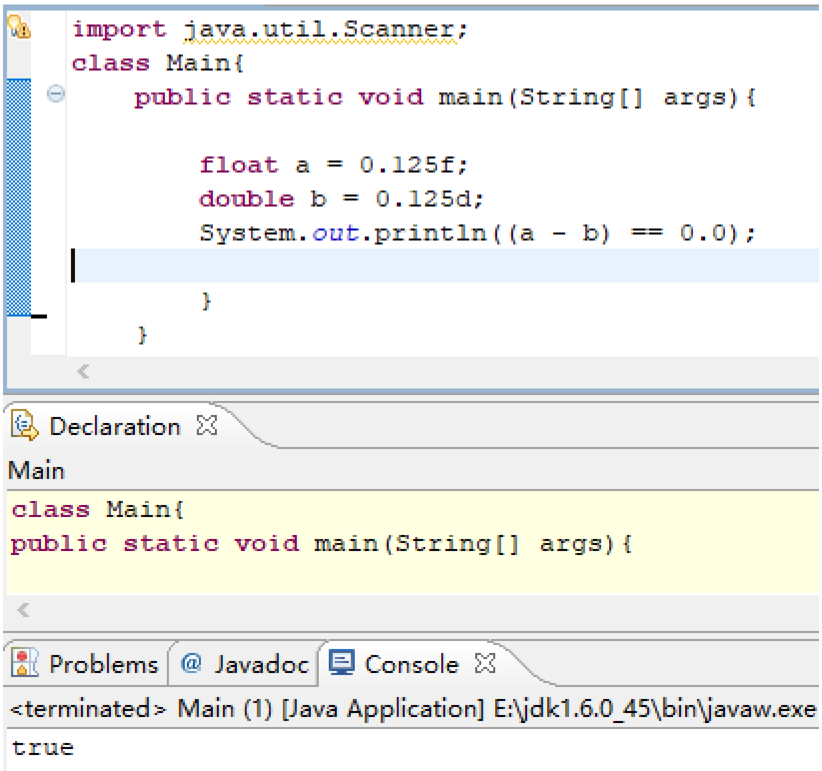
一些关于浮点运算的示例

1. 单精度与双精度的计算[1]

```
class Main{
    public static void main (String[ ] args){
        float a = 0.125f;
        double b = 0.125d;
        System.out.printIn((a - b) == 0.0);
    }
}
```

输出结果是什么？



原因：0.125f和0.125d不存在精度损失。

```
#include <studio.h>
#define EPSILON 0.0001   // Define your own tolerance
#define FLOAT_EQ(x,v) (((v - EPSILON) < x) && (x <( v + EPSILON)))

int main() {
   float a, b, c;

   a = 1.345f;
   b = 1.123f;
   c = a + b;
   // if (FLOAT_EQ(c, 2.468)) // Remove comment for correct result
   if (c == 2.468)            // Comment this line for correct result
```

```
        printf_s("They are equal.\n");
    else
        printf_s("They are not equal! The value of c is %13.10f "
                 "or %f",c,c);
}
```

输出结果是什么？ [2]

OUTPUT:

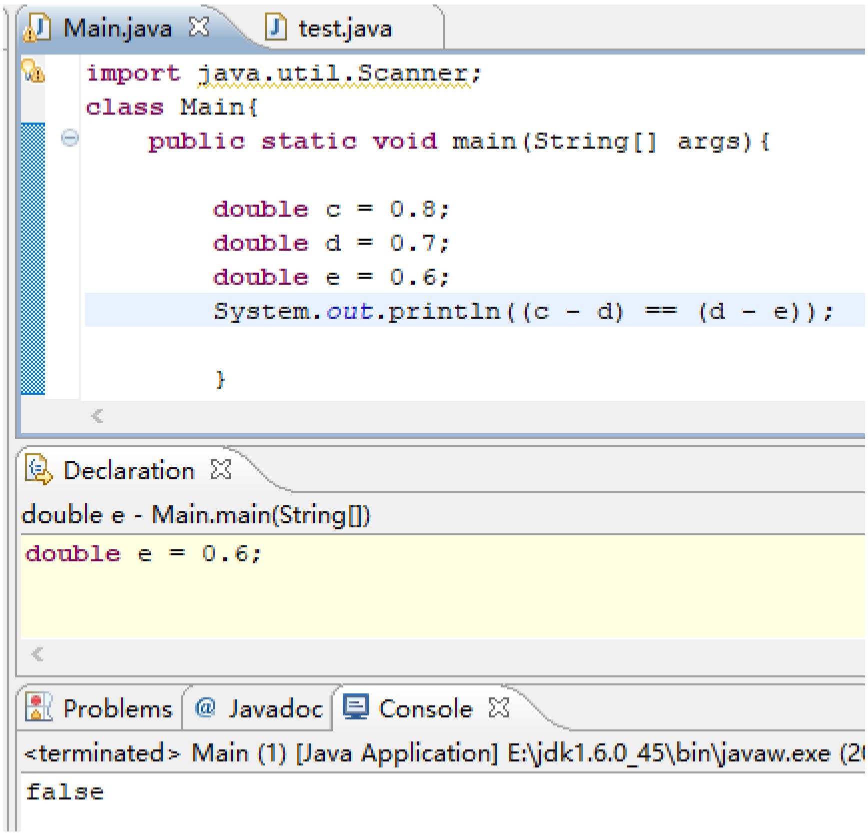
```
They are not equal! The value of c is  2.4679999352 or 2.468000
```

