
  - 规格化结果：
    - 左规
    - 右规

溢出

- **阶值上溢**
  - 正阶值超过可能的最大允许阶值　　**11111110 (127)**
  - 标记为 +∞ 或者 −∞
- **阶值下溢**
  - 负阶值小于可能的最小允许阶值　　**00000001 (-126)**
  - 报告为 0

- **有效值上溢**
  - 同符号的两个有效值相加可能导致最高有效位的进位
  - 通过重新对齐来修补
- **有效值下溢**
  - 在有效值对齐过程中，可能有数字被移出右端最低位而丢失
  - 需要某种形式的四舍五入

右规

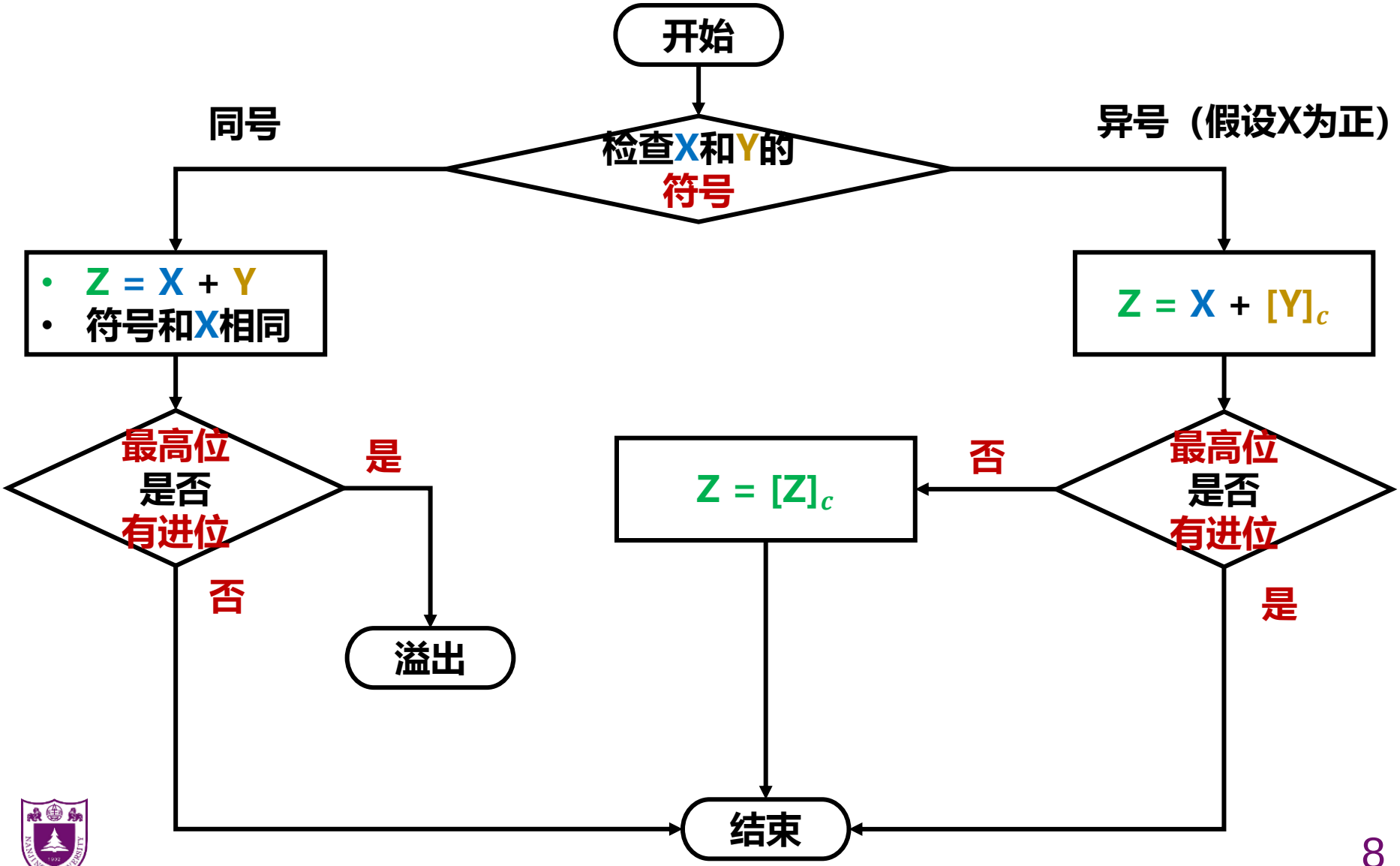
IEEE754标准规定的五种异常

- **无效运算（结果为NaN）**
  - 运算时有一个数是非有限数，如：加/减 $\pm\infty$ ，$\pm\infty/\pm\infty$， $0\times\pm\infty$，等
  - 结果无效，如：0/0，源操作数为NaN，一个数对0取余等
- **有限数除以0（结果为$\pm\infty$）**
- **数太大**（阶上溢，**结果为**$\pm\infty$）：如对于单精度，阶码 > 1111 1110（127）
- **数太小**（阶下溢，**结果用非规格化数表示**）：如对于单精度，阶码 < 0000 0001（-126）。注：IEEE754出现前阶下溢一般为0，换言之，IEEE754解决了这一问题。

