COA2023-programming04

> 略有小成

1 实验要求

在FPU类中实现4个方法，具体如下

1.计算两个浮点数真值的和

```
public DataType add(DataType src, DataType dest)
```

2.计算两个浮点数真值的差

```
public DataType sub(DataType src, DataType dest)
```

3.计算两个浮点数真值的积 dest × src

```
public DataType mul(DataType src, DataType dest)
```

4.计算两个浮点数真值的商 dest ÷ src

注意：除数为0，且被除数不为0时要求能够正确抛出ArithmeticException异常

```
public DataType div(DataType src, DataType dest)
```

2 实验指导

2.1 代码实现要求

本次实验中，我们仍然**明确禁止**各位采用直接转浮点数进行四则运算来完成本次实验。

2.2 浮点数加减运算代码实现流程

对于浮点数加减运算的流程，课件与课本都有很详细的讲解，在此不再赘述。对于代码实现层面，大致可分为以下几步:

1. 处理边界情况(NaN, 0, INF)
2. 提取符号、阶码、尾数
3. 模拟运算得到中间结果
4. 规格化并舍入后返回

2.2.1 处理边界情况

在框架代码中，我们提供了cornerCheck方法来检查0和INF的情况，大家直接调用即可。

此外，对于NaN的情况，我们提供了一个基于正则表达式的处理方案，可用如下代码进行检查：

```
String a = dest.toString();
String b = src.toString();
if (a.matches(IEEE754Float.NaN_Regular) || b.matches(IEEE754Float.NaN_Regular)) {
    return new DataType(IEEE754Float.NaN);
}
```

在util.IEEE754Float类中，我们提供了NaN的正则表达式，对于正则表达式的作用机制大家可以自行查阅。

在本次作业中，大家直接调用cornerCheck方法以及上述正则表达式的解决方案即可轻松完成第一步：对边界情况的检查。

如果你顺利实现第一步，应该可以在平台上拿到部分分数。

2.2.2 提取符号、阶码、尾数

在本次作业中，我们使用IEE754浮点数运算标准，模拟32位单精度浮点数，符号位、指数部分与尾数部分分别为1、8、23位，同时使用3位保护位(GRS保护位)，大家经过简单操作即可完成这一步。

这一步有三个需要特殊处理的地方：

1. 当有一个操作数提取出的阶码为全1时，应该返回其本身。（为什么？考虑无穷加减其他数的结果）
2. 当提取出的阶码为全0时，说明该操作数是一个非规格化数，此时应该对阶码+1使其真实值变为1，以保证后面的对阶操作不会出错。（为什么？可以考察IEEE754浮点数标准中阶码为0和阶码为1分别表示2的多少次方）
3. 在这一步中不要忘记尾数的最前面添加上隐藏位，规格化数为1，非规格化数为0。所以提取结束后尾数的位数应该等于1+23+3=27。

2.2.3 模拟运算得到中间结果

这一步是要求大家实现的重要步骤。这一步主要做两件事情。

第一件事情是对阶，采用小阶向大阶看齐的方式，小阶增加至大阶，同时尾数右移，保证对应真值不变。注意，基于GRS保护位标准，尾数右移时不能直接将最低位去掉。我们提供了对尾数进行右移的方法，方法签名如下：

```
private String rightShift(String operand, int n)
```

第一个参数为待右移的尾数，第二个参数为右移的位数。请大家每次对尾数进行右移操作时都调用这个方法，否则很可能出现最后对保护位进行舍入后，尾数与结果差1的情况。

第二件事情是尾数相加或相减。这一步相对简单，大家可以调用提供的ALU类进行操作，也可以拷贝上次实验中自己写的代码进行操作。

2.2.4 规格化并舍入后返回

在这一步中，我们只要求大家进行规格化的处理。这里需要大家思考的是，在上一步运算结束后，有哪几种情况会导致结果不符合规格化的条件?

1. 当运算后尾数大于27位时，此时应该将尾数右移1位并将阶码加1。
    - 注意，这个将阶码加1的操作可能会导致阶码达到"11111111"，导致溢出。针对这种阶码上溢的情况，应该返回什么?
2. 当运算后尾数小于27位时，此时应该不断将尾数左移并将阶码减少，直至尾数达到27位或阶码已经减为0。
    - 注意，若阶码已经减为0，则说明运算得到了非规格化数，此时应该怎么办？（可以考察阶码为0000 0001，尾数为0.1000 0000 0000 0000 0000 0000 00的浮点数该如何正确表示）