原因： 使用的数字之间存在类型不匹配（例如，混合浮点数和双精度数）。

2. double计算一定准确吗?

```
class Main{
    public static void main (String[] args)
    {
        double c = 0.8;
        double d = 0.7;
        double e = 0.6;
        System.out.printIn((c - d) == (d - e));
    }
}
```

c-d与d-e的结果相等吗?

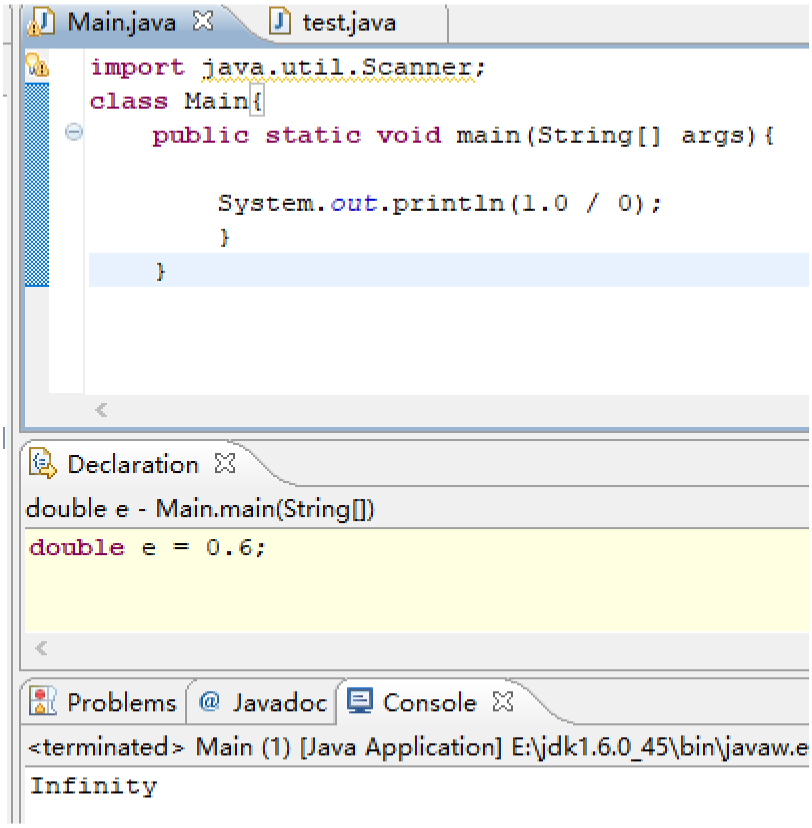


原因：浮点数无法精确表示，存在精度损失。十进制数的二进制表示可能不准确。

3. 除0会抛出异常吗?

```
class Main{
    public static void main (String[] args)
    {
        System.out.printIn(1.0 / 0);
    }
}
```

