使用定点数运算实现单精度浮点数加法-IEEE754标准

学号: 171180526 姓名: 刘晗桐 专业: 电子信息科学与技术

> 内容目录

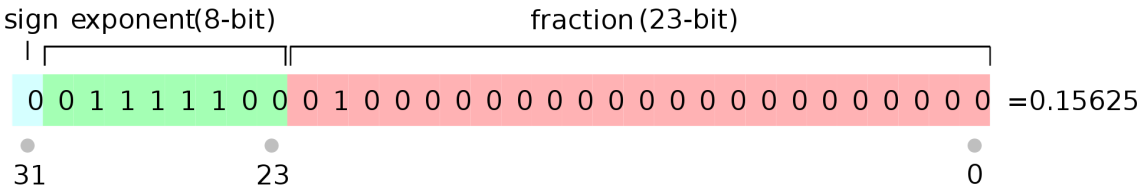
零、介绍

本文介绍了一种使用定点数运算（整数运算）实现计算单精度浮点数加法的一种方法，使用C语言实现于`Ubuntu 18.04`，由于穷举所有浮点数加法情况所需时间是个天文数字，只经过4794967295次运算测试，运算结果各位均与直接调用C语言浮点数加法操作符结果相同，可以保证算法的可靠性。

一、浮点数加法原理简释

单精度浮点数表示

IEEE 754标准制定的单精度浮点数的二进制表示示例如下（图片来源）：



一个单精度浮点数由符号位（sign）、阶码（exponent）与尾数（fraction）所共同组成，其中规格化数尾数部分最高位含有隐藏位1，由这种方法表示的浮点数共有五种表示与解读方法，一个简单的示例表如下：

名称	二进制表示	表示范围	示例
+0/-0	全0阶码全0尾数	+0/-0	0x00000000 | +0
非规格化数	全0阶码非0尾数	$(-1)^s \times 0.f \times 2^{-126}$	0x00400000 | $1 \times 0.5 \times 2^{-126}$
+∞/-∞	全1阶码全0尾数	+∞/-∞	0x7f800000 | +∞
NaN（Not a Number 非数）	全1阶码非0尾数	SNaN（发信号）/QNaN（静止的）	0x7fc00000 | QNaN
规格化非0数	阶码非全0且非全1	$(-1)^s \times 1.f \times 2^{e-127}$	如前

单精度浮点数加法

处理两个单精度浮点数加法时，需要进行以下几个步骤：

0. 处理特殊情况（corner cases）

开始进行加法时，我们拿到的可能不是通常意义上的小数，两个加数的值可能是Inf（无穷）、Zero（正0或负0）甚至NaN（非数），将这三种值统称为**特殊值**，对于这些特殊情况，IEEE754标准给出了相对应的计算标准，列表如下：

加数均为0	计算结果
两个数都为+0	+0
两个数都为-0	-0
其中一个为+0，另一个为-0	+0

加数均为Inf（无穷）	计算结果
正无穷+负无穷	负无穷
负无穷+正无穷	正无穷
正无穷+正无穷	正无穷
负无穷+负无穷	负无穷

其中一个为Inf（无穷），另一个为NaN（非数）	计算结果
同上	加数中的非数设置为QNaN的结果

加数均为NaN（非数）	计算结果
同上	置第二个加数为QNaN，并返回

其中一个加数为特殊值，另一个为非特殊值	计算结果
特殊值为0	返回非特殊值
特殊值为NaN	置NaN为QNaN，返回该QNaN
特殊值为Inf	返回该Inf

这些特殊情况在算法中均有体现。

1. 对阶

开始进行尾数相加时，首先调整较小的阶与较大的阶对齐（防止精度损失过大，不调整较大阶）

2. 尾数相加

为了保证计算精度精度，首先需将尾数左移保留附加位，（guard、round与sticky位），再进行相加，这三位分别称为保护位、舍入位和粘位，在后续的尾数舍入处理时进行舍入。由于在IEEE754标准中，尾数均为源码表示，所以进行尾数保留附加位后需根据加数的符号将尾数转化为补码表示后再进行相加。

3. 尾数规格化

保留附加位后的尾数长度为1（隐藏位）+23（尾数）+3（附加位） = 27位，相加后的尾数可能会发生较大变化，需要通过左规或者右规调整尾数长度，使之符合尾数长度要求，在左规和右规过程中要密切关注阶码的溢出问题，详细的规格化过程在后续的代码解读中进行阐释