  结果不精确（舍入时引起）：如1/3不能精确表示为一个浮点数

上述异常可以通过硬件识别和处理，也可以让软件处理
硬件处理时，称为硬件陷阱

原码的加法


开始
检查X和Y的符号
同号
异号（假设X为正）
Z = X + Y
符号和X相同
Z = X + [Y]c
最高位是否有进位
是
否
溢出
Z = [Z]c
否
最高位是否有进位
是
结束

尾数的原码加法示例

-0.8125-0.625 = -1.4375

0.8125+0.625 = 1.4375

$$\begin{array}{r} 0.1101 \\ +\ 0.1010 \\ \hline 1.0111 \end{array}$$

1.4375

尾数相加会溢出吗?

0.625-0.8125=-0.1875

$$\begin{array}{r} 0.1010 \\ +\ 0.0011 \\ \hline 0.1101 \end{array}$$

↓

0.0011

- 0.1875

无进位
求补码

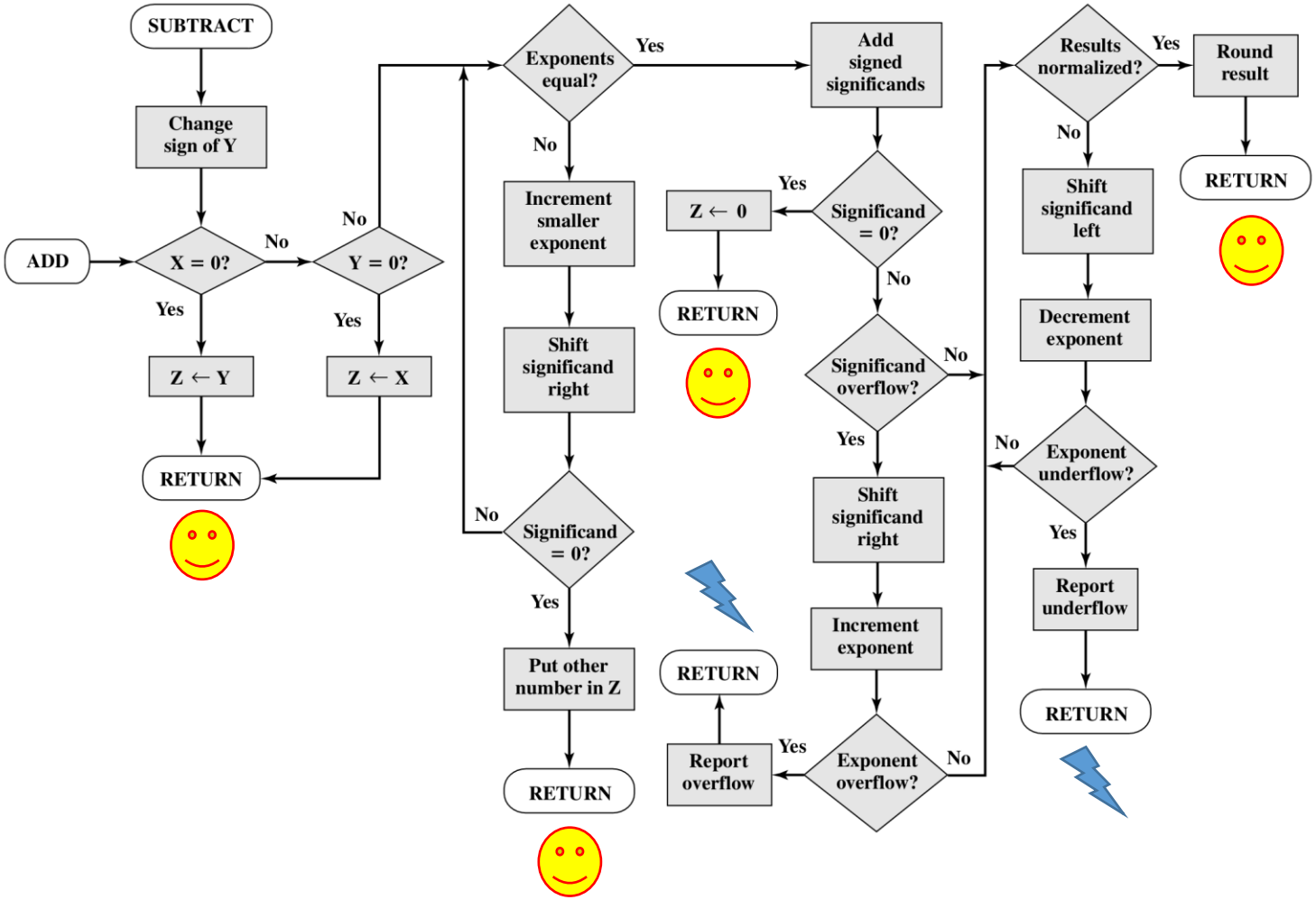
0.8125-0.625 = 0.1875

$$\begin{array}{r} 0.1101 \\ +\ 0.0110 \\ \hline 1.0011 \end{array}$$

+ 0.1875

有进位
去进位

加法和减法流程图


SUBTRACT
Change sign of Y
ADD
X = 0?
No
Yes
Z ← Y
RETURN
Y = 0?
No
Yes
Z ← X
Exponents equal?
Yes
No
Increment smaller exponent
Shift significand right
Significand = 0?
No