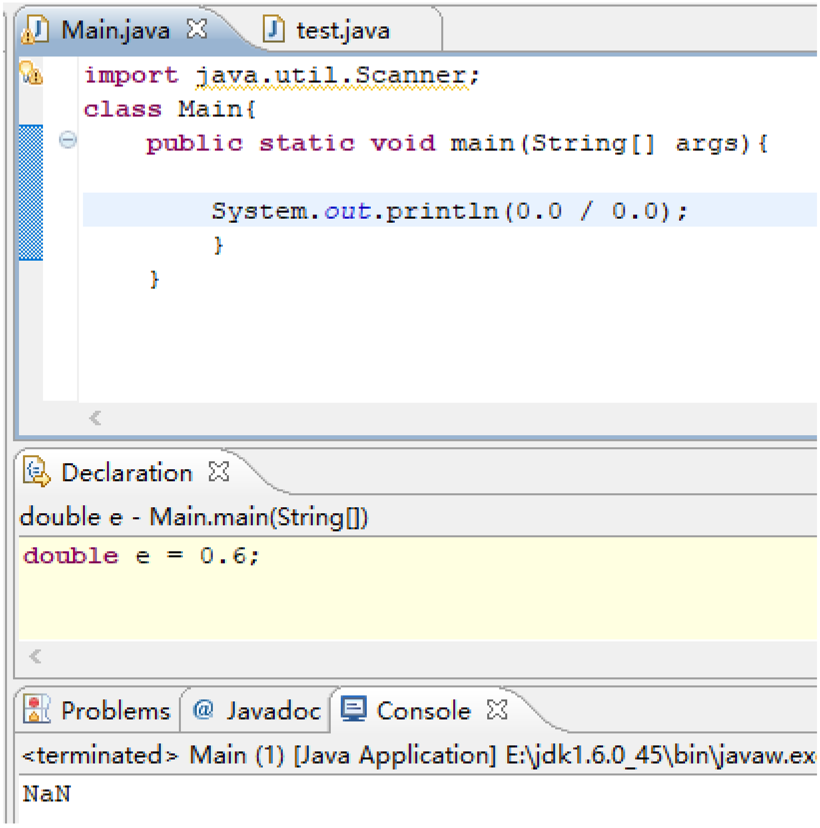
程序会抛出异常吗?



原因：float、double数据类型支持无穷大。

```
class Main{
    public static void main (String[] args)
    {
        System.out.printIn(0.0 / 0.0);
    }
}
```

程序会抛出异常吗?

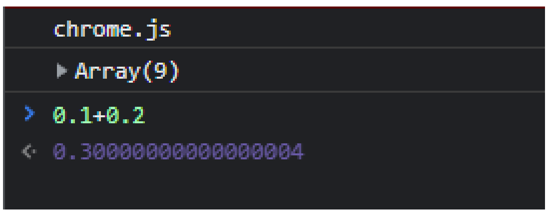


原因：double源码定义

```
public static final double POSITIVE_INFINITY = 1.0 / 0.0;
/**
 * 一个常数，保持类型的负无穷大
*/
public static final double NEGATIVE_INFINITY = -1.0 / 0.0;
/**
  * 一个常数,非数值类型
*/
public static final double NaN = 0.0d / 0.0;
```

计算机中的数据表示方法

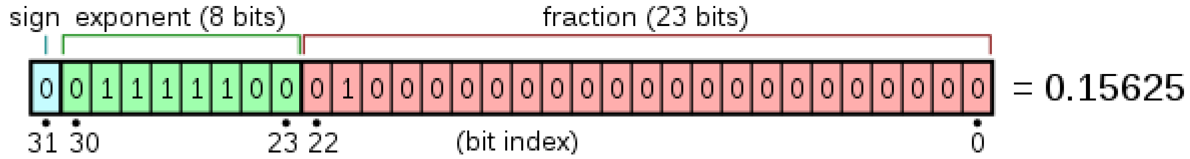
示例

- 在Chrome中的Console里，键入表达式`0.1+0.2`，键入回车
- 结果显示是`0.30000000000000004`
- 

$IEEE754$[4]

浮点数的一般格式

IEEE754浮点标准指定浮点的二进制表示格式如下，



- 浮点：浮点数的符号（1bit），0表示正，1表示负。
- 指数：以2为基数（8bit）置于指数字段中的值。
- 分数：一般在0到1之间的值，放在分数字段（23bit）中。
- 从编码通过以下公式计算表示数值。 $$(-1)^{sign}\times(1+fraction)\times2^{exponent - 127}$$

常用格式[1]

IEEE754规定了四种表示浮点数值的方式：单精确度（32位）、双精确度（64位）、延伸单精确度（43比特以上，很少使用）与延伸双精确度（79比特以上，通常以80位实现）。其中最常用的就是32位单精度浮点数和64位双精度浮点数。