4. 尾数的舍入处理

IEEE 754标准提供了多种附加位的舍入方法，在我进行的实验平台上，采用**就近舍入到偶数**的方法，舍入后，需要再次判断最高位是否有进位并进行再次规格化

二、代码解读

浮点数表示

为了方便将浮点数各部分转化为定点数（整数）进行运算，定义一个`union`类型保存浮点数, 定义于`float.h`中:

```
typedef union {
    struct
    {
        uint32_t fraction : 23;  //尾数
        uint32_t exponent : 8;   //阶数
        uint32_t sign : 1;       //符号位
    };
    float fval;                  //解释为浮点数
    uint32_t val;                //解释为无符号整型
} FLOAT;
```

由于我的电脑采用小端方式保存数据, 所以取的尾数, 阶数和符号位顺序如上所示, 采用`union`类型保存浮点数, 可以方便我直接操作浮点数的各部分并打印查看

特殊情况处理

定义浮点数加法函数`float_add`, 设置`FLOAT.val`为函数参数, 同时返回一个无符号整型作为计算结果, 定义与实现均在`float.c`中

```
uint32_t float_add(uint32_t a, uint32_t b)
```

根据前一部分原理阐释, 我把所有浮点数加法中的特殊情况分为四种, 分别对四种情况进行了处理:

情形1: 两数都为0或者两数都为无穷, 由于结果的确定性, 我将所有情况都定义在了一个结构类型`corner_add[]`中, 位于`float_add`函数的上方, 其中每种情况`CORNER_CASE_RULE`的三个参数表明了这种情况的加数以及结果:

```
// float.h
typedef struct
{
    uint32_t a;
    uint32_t b;
    uint32_t res;
} CORNER_CASE_RULE;

// float.c
CORNER_CASE_RULE corner_add[] = {
    {P_ZERO_F, P_ZERO_F, P_ZERO_F},
    {N_ZERO_F, P_ZERO_F, P_ZERO_F},
    {P_ZERO_F, N_ZERO_F, P_ZERO_F},
    {N_ZERO_F, N_ZERO_F, N_ZERO_F},

    {P_INF_F, N_INF_F, N_NAN_F},
    {N_INF_F, P_INF_F, N_NAN_F},
};
```

循环判断加数类型, 并将得到的结果返回, 若不属于这些类型, 则退出到下一个情形:

```
int i = 0;
for (; i < sizeof(corner_add) / sizeof(CORNER_CASE_RULE); i++)
{
    if (a == corner_add[i].a && b == corner_add[i].b)
        return corner_add[i].res;
}
```

情形2: 加数为NaN和Inf, 直接返回非数, 需要先将非数置为SNaN

```
if (isINF(a) && isNaN(b))
    return (b | (uint32_t)0x400000);

else if (isINF(b) && isNaN(a))
    return (a | (uint32_t)0x400000);
```

情形3: 其中一个数为0, 另一个为非零

```
if (a == P_ZERO_F || a == N_ZERO_F)
{
    if (isNaN(b))
        return (b | (uint32_t)0x400000);
}

if (b == P_ZERO_F || b == N_ZERO_F)
{
    if (isNaN(a))
        return (a | (uint32_t)0x400000);
}
```

需要注意的是, 如果另一个数为非数, 则需要将其处理为SNaN后再返回, 否则直接返回另一个数

情形4: 其中一个加数为无穷或者非数, 另一个非特殊值

```
if (fb.exponent == 0xff)
{
    if (isNaN(fb.val))
        fb.val|= 0x400000;
    return fb.val;
}

if (fa.exponent == 0xff)
{
    if (isNaN(fa.val))
        fa.val |= 0x400000;
    return fa.val;
}
```

需要返回这个无穷数或者非数

对阶

根据两个加数的大小关系进行对阶, 需要注意的是, 应首先将隐藏位以及附加位添加给尾数, 方便在改变阶数的过程中确保尾数变化的精度, 详细对阶过程解读见下方:

```
// 调整fa的阶数为较小阶
if (fa.exponent > fb.exponent)
{
    fa.val = b;
    fb.val = a;
}
// 添加隐藏位1
uint32_t sig_a, sig_b, sig_res;
sig_a = fa.fraction;
if (fa.exponent != 0)
    sig_a |= 0x800000; // the hidden 1
sig_b = fb.fraction;
if (fb.exponent != 0)
    sig_b |= 0x800000; // the hidden 1
// 根据阶码的规格化或非规格化计算阶码的差, 由于偏置数的存在, 只需要使用原码计算即可
// alignment shift for fa
uint32_t shift = 0;
shift = (fb.exponent == 0 ? fb.exponent + 1 : fb.exponent) - (fa.exponent == 0 ?
fa.exponent + 1 : fa.exponent);
```

```
// 添加几个附加位
sig_a = (sig_a << 3); // guard, round, sticky
sig_b = (sig_b << 3);
// 对阶，并注意移动尾数使之保持与阶码的对应性
uint32_t sticky = 0;
while (shift > 0)
{
    sticky = sticky | (sig_a & 0x1);
    sig_a = sig_a >> 1;
    sig_a |= sticky;
    shift--;
}
```