Yes
Put other number in Z
RETURN
Add signed significands
Significand = 0?
Yes
Z ← 0
RETURN
No
Significand overflow?
No
Yes
Shift significand right
Increment exponent
Exponent overflow?
Yes
Report overflow
RETURN
No
Results normalized?
Yes
Round result
RETURN
No
Shift significand left
Decrement exponent
Exponent underflow?
No
Yes
Report underflow
RETURN

加法和减法示例

0.5 – (-0.4375) = 0.9375

0.5　　0 01111110 000…00 (23)

-0.4375　1 01111101 110…00 (21)

01111110 – 01111101 = 01111110 + 10000011 = 00000001

```
   1 0000…00
 + 0 1110…00
 ------------
   1 1110…00
```

0 01111110 1110…00 (20)

加法和减法示例

0.5 + (-0.4375) = 0.0625

0.5　　0 01111110 000…00 (23)

-0.4375　1 01111101 110…00 (21)

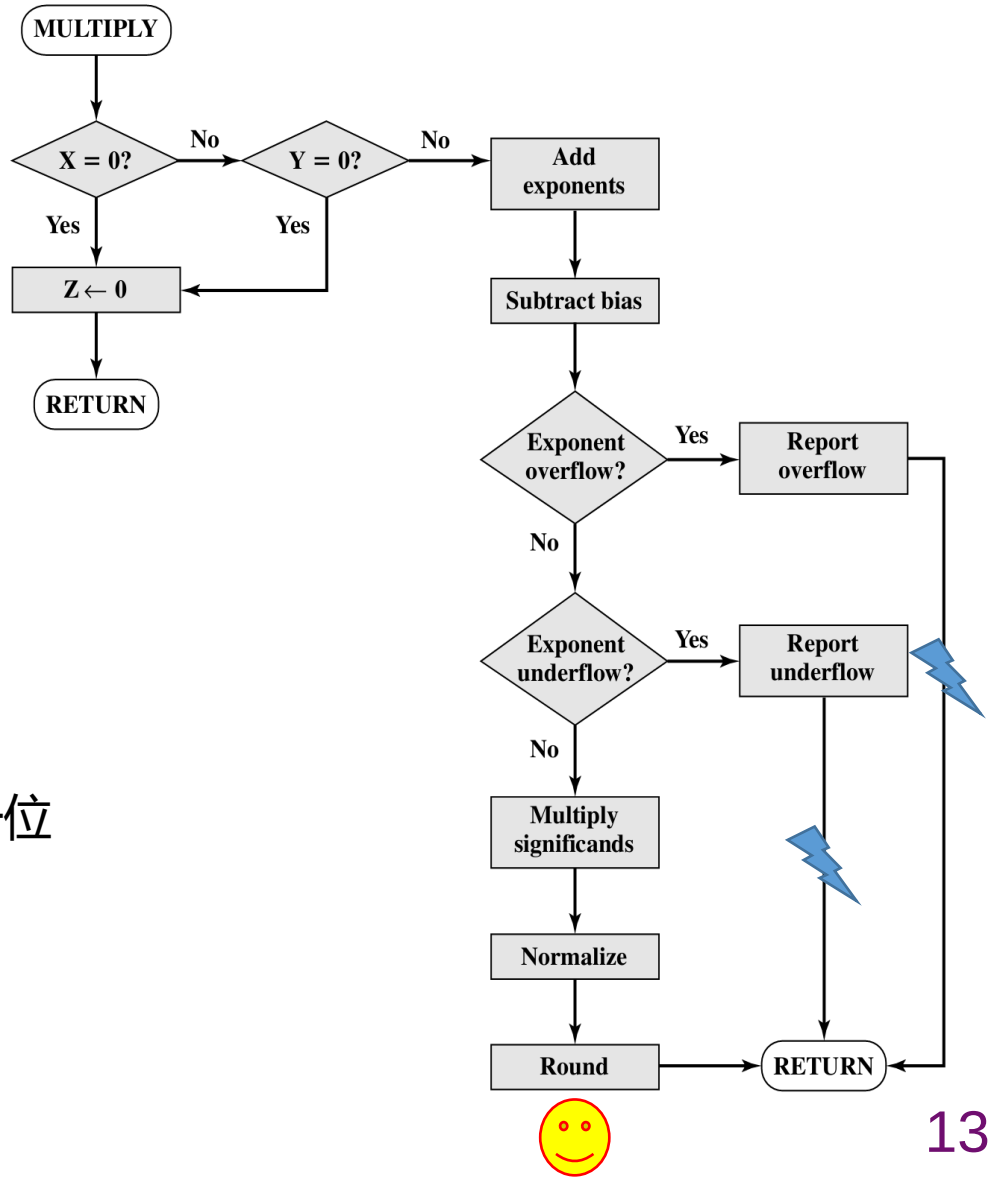
01111110 – 01111101 = 01111110 + 10000011 = 00000001