单精度浮点数在计算机存储器中占用4个字节（32 bits） 双精度浮点数(double)使用 64 位（8字节） 来存储一个浮点数

格式	单精度	双精度
指数域(E)	8 bit	11 bit
尾数域(f)	23 bit	52 bit
有效位数	24 bit	53 bit
移码偏移值（bias）	127	1023
规范化E的最大值	127 （全1是特殊值）	1023（全1是特殊值）
规范化E的最小值	-126（全0是特殊值）	-1022（全0是特殊值）
最小正规范数	1.175×10^{-38}	2.225×10^{-308}
最大正规范数	$3,402\times10^{38}$	1.797×10^{308}

特殊值

- 非规范化数（Denormalized Number）
  - 定义：指数域全0，尾数域不为0（即去掉隐含整数域为1的约定）
  - 意义：可以保存绝对值更小的数，所有可以表示的浮点数的差值都可以表示
- ±0
  - 定义：指数域全0，尾数域全0（去掉隐含整数域为1的约定）。
  - 意义：在涉及无穷的运算中避免丢失符号信息。如$\dfrac{1}{\dfrac{1}{x}}=x$，如果0不区分正负，在$x=\pm\infty$时不成立。

- $±∞$
  - 定义：指数域全1，尾数全0
  - 意义：用于表达计算中产生的上溢（overflow），使得计算中出现上溢不止于终止计算，
  - 产生：除了NaN外的非零值除以0，其结果为正负无穷。
- NaN（Not a Number）
  - 定义：指数域全1，尾数域不为0
  - 意义：表示计算中的错误情况，例如$\dfrac{0.0}{0.0}$，使得计算中出现错误不止于终止计算
  - 特点：NaN是无序的，比较操作符在任一操作数为NaN是为false，!=在任一操作数为NaN时为true，这意味着NaN != NaN.

计算机表示实数的步骤[5]

- 将实数转化为二进制格式
  - 这个步骤可能损失精度，换句话说，有些数会损失精度，而有些数不会，**这取决于表示这个数所需要的信息量和浮点数的存储格式**
- 将二进制转换为科学计数法表示
- 转换为`IEEE754`标准格式
  - 大于浮点数可以表示的最大绝对值：上溢（溢出到$±∞$）
  - 小于浮点数可以表示的最小绝对值：下溢（溢出到$±0$）
  - 尾数有效位数超过尾数域位数（另外还有隐含的整数位1）：舍入误差

为什么浮点数计算不准确?

浮点数的精度有限，而十进制转二进制表示不一定准确 [2]

- 例如[3]，1.0（十进制）编码为$00111111100000000000000000000000$
  - 即$$0(sign)01111111(exponent)00000000000000000000000(fraction)$$ 通过转换公式计算二进制数，得到结果$$1.00000000000000000000000(binary) × 2^{0}$$
  - 可以添加或减去的最小数字是$$0.00000000000000000000001(binary) = 0.00000011920928955078(decimal)$$
  - 如果添加/减去小于$0.00000011920928955078(decimal)$的数字，结果不会改变。这是使用单精度浮点数时1.0的精度。
- 例如[3]，$1000000.0(decimal)$。它被编码为$$01001001011101000010010000000000$$
  - 即$$0(sign)10010010(exponent)11101000010010000000000(fraction)$$
  - 其二进制数为$$1.11101000010010000000000(binary)×2^{19}$$，即$$11110100001001000000.0000(binary)$$
  - 可以加减的最小数字是$0.0001(binary)=0.0625(decimal)$
  - 在这种情况下，与1.0相比，精度要低很多。

浮点数的精度有限，数学运算的规则和性质并不直接适用于浮点运算。

- 例如，下表显示单个精度值A、B和C，以及使用不同的结合律计算它们的和的数学精确值。
- $A = 2^1 × 1.00000000000000000000001$