我们提供了相关的本地用例，大家可以仔细揣摩其中的奥妙。

对于规格化后的舍入操作，我们不要求掌握GRS保护位相关的舍入操作，感兴趣的同学可以自行查阅。我们提供了舍入操作的函数如下

```
private String round(char sign, String exp, String sig_grs)
```

请注意，在调用此方法前，请确保你传入的参数已经进行了规格化，务必确保传入的符号位为1位，阶码为8位，尾数为1+23+3=27位。

在此方法中，我们已经对GRS保护位进行了相应的处理并完成舍入，返回的结果即为32位的字符串，转化为DataType类型后即可通过测试。

至此，你已经完成了浮点数加减法的全部工作(・ω・)ノ

2.3 GRS保护位

注：本部分内容不需要掌握

GRS保护位机制使用3个保护位辅助完成舍入过程。一个27位的尾数可表示为

```
1(0) . m1 m2 m3 …… m22 m23 G R S
```

这里G为保护位（guard bit），用于暂时提高浮点数的精度。R为舍入位（rounding bit），用于辅助完成舍入。S为粘位（sticky bit）。粘位是R位右侧的所有位进行逻辑或运算后的结果，简单来说，在右移过程中，一旦粘位被置为1（表明右边有一个或多个位为1）它就将保持为1。

在round函数中，根据GRS位的取值情况进行舍入，舍入算法采用就近舍入到偶数。简单来说，在进行舍入时分为以下三种情况。

1. 当GRS取值为"101" "110" "111"时，进行舍入时应在23位尾数的最后一位上加1。
2. 当GRS取值为"000" "001" "010" "011"时，进行舍入时直接舍去保护位，不对23位尾数进行任何操作。
3. 当GRS取值为"100"时，若23位尾数为奇数则加1使其变成偶数，若23位尾数为偶数则不进行任何操作。

2.4 浮点数乘除运算代码实现流程

在充分掌握了浮点数的加减运算后，浮点数的乘除运算就十分简单了。其基本步骤和加法类似，相比加法运算，还可以免去对阶的过程。基本流程仍然可以分为以下四步：

1. 处理边界情况(NaN, 0, INF)
2. 提取符号、阶码、尾数
3. 模拟运算得到中间结果
4. 规格化并舍入后返回

2.4.1 处理边界情况

在本部分实验中，你仍然需要使用cornerCheck和util.IEEE754Float类中提供的NaN正则表达式来处理边界情况。

注意，在除法运算中，还需要额外判断除数为0且被除数不为0的情况。

2.4.2 提取符号、阶码、尾数

在本次作业中，我们使用IEE754浮点数运算标准，模拟32位单精度浮点数，符号位、指数部分与尾数部分分别为1、8、23位，同时使用3位保护位(GRS保护位)，大家经过简单操作即可完成这一步。

这一步有三个需要特殊处理的地方：

1. 当有一个操作数提取出的阶码为全1时，应该返回正无穷或负无穷，注意符号需要额外判断。（为什么？考虑无穷乘其他数的结果）
2. 当提取出的阶码为全0时，说明该操作数是一个非规格化数，此时应该对阶码+1使其真实值变为1，以保证后面的对阶操作不会出错。（为什么？可以考察IEEE754浮点数标准中阶码为0和阶码为1分别表示2的多少次方）
3. 在这一步中不要忘记尾数的最前面添加上隐藏位，规格化数为1，非规格化数为0。所以提取结束后尾数的位数应该等于1+23+3=27。

聪明的你是不是发现了，至此的所有操作都跟上一部分几乎一模一样。该怎么~~Ctrl C + Ctrl V~~操作就不用我多说了吧。

2.4.3 模拟运算得到中间结果

乘除法运算对于符号位的计算非常简单，直接可以根据两个操作数的符号位得到结果的符号位，在此不作更深入的讲解。

对于阶码的计算，与加减法运算不同的是，乘除法运算不再需要对阶操作，而是直接计算结果阶码。其计算过程分别为

- 乘法：尾数相乘，阶码相加后减去偏置常数
- 除法：尾数相除，阶码相减后加上偏置常数

对于尾数的计算，在此需要大家分别实现27位无符号数的乘法与除法，运算流程可以参考课件。相信有了ALU的乘除法基础，这一步不会花费太多时间。

需要注意的是，对于27位乘法运算，返回的结果是54位的乘积。由于两个操作数的隐藏位均为1位，所以乘积的隐藏位为2位（为什么？）。为了方便后续操作，需要通过阶码加1的方式来间接实现小数点的左移，修正这个误差，以保证尾数的隐藏位均为1位。