$$
\begin{array}{r}
1\ 0000\ldots00 \\
+\ 1\ 0010\ldots00 \\
\hline
1\ 0\ 0010\ldots00
\end{array}
$$

0 01111011 000…00 (23)

乘法

- 无论哪个操作数是0，乘积即为0
- 从阶值的和中减去一个偏移量
- 有效值相乘
- 结果的规格化和舍入处理
  - 规格化可能导致阶值下溢
  - IEEE 754标准中，只有右规
    - 高两位为01则无需处理
    - 高两位为10、11则右规一位

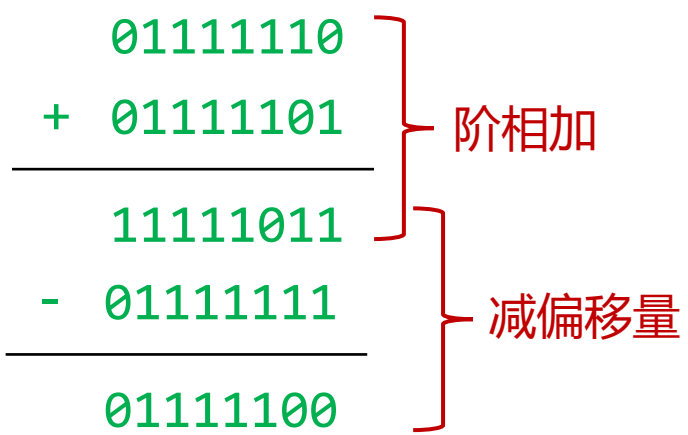


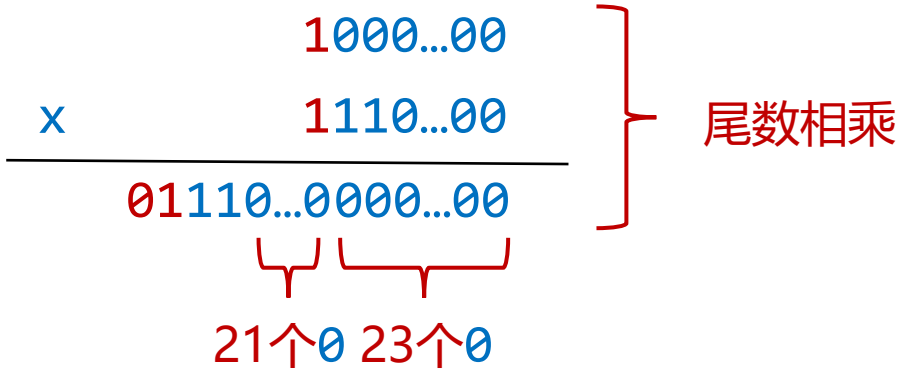
乘法示例

0.5×0.4375 = 0.21875

0.5　　0 01111110 000…00 (23)

0.4375　　0 01111101 110…00 (21)