$B = 2^0 × 1.00000000000000000000001$

$C = 2^3 × 1.00000000000000000000001$

$(A+B)+C = 2^3 × 1.01100000000000000000001011$

$A+(B+C) = 2^3 × 1.01100000000000000000001011$

数学上，结合律是成立的。

设$rn(x)$表示x上的一个舍入步骤。

$A+B = 2^1 × 1.10000000000000000000000110000...$

$rn(A+B) = 2^1 × 1.10000000000000000000010$

$B+C = 2^3 × 1.00100000000000000000000100100...$

$rn(B+C) = 2^1 × 1.00100000000000000000001$

$A+B+C = 2^3 ×1.01100000000000000000000101100...$

$rn(rn(A+B)+C) = 2^3 × 1.01100000000000000000010$

$rn(A+rn(B+C)) = 2^3 × 1.01100000000000000000001$

- `IEEE754`计算的结果不仅与精确的数学结果不一样，而且$rn(rn(A+B)+C)$和$rn(rn(A+B)+C)$更接近正确的数学结果

浮点数计算真的不准确吗？

`strict IEEE 754 vs fast math mode`

- 编译器可以选择`strict IEEE 754 mode`或`fast mode`中一种模式来进行浮点数计算
- **`strict IEEE 754`使得浮点计算变得可预测并且尽可能精确。** 严格的IEEE754语义不允许使用可能影响结果精度的代数变换的近似指令和优化。目标是使计算精确和可预测。理想情况下，如果启用了 IEEE 754 严格语义标志，则使用不同编译器编译的同一程序应该产生相同的结果[6]。
    - 因此编译器在IEEE754语义下处理浮点数时，在进行优化时必须小心，因为旨在提高速度的微小更改可能会导致程序输出不同的结果。
- `fast math mode`[7]。大多数时候，由于编译优化导致的结果差异很小。正因为如此，几乎所有编译器可以启用`fast math mode`，这样可以产生更快的代码，但不保证结果的精度。
    - `fast math mode`可以**显著提升性能**，例如在游戏中，玩家不会在乎npc皮肤的颜色上的红色分量是否产生了偏离，但是玩家可能会非常在乎游戏的帧率。
    - 但是，往往在科学领域，由于精度损失和舍入误差导致的微小差异会随着时间的推移而累积，并可能导致不正确的结果。这些程序通常会在非`fast math mode`下进行编译。
    - `fast math mode`下，编译器可以做三类优化。
        - 由算术规则引起的优化，（例如`(-a)* (-b)=a * b`;
        - 使用近似指令引起的优化，例如`a / b = a * (1 / b), (1 / b)`使用近似倒数除法指令计算表达式，这比常规除法快得多；

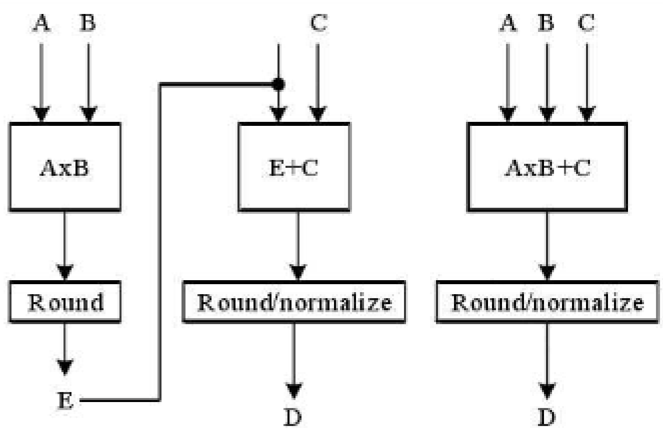
- 与浮点错误管理相关的优化。这为优化为其他优化打开了大门，如公共子表达式消除、循环不变代码运动、矢量化等。