2.4.4 规格化并舍入后返回

在这一步中，我们仍然只要求大家进行规格化的处理。相比于加减法运算，乘除法的运算结果破坏规格化的情况更多，增加了阶码为负数的情况。简单分类如下：

1. 运算后54位尾数的隐藏位为0且阶码大于0，此时应该不断将尾数左移并将阶码减少，直至尾数隐藏位恢复为1或阶码已经减为0。
2. 运算后阶码小于0且54位尾数的前27位不全为0，此时应该不断将尾数右移并将阶码增加，直至阶码增加至0或尾数的前27位已经移动至全0。
3. 经过上述两步操作后，尾数基本恢复规格化，但阶码仍有可能破坏规格化，分为以下三种情况：
    - 阶码为"11111111"，发生阶码上溢，此时应该返回什么？
    - 阶码为0，则说明运算得到了非规格化数，此时应该将尾数额外右移一次，使其符合非规格化数的规范。（为什么？可以考察阶码为0000 0001，尾数为0.1000 0000 0000 0000 0000 0000 00的浮点数的规格化过程）
    - 阶码仍小于0，发生阶码下溢，此时又应该返回什么？

可能大家看到这里觉得很乱，没关系，我们提供的fpuMulTest9涵盖了这里面的所有情况，大家可以在debug的过程中体会其中的玄机。以上规格化过程可用伪代码表示如下：

```
while (隐藏位 == 0 && 阶码 > 0) {
    尾数左移，阶码减1; // 左规
}
while (尾数前27位不全为0 && 阶码 < 0) {
    尾数右移，阶码加1; // 右规
}

if (阶码上溢) {
    将结果置为无穷;
} else if (阶码下溢) {
    将结果置为0;
} else if(阶码 == 0) {
  尾数右移一次化为非规格化数;
} else {
    此时阶码正常，无需任何操作;
}
```

对于规格化后的舍入操作，我们不要求掌握GRS保护位相关的舍入操作，感兴趣的同学可以阅读2.5节内容。我们依然提供了舍入操作的函数，方法签名如下

```
private String round(char sign, String exp, String sig_grs)
```

请注意，在调用此方法前，请确保你传入的参数已经进行了规格化，务必确保传入的符号位为1位，阶码为8位。可以传入位数大于等于27位的尾数，round函数会先取出前27位作为1位隐藏位+23位有效位+3位GRS保护位，剩余的所有位数都将舍入到保护位的最后一位中。

在此方法中，我们已经对GRS保护位进行了相应的处理并完成舍入，返回的结果即为32位的字符串，转化为DataType类型后即可进行返回。

至此，你已经完成了浮点数乘法的全部工作(・ω・)ノ

2.5 对浮点数除法的相关说明

浮点数除法和乘法的主要区别在第三步：模拟运算得到中间结果上面。由于27位尾数进行无符号除法后，得到的商也是27位的，已经符合了“1位隐藏位+23位有效位+3位保护位”的要求，所以不再需要额外的操作。

同时，也正是因为这个27位尾数的除法，得到的27位商的精度将会严重损失（为什么？）。因此，我们无法对除法运算提供像加减乘一样如此精心打磨的test9，也无法提供RandomTest。我们本可以通过一些额外的操作来改进这一步运算的精度（比如将尾数扩展至更多位数，进行运算前将被除数尽可能左移，将除数尽可能右移等），但考虑这会大幅增加作业难度，我们很遗憾地放弃了这个改进。

因此，由于精度限制，本次作业中的除法的所有用例都是规格化数，大家无需考虑非规格化数的情况。此外，由于大规模用例的缺失，为了让大家也能够拥有足够的测试用例对除法进行debug，我们只会隐藏很简单的一些用例，其余用例全部提供给大家。

2.6 测试用例相关

本次实验中，test9方法会使用表驱动的方法进行多次的运算。如果出现了报错，但却不知道是哪一对数字报的错，可以在函数运行过程中，每当遇到expect结果跟actual结果不一样的情况时，将src、dest、expect与actual分别打印到控制台，然后再对这组数据进行单步调试。这种调试方法不但在本次作业中非常有用，并且也会让你在以后的debug生涯中受益匪浅。

例如，可以在test9中的`assertEquals`语句前插入如下代码。

```
String expect = Transformer.intToBinary(Integer.toString(Float.floatToIntBits(input[i] +
input[j])));
if (!expect.equals(result.toString())) {
    System.out.println("i = " + i + ", j = " + j);
    System.out.println("src: " + src);
    System.out.println("dest:" + dest);
    System.out.println("Expect: " + expect);
    System.out.println("Actual: " + result);
    System.out.println();
}
```