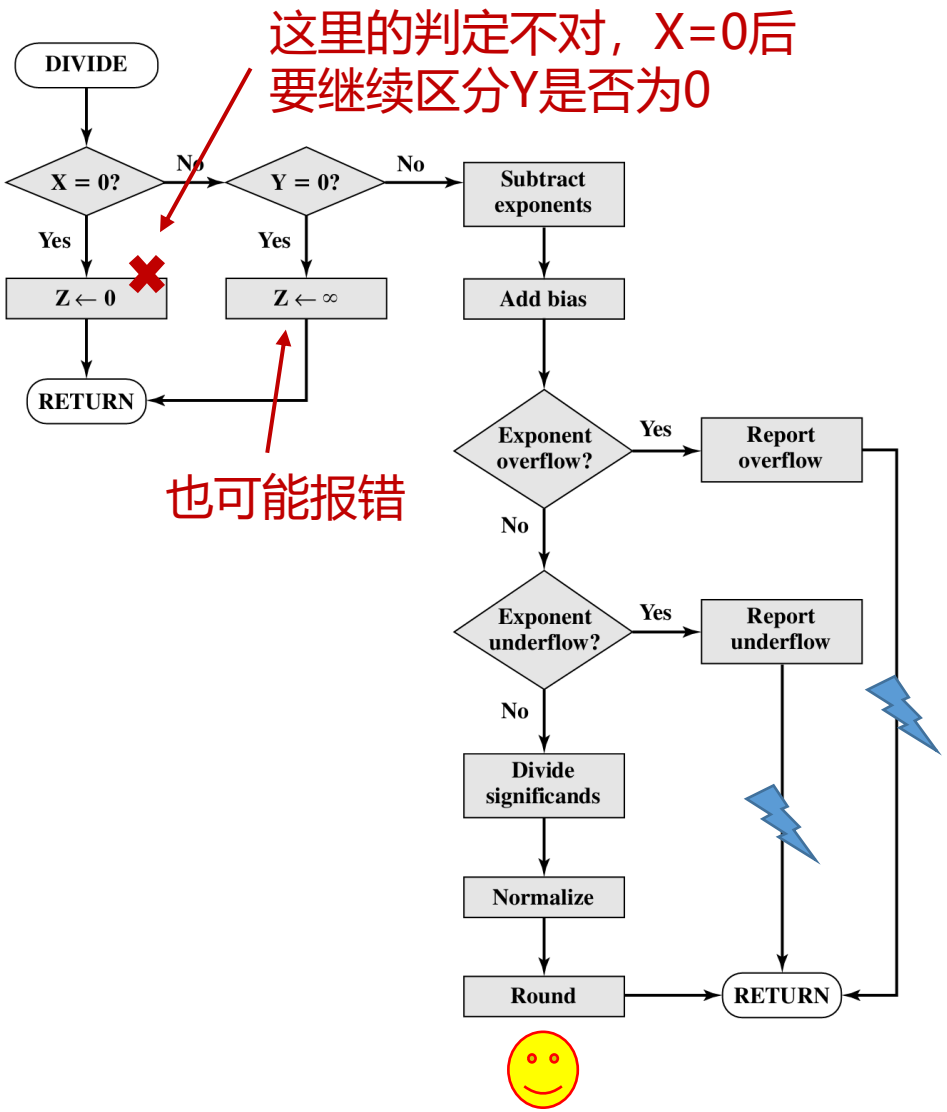


乘积高两位为01，不用处理

0 01111100 110…00 (21)

除法

- 检查0：
  - 如果除数为0，则报告出错，或将结果设置为无穷大
  - 如果被除数是0，则结果是0
- 被除数的阶值减除数的阶值，加上一个偏移量
- 有效值相除
- 结果规格化和舍入处理
  - IEEE 754标准中，只有左规
    - 高两位为01则左规一位
    - 高两位为10、11则无需处理

