# COA2021-programming04

| 版本号 | 修改内容 |
| --- | --- |
| v1.1 | 修改2.2.2，对2.2.1和2.4新增内容 |
| v1.2 | 2.2.2、2.2.4、2.3新增提示信息 |

# 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)
```

# 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

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seecoder平台上拿到15分。

### 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为“1”，~~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）

### 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）

~~提示一下，运算后尾数有进位变成了28位该如何处理？又或者尾数变成了小于27位的时候又该如何处理？~~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 测试用例相关

本次实验中，test9方法会进行多次的运算。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。

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。

## 2.4 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 …… 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与运算后的结果，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~