尾数相加

阶码对接完成后，需要注意根据浮点数的符号将尾数相加，负数应转化为补码进行计算，同时尾数相加可能会超出32位，所以采用64位无符号整型进行存储：

```
// 根据符号将尾数转化为补码表示
if (fa.sign)
{
    sig_a *= -1;
}
if (fb.sign)
{
    sig_b *= -1;
}
// 准化为补码后相加
sig_res = sig_a + sig_b;
// 根据尾数符号再次将尾数转化为原码
if (sign(sig_res))
{
    f.sign = 1;
    sig_res *= -1;
}
else
{
    f.sign = 0;
}
```

尾数相加结束后，再将结果的阶码设置为大的阶码

```
uint32_t exp_res = fb.exponent;
```

此时以基本完成了结果的给出，最后一步再将得到的结果规格化为32位表示即可

尾数规格化

在`float.c`中，我定义了一个规格化函数完成最后对尾数的规格化处理

```
uint32_t normalize(uint32_t sign, int32_t exp, uint64_t sig_grs)
```

该函数返回规格化完成的32位数，将该返回值解释为浮点数即完成了完整的浮点数加法过程

为了体现在规格化过程中对特殊情况的处理，我定义了一个bool类型对溢出进行判断，由于C语言缺少bool类型，我使用宏定义进行表示：

```
// float.h
typedef enum { false, true } bool;

// float.c
uint32_t normalize(uint32_t sign, int32_t exp, uint64_t sig_grs)
{
    bool overflow = false;
    .......
}
```

规格化分为左规和右规, 若尾数超出27位, 需要右规, 在右规过程中需要对阶码上溢进行处理, 若发生阶码上溢, 则需要置结果为正无穷

```
if ((sig_grs >> (23 + 3)) > 1 || exp < 0)
{
    // 右规并根据移出的位设置粘位(Sticky), 若粘位右边有1, 则置为1
    uint32_t sticky = sig_grs & 0x1;
    while (((sig_grs >> (23 + 3)) > 1) && exp < 0xff)
    {
        // shift right and pay attention to sticker bit
        sticky = sticky | (sig_grs & 0x1);
        sig_grs = sig_grs >> 1;
        sig_grs |= sticky;
        exp++;
    }
    //如果溢出了, 置结果为正无穷即可
    if (exp >= 0xff)
    {
        // assign the number to infinity
        exp = 0xff;
        sig_grs = 0;
        overflow = true;
    }
}
```

如果小于27位, 则需要对尾数进行左规, 需要注意的是, 如果阶码降为0, 则根据IEEE 754标准, 则需要右规一次以消除粘位, 达到`0.f`的效果

```
else if (((sig_grs >> (23 + 3)) == 0) && exp > 0)
{
    while (((sig_grs >> (23 + 3)) == 0) && exp > 0)
    {
        // 左规
        sig_grs = sig_grs << 1;
        exp--;
    }
    if (exp == 0)
    {
        // 右规一次
        if((sig_grs & 0x1) == 1)
        {
            sig_grs = sig_grs >> 1;
            // keep the sticker
            sig_grs = sig_grs | 0x1;
        }
        else if((sig_grs & 0x1) == 0)
```

```
        {
            sig_grs = sig_grs >> 1;
            sig_grs = sig_grs & 0xfffffffe;
        }
    }
}
```

最后一种情况是两个非规格化数相加后变为规格化数, 需要进行额外处理

```
else if (exp == 0 && sig_grs >> (23 + 3) == 1)
{
    // two denormals result in a normal
    exp++;
}
```

尾数舍入处理

规格化完成, 最后一步就是舍入三个附加位了(g, r, s), 如果结果没有溢出, 则进行舍入, 按照**就近舍入到偶数**的方式进行舍入, 需要注意的是, 如果舍入的过程中发生了尾数再次加一位, 则需要进行再次右规, 同时在右规的过程中要再次判断是否发生了溢出:

```
if (!overflow)
{
    // 根据GRS位的大小进行舍入
    uint32_t GRS = sig_grs & 0x7;
    uint32_t closeGRS = (sig_grs & 0x8) >> 3;

    if(GRS == 4 && closeGRS == 1)
        sig_grs += 8;
    else if (GRS == 4 && closeGRS == 0)
        sig_grs = sig_grs + 0;
    else if (GRS > 4)
        sig_grs += 8;
    else if (GRS < 4)
        sig_grs = sig_grs + 0;
    // 舍入后删掉GRS位
    sig_grs = sig_grs >> 3;
    //尾数加一位, 需要再次右规
    if((sig_grs >> (23)) > 1)
    {
        // shift right
        sig_grs = sig_grs >> 1;
        exp++;
        //如果溢出了, 置为溢出
        if((exp & 0xff) == 0xff)
            overflow = true;
    }

    // 最后一步把隐藏位去掉
    sig_grs = sig_grs & 0x7fffff;
}
```