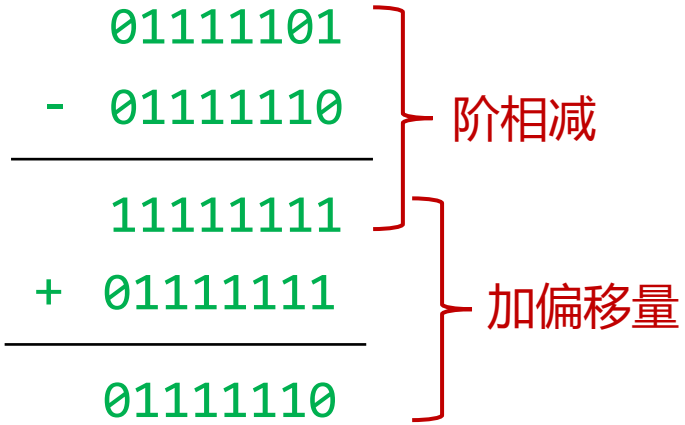


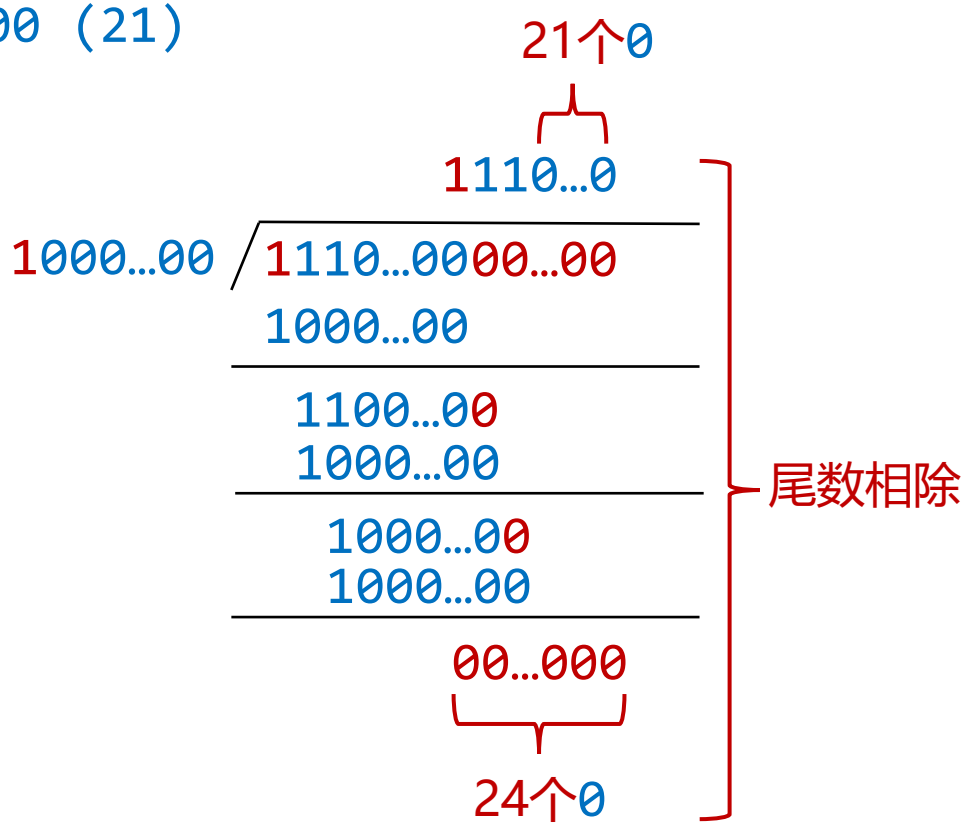
除法示例

0.4375/0.5=0.875

0.5　　0 01111110 000…00 (23)

0.4375　　0 01111101 110…00 (21)





0 01111110 110…00 (21)

商高两位为11，不用处理

精度考虑：附加位

- 附加位
  - 寄存器的长度几乎总是大于有效值位长与一个隐含位之和
  - 寄存器包含的这些附加位，也称为保护位
  - 保护位用0填充，用于扩充有效值的右端

$$x = 1.00\ldots00 \times 2^1, y = 1.11\ldots11 \times 2^0$$

$$\begin{aligned} x &= 1.000.....00 \times 2^1 \\ -y &= 0.111.....11 \times 2^1 \\ z &= 0.000.....01 \times 2^1 \\ &= 1.000.....00 \times 2^{-22} \end{aligned}$$

不使用附加位

$$\begin{aligned} x &= 1.000.....00\ 0000 \times 2^1 \\ -y &= 0.111.....11\ 1000 \times 2^1 \\ z &= 0.000.....00\ 1000 \times 2^1 \\ &= 1.000.....00\ 0000 \times 2^{-23} \end{aligned}$$

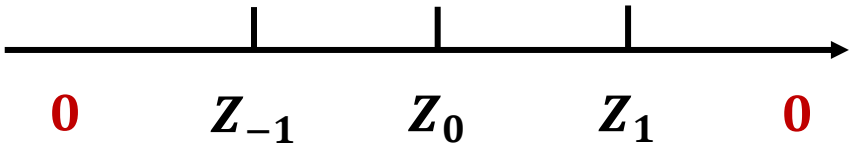
使用附加位

- **IEEE754规定：中间结果须在右边加2个附加位（Guard & Round）**
  - **保护位（Guard）**：在尾数右边的位，左规时被移到尾数中
  - **舍入位（Round）**：在保护位右边的位，作为舍入的依据

精度考虑：舍入

- **舍入**
  - 对有效值操作的结果通常保存在更长的寄存器中
  - 当结果转换回浮点格式时，必须要去掉多余的位
    - **就近舍入（默认方式）**：结果被舍入成最近的可表示的数
      - **非中间值**：0舍1入
      - **中间值**：强制结果为偶数

      01：舍
      11：入
      10：强制结果为偶数
    - **朝+∞舍入**：结果朝正无穷大方向向上舍入，舍入为Z_1
    - **朝−∞舍入**：结果朝负无穷大方向向下舍入，舍入为Z_{-1}
    - **朝 0 舍入**：结果朝 0 舍入，正数Z_{-1}，负数Z_1



