计算机系统原理实验报告

> 浮点数的表示与算术运算算法
>
> 王子腾 3180102173 软件工程
>
> 2020.04.07

问题描述

浮点数的表示与算术运算算法分析，要求理论推导与程序模拟。 算术运算包括字符串到浮点数、浮点数到字符串的转换，加、减、乘、除四则运算等。要有说明文档，包括算法证明、程序框图、使用方法、特殊处理(溢出、数位扩展)、实例分析等等。字符串转换可能稍难。

```
typedef unsigned int dwrd; #32-bit
char* ftoa(dwrd);
dwrd atof(char*);
dwrd fadd(dwrd, dwrd);
dwrd fsub(dwrd, dwrd);
dwrd fmul(dwrd, dwrd);
dwrd fdiv(dwrd, dwrd);
```

1、主程序只供验证，你所须写的就是6个子程序。atof和ftoa可能会比较麻烦。

2、对结果应该进行分析，讨论你的浮点运算适用范围，可能的问题等等。

3、验证要有大数、小数，比如：123456789012345678901234567 89.123456789。浮点的范围是十的正负38次方。如果不能，分析原因。

4、可以增加十六进制的对比。比如显示 z1.f 与 (u.f+v.f) 的十六进制。这样可以分析四舍五入情况。

算法分析

字符串转换成浮点数

为了能够分别计算整数部分和小数部分，我们将整个字符串拆分成两部分：整数部分和小数部分。拆分后， 需要考虑一个特殊情况：即浮点数为 0 的情况，此时应直接返回。

- 整数部分的转化思路为除二取余，以此得到整数部分的二进制串。
- 小数部分的转换比较麻烦，我们以0.1234为例子，通过字符串切割得到小数部分1234，乘二取整的结果取决于是否前面多出了一位，比如70乘二时得到了140，此时需要验证，而40乘二得到的80 < 100不需要取证，因此设小数部分的字符串长度为 decimalsize，则乘二取整得到 1的条件为修改后decimal.size()>=decimalsize。

将整数和小数的字符串拼接后截取前23位作为尾数并舍入，再与符号位和阶码拼接即可。

```
//字符串转2进制32位浮点数
dwrd atof(string s)
{
    string res, mantissa;       //#32二进制
    string integer = "0", decimal = "0";    //十进制数
    dwrd pos = 0, i = 0;
    dwrd exp = 0,  ans= 0;

    if(s[0] == '-')     //符号位
    {
        res += '1';
        pos++;
    }
    else res +='0';

    res += "00000000";   //指数部分预填0
    i = pos;

    while(i!=s.size() && s[i]!='.') //分割整数和小数
        i++;
    if(i < s.size())
        decimal = s.substr(i+1);
    integer = s.substr(pos, i-pos);
    // 0
    if(!integer.compare("0") && !decimal.compare("0"))
        return 0;
    // 换算整数
    if(!integer.compare("0"))
        mantissa = "0";
    else
    {
```

```
        while(integer != "0")
        {
            mantissa += mod(integer, 2) ? '1': '0';
            integer = divide(integer, 2);
        }
        reverse(mantissa.begin(), mantissa.end());
    }
    exp = mantissa.size()-1+127;
    // 换算小数
    dwrd decimalsize = decimal.size();
    while(mantissa.size() < 40)
    {
        decimal = multi(decimal, 2);
        if(decimal.size() > decimalsize)
        {
            mantissa += '1';
            decimal.erase(decimal.begin());
        }
        else mantissa += '0';
    }
    while(mantissa[0] == '0')
    {
        mantissa.erase(mantissa.begin());
        exp--;
    }
    mantissa.erase(mantissa.begin());
    i = 22;
    if(mantissa[23] == '1')
    {
        while(mantissa[i] == '1')
        {
            mantissa[i] = '0';
            i--;
        }
        mantissa[i] = '1';
    }
    res += mantissa.substr(0, 23);
    for(i = 0; i < 32; i++) ans = ans*2 + (res[i]-'0');
    return ans + (exp << 23);
}
```

浮点数转换成字符串

浮点数转换成字符串较为繁琐，首先需要考虑特殊情况，如果是0则直接输出0，如果阶码是255则直接输出INF表示 无穷大。其次是考虑符号位，根据最高位判断出符号位后直接加到字符串中。