完成上述所有步骤, 我们就得到了结果的尾数 | 阶码值, 再加上符号数值, 我们就可以构造结果并返回了

```
FLOAT f;
f.sign = sign;
f.exponent = (uint32_t)(exp & 0xff);
f.fraction = sig_grs;
return f.val;
```

以上所有的函数以及常量声明等都定义在`float.h`以及`float.c`中

三、测试原理及结果

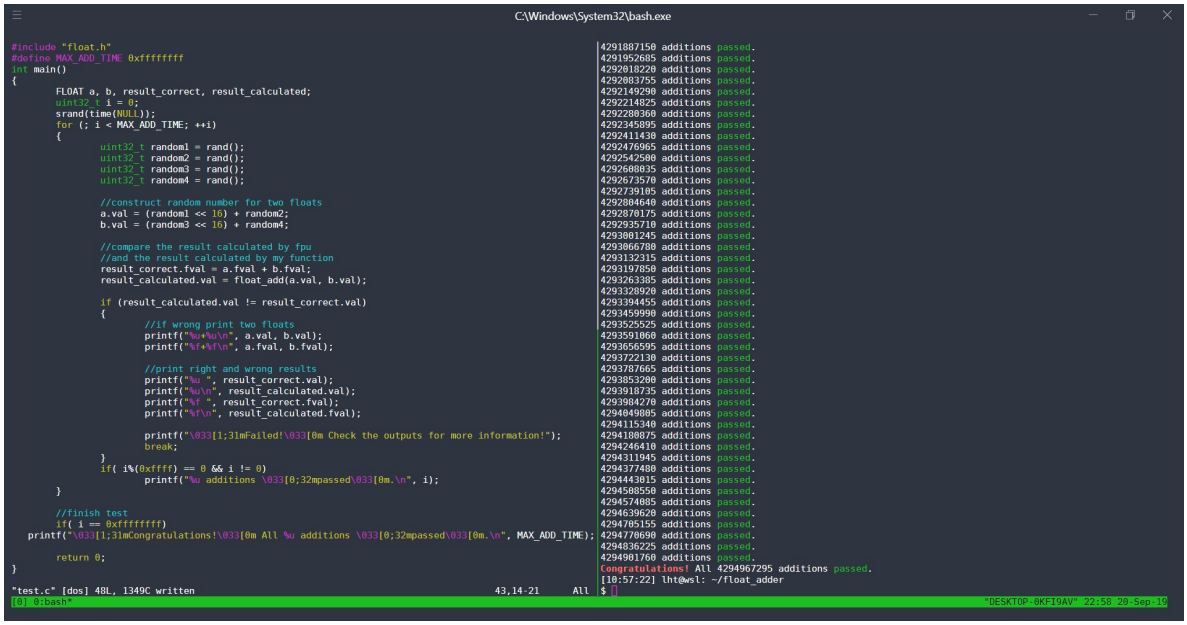
完成第二部分的所有步骤, 我们就得到了一个可以计算浮点数加法的定点数计算器, 它是一个函数:

```
uint32_t float_add(uint32_t a, uint32_t b)
```

我们需要构造尽可能多的浮点数对, 通过直接使用`float a + float b`与使用`float_add(float a, float b)`进行结果对比, 来验证我们的计算器的可靠性, 测试main函数定义在`test.c`中. 原理很简单, 通过不断构造随机的"浮点数", 并比较两种方式结果的全部32位来进行判断, 如果足够多的测试结果都证明是正确的, 那么就可以说明我们的计算器是可靠的, 在`test.c`中, 我一共比较了

$0xffffffff = 4294967295$次计算结果, 全部正确

`test.c`代码及其测试结果截图见下:



四、参考资料

- 袁春风, 余子濠. *《计算机系统基础》*第二版. 机械工业出版社. 2018年8月.
- 方元, 彭成磊. *《数字系统I补充教材》*南京大学电子科学与工程学院. 2019年夏.
- Bryant, Randal. O'Hallaron, David. *Computer Systems A Programmer's Perspective*. Third Edition, CMP, 2016.
- Prof. W. Kahan. *Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic*. Elect. Eng. & Computer Science, 1997.
- Hollasch, Steve. *IEEE Standard 754 Floating Point Numbers*, 08/24/2018.