Z_{-1}和Z_1是结果Z_0的相邻可表示数

就近舍入示例：
1.110101 -> 1.1101
1.110111 -> 1.1110
1.110110 -> 1.1110
1.111010 -> 1.1110

精度考虑：舍入

为什么就近舍入更好？中间值为什么强制结果为偶数？

Exactly Rounded Operations

When floating-point operations are done with a guard digit, they are not as accurate as if they were computed exactly then rounded to the nearest floating-point number. Operations performed in this manner will be called *exactly rounded*.[8] The example immediately preceding Theorem 2 shows that a single guard digit will not always give exactly rounded results. The previous section gave several examples of algorithms that require a guard digit in order to work properly. This section gives examples of algorithms that require exact rounding.

So far, the definition of rounding has not been given. Rounding is straightforward, with the exception of how to round halfway cases; for example, should 12.5 round to 12 or 13? One school of thought divides the 10 digits in half, letting {0, 1, 2, 3, 4} round down, and {5, 6, 7, 8, 9} round up; thus 12.5 would round to 13. This is how rounding works on Digital Equipment Corporation's VAX computers. Another school of thought says that since numbers ending in 5 are halfway between two possible roundings, they should round down half the time and round up the other half. One way of obtaining this 50% behavior to require that the rounded result have its least significant digit be even. Thus 12.5 rounds to 12 rather than 13 because 2 is even. Which of these methods is best, round up or round to even? Reiser and Knuth [1975] offer the following reason for preferring round to even.

Theorem 5

Let x and y be floating-point numbers, and define $x_0 = x$, $x_1 = (x_0 \ominus y) \oplus y$, ..., $x_n = (x_{n-1} \ominus y) \oplus y$. If $\oplus$ and $\ominus$ are exactly rounded using round to even, then either $x_n = x$ for all n or $x_n = x_1$ for all $n \geq 1$. z

To clarify this result, consider $\beta = 10$, $p = 3$ and let $x = 1.00$, $y = -.555$. When rounding up, the sequence becomes

$x_0 \ominus y = 1.56$, $x_1 = 1.56 \ominus .555 = 1.01$, $x_1 \ominus y = 1.01 \oplus .555 = 1.57$,

and each successive value of x_n increases by .01, until $x_n = 9.45$ ($n \leq 845$)[9]. Under round to even, x_n is always 1.00. This example suggests that when using the round up rule, computations can gradually drift upward, whereas when using round to even the theorem says this cannot happen. Throughout the rest of this paper, round to even will be used.

One application of exact rounding occurs in multiple precision arithmetic. There are two basic approaches to higher precision. One approach represents floating-point numbers using a very large significand, which is stored in an array of words, and codes the routines for manipulating these numbers in assembly language. The second approach represents higher precision floating-point numbers as an array of ordinary floating-point numbers, where adding the elements of the array in infinite precision recovers the high precision floating-point number. It is this second approach that will be discussed here. The advantage of using an array of floating-point numbers is that it can be coded portably in a high level language, but it requires exactly rounded arithmetic.

The key to multiplication in this system is representing a product xy as a sum, where each summand has the same precision as x and y. This can be done by splitting x and y. Writing $x = x_h + x_l$ and $y = y_h + y_l$, the exact product is

$xy = x_h y_h + x_h y_l + x_l y_h + x_l y_l$.

If x and y have p bit significands, the summands will also have p bit significands provided that x_l, x_h, y_h, y_l can be represented using $\lceil p/2 \rceil$ bits. When p is even, it is easy to find a splitting. The number $x_0.x_1 \ldots x_{p-1}$ can be written as the sum of $x_0.x_1 \ldots x_{p/2-1}$ and $0.0 \ldots 0x_{p/2} \ldots x_{p-1}$. When p is odd, this simple splitting method will not work. An extra bit can, however, be gained by using negative numbers. For example, if $\beta = 2$, $p = 5$, and $x = .10111$, x can be split as $x_h = .11$ and $x_l = -.00001$. There is more than one way to split a number. A splitting method that is easy to compute is due to Dekker [1971], but it requires more than a single guard digit.

[1] David Goldberg. 1991. *What every computer scientist should know about floating-point arithmetic.* ACM Comput. Surv. 23, 1 (March 1991), 5–48. https://doi.org/10.1145/103162.103163

[2] Reiser, John F. and Donald Ervin Knuth. *Evading the Drift in Floating-Point Addition.* Inf. Process. Lett. 3 (1975): 84-87.