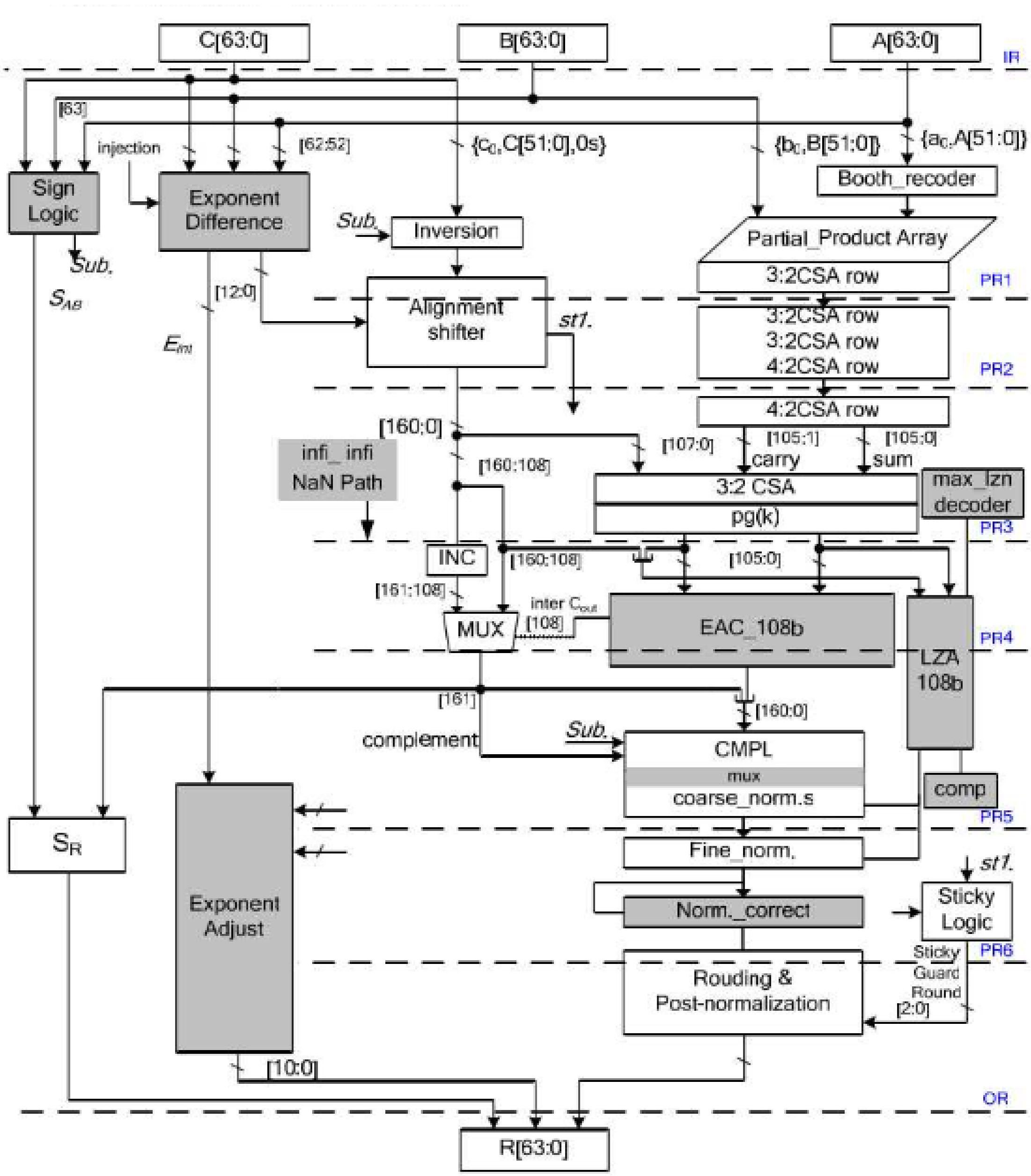
怎么提高浮点计算的准确性呢?

- 浮点融合乘加[8]（Fused Multiple-Add）：即浮点乘法加法运算，在一步执行完毕，采用单次舍入。例如$a + b × c$，未融合乘加将计算乘积$b × c$，将其舍入到N有效位，将结果加到a，然后舍入到N个有效位置；融合乘加将计算整个表达式$a + (b × c)$在将最终结果向下舍入到N个有效位之前达到其全精度。在2008年，IEEE754更新加入了FMA操作。

  +快速FMA可以加快许多涉及乘积累积的计算，并提高计算准确性。 + 点积 + 矩阵乘法 + 多项式评估 + 牛顿计算函数的方法 + 卷积和人工神经网络

  - 浮点融合乘加部件[9]
    - 浮点乘加部件的延时不会大于顺序执行浮点乘和浮点加的总延时；
    - 浮点乘加部件的实现不会大幅度增加浮点加法的延时；
    - 浮点乘加部件的硬件开销不会太大。

  - 



- 其他替代$IEEE754$的方式：Posit [10]

Reference

- [1] 「计算机原理」| 为什么浮点数运算不精确？（阿里笔试）
- [2] Why Floating-Point Numbers May Lose Precision
- [3] Why Floating-Point Numbers are not Always Accurate
- [4] IEEE 754
- [5] Why are floating point numbers inaccurate?
- [6] Consistency of Floating-Point Results using the Intel® Compiler
- [7] Semantics of Floating Point Math in GCC
- [8] Multiply–accumulate operation

- [9] 谢启华
- [10] Posit: A Potential Replacement for IEEE 754