COA2021-programming05

1 实验要求

在 FPU 类中实现 2 个方法，具体如下

1.计算两个浮点数真值的积 dest × src

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

2.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 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

接下来以乘法为例，详细描述代码实现的相应步骤。

2.2.1 处理边界情况

在框架代码中，我们仍然提供了 cornerCheck 方法，使用正则表达式处理 NaN 的方法也同样适用，在此不再赘述。

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。

2.2.2 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。

注意，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。

同时需要特别注意，当提取出的阶码为全 0 且尾数不全为 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，以保证后面对阶码的计算不会出错。（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）

聪明的你是不是发现了，至此的所有操作都跟上次作业几乎一模一样。该怎么操作就不用我多说了吧。

2.2.3 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。

2.2.4 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。

本次作业对 round 函数做了优化，你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

2.3 对除法的相关说明

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。

2.4 如何进行调试

上次实验中，我们说过 test9 会进行多次的运算。如果出现了报错，但却不知道是哪一对数字报的错，可以在 fpu 类中编写 main 函数，将 test9 的代码复制到 main 函数中进行 debug。在 main 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，可以将 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。以下是一个 main 函数的例子，还不会如何操作的同学可以参考。

```
public static void main(String[] args) {
    FPU fpu = new FPU();
    Transformer transformer = new Transformer();
```

```
    DataType src;
    DataType dest;
    DataType result;

    String deNorm1 = "00000000000000000000000000000001";
    String deNorm2 = "00000000000000000000000000000010";
    String deNorm3 = "10000000100000000000000000000000";
    String small1 = "00000000100000000000000000000000";
    String small2 = "00000000100000000000000000000001";
    String big1 = "01111111000000000000000000000001";
    String big2 = "11111111000000000000000000000001";
    String[] strings = {deNorm1, deNorm2, deNorm3, small1, small2, big1, big2};
    double[] doubles = {10000000, 1.2, 1.1, 1, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, -0.1, -
0.2, -0.3, -0.4, -0.5, -0.6, -0.7, -0.8, -0.9, -1, -10000000};

    float[] input = new float[strings.length + doubles.length];
    for (int i = 0; i < strings.length; i++) {
    input[i] = Float.parseFloat(transformer.binaryToFloat(strings[i]));
    }
    for (int i = 0; i < doubles.length; i++) {
    input[i + strings.length] = (float) doubles[i];
    }

    for (int i = 0; i < input.length; i++) {
    for (int j = 0; j < input.length; j++) {
    src = new
DataType(transformer.intToBinary(Integer.toString(Float.floatToIntBits(input[i]))));
    dest = new
DataType(transformer.intToBinary(Integer.toString(Float.floatToIntBits(input[j]))));
    result = fpu.add(src, dest);
    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();
    }
    }
    }
    }
```

注意不要直接在 test 文件上进行修改，否则将代码 push 到 seecoder 平台上时可能会出错。

2.5 GRS 保护位

注：以下内容不需要掌握

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

$$1(0)\,.\,m1\ m2\ m3\ \ldots\ldots\ m22\ m23\ G\ R\ S$$

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。

最后，good luck and have fun~