精度考虑：数据类型

- X为int型，F为float型，D为double型，判断下列关系是否恒为真：
  - 否 • X == (int)(float) X
  - 是 • X == (int)(double) X
  - 是 • F == (float)(double) F
  - 否 • D == (float) D
  - 否 • X == (float) X　　浮点数**是**，整数**否**
  - 否 • X * X >= 0
  - 是 • F * F >= 0
  - 否 • (D+F) – D == F
  - 否 • (D+F) – F == D
  - 否 • 1/3 == 1/3.0

精度考虑：数据类型程序示例

$D = 2^k, F = 1.0$　　$F = D + F - D$从$k = 53$开始为假

hello.cpp

```cpp
#include <iostream>
#include<cmath>
using namespace std;

int main()
{
    int p=1;
    for(int i=0;i<100;i++){

        double d = 1*pow(2,i);
        float f = 1.0;
        if(!(d+f-d == f)){
            printf("%d\n",i);
            p=0;
        }
    }
    if(p==1){
        printf("equal\n");
    }
   return 0;
}
```

C++ (GCC 9.2.0)　请输入命令行参数　运行(F9)

Output Empty

标准输出:

```
53
54
55
```

精度考虑：数据类型程序示例

$D = \frac{1}{2^k}, F = 1.0$　　$F = D + F - D$仅当$k = 53$为假

hello.cpp

```cpp
#include <iostream>
#include<cmath>
using namespace std;

int main()
{
    int p=1;
    for(int i=0;i<100;i++){

        double d = 1/pow(2,i);
        float f = 1.0;
        if(!(d+f-d == f)){
            printf("%d\n",i);
            p=0;
        }
    }
    if(p==1){
        printf("equal\n");
    }
   return 0;
}
```

C++ (GCC 9.2.0)　请输入命令行参数　运行(F9)

Output　Empty

标准输出:

```
53
```

精度考虑：数据类型程序示例

$D = 2^k, F = 1.0$　　$F = D - D + F$恒为真

hello.cpp

```cpp
#include <iostream>
#include<cmath>
using namespace std;

int main()
{
    int p=1;
    for(int i=0;i<100;i++){

        double d = 1*pow(2,i);
        float f = 1.0;
        if(!(d-d+f == f)){
            printf("%d\n",i);
            p=0;
        }
    }
    if(p==1){
        printf("equal\n");
    }
   return 0;
}
```

C++ (GCC 9.2.0)　请输入命令行参数　▶ 运行(F9)

Output　Empty

标准输出:

```
equal
```

精度考虑：数据类型程序示例

$D = \frac{1}{2^k}, F = 1.0$　　$F = D - D + F$恒为真

hello.cpp

C++ (GCC 9.2.0)　请输入命令行参数　运行(F9)

```cpp
#include <iostream>
#include<cmath>
using namespace std;

int main()
{
    int p=1;
    for(int i=0;i<100;i++){

        double d = 1/pow(2,i);
        float f = 1.0;
        if(!(d-d+f == f)){
            printf("%d\n",i);
            p=0;
        }
    }
    if(p==1){
        printf("equal\n");
    }
   return 0;
}
```

Output　Empty

标准输出:

```
equal
```

精度考虑：舍入示例

- 假设用16位表示一个浮点数，其中1位为符号，9位为尾数有效值，6位为阶值（偏移量为31），如何表示652.13 和 -7.48?

+ 652.13	0 101000 010001100	+ 652.0
-7.48	1 100001 110111101	- 7.4765625

- 假设 ALU 有 16 位，分别采取使用附加位和不使用附加位的方法计算652.13 + (-7.48)

101000 – 100001 = 000111

```
   1 010001100
 -   0 000000111
-------------------
   1 010000101
```

0 101000 010000101

645.0

```
   1 010001100 000000
 -   0 000000111 011110
-------------------------
   1 010000100 100010
```

0 101000 010000100 644.0

或 0 101000 010000101 645.0

牢记IEEE 754是因为它不完美

尽量简化运算步骤

- 每一个运算符都意味着一次运算
- 每一次运算，都有可能导致精度损失
- 独立地去看每一次运算

低精度和一个高精度在运算时，低精度的数变成了高精度(补0)

- 对阶时，有效值的位数和高精度一致
- 计算的结果为高精度

运算后注意舍入

- 如需左规，先左规再舍入
- C++默认为就近舍入

运算时，警惕二进制不能精确表示的数

- 十进制常见的有限小数，二进制很可能无法精确表示
- 例如 (double)0.3+ (double)0.9 - (double)0.3 != (double)0.9

运算时，警惕2^k

- 既要警惕阶的取值k
- 也要警惕它的尾数是1.0...0

总结

- 浮点算数运算
  - 加法
  - 减法
  - 乘法
  - 除法
- 精度考虑

谢谢

bohanliu@nju.edu.cn

南京大學
NANJING UNIVERSITY