接来需要分情况讨论：

- 如果是一个大于1的数，则需要得到其整数部分和小数部分。以12.5（1100.1000<24*0>）为例, 整数部分较为容易，用高精度算法直接乘幂即可。而小数部分的处理比较麻烦。由于0.5 = 5/10，因此我们 可以通过“移位乘5”的方式得到每次运算的权重，如0.5 - 0.25 - 0.125 - 0.0625的权重变化可以看做是：5 - 25 - 125 - 0625这一变换得到的，因此我们每次构造一个以5的幂的权重进行计算即可。
- 如果是一个小于1的数，则不需要处理整数部分，小数部分的处理方法同上。

```
string ftoa(dwrd a)
{
    if(a == 0)  return "0";

    string buffer;
    if(a & N)   buffer += "-";
    dwrd exp = (a>>23) & 0xff;
    dwrd matissa = (a & 0x7fffff) | 0x800000;
    dwrd integer = 0, decimal = 0;
    if(exp == 255)  return "INF";
    if(exp >= 127)  // a>1
    {
        exp -= 127;
        string res = "0", power = "1";
        for(int i = exp+1; i > 0; i--)
        {
            if((matissa >> (24-i)) & 1) res = add(res, power);
            power = multi(power, 2);
        }
        buffer = buffer + res + '.';
        res = "0";
        power = "5";

        for(int k = exp + 2; k <= 24; k++)
        {
```

```
            if((matissa >> (24-k)) & 1)
            {
                string temp = power;
                reverse(temp.begin(), temp.end());
                res = add(res, temp);
            }
            power = "0" + power;
            power = multi(power, 5);
        }
        reverse(res.begin(), res.end());
        buffer += res;
    }
    else    // a<1
    {
        buffer += "0.";
        int len = 24;
        string res = "0", power = "5";
        for(dwrd i = 1; i < 127 - exp; i++)
        {
            power = "0" + power;
            power = multi(power, 5);
        }
        while((matissa & 1) == 0)
        {
            matissa = matissa >> 1;
            len--;
        }
        while(len > 0)
        {
            if((matissa >> (len - 1)) & 1)
            {
                string tem = power;
                reverse(tem.begin(), tem.end());
                res = add(res, tem);
            }
            power = "0" + power;
            power = multi(power, 5);
            len--;
        }
        reverse(res.begin(), res.end());
        buffer += res;
    }
    return buffer;
}
```

加减法

浮点数加法的原则为：“小阶向大阶看齐“，根据浮点数的格式取出 ea, eb 分别为浮点数 a 和 b 的阶码，而 ma , mb 为其尾数。计算时，将小阶对应的尾数右移后向大阶看齐，然后将尾数相减得到新的尾数。相减后进行重新规格化，但此时要注意如果两尾数相减后为0则直接返回0即可。

浮点数的减法可以通过加法来实现：a - b = a + (-b) ，因此只需改变 b 的符号位后再与 a 做加法即可。

```
dwrd fadd(dwrd a, dwrd b)
{
    if(a == 0)  return b;
    if(b == 0)  return a;
    if((a<<1) < (b<<1)) swap(a, b); //|a| >= |b|
    dwrd ea = (a>>23)&0xff;
    dwrd eb = (b>>23)&0xff;
    dwrd ma = (a & 0x7fffff) | 0x800000;
    dwrd mb = (b & 0x7fffff) | 0x800000;
    mb = mb >> (ea-eb);
    if((a^b)&0x80000000)
    {
        ma -= mb;
        if(ma == 0) return 0;
        while((ma&0x00800000) == 0)
        {
            ma <<= 1;
            ea--;
        }
    }
    else
    {
        ma += mb;
        if(ma == 0) return 0;
        while(ma & 0xff000000)
        {
            ma >>= 1;
```

```
            ea++;
        }
    }
    return (a & 0x80000000) | (ea << 23) |(ma & 0x7fffff);
}


dwrd fsub(dwrd a, dwrd b)
{
    return fadd(a, b^0x80000000);
}
```

乘法

- 根据异或值法则，浮点数乘法的符号位即为若两数最高位异或值：
- 乘法的阶码运算法则，设 E(a) 为阶码， a 为阶：

> E(a) = a + 127
>
> E(a*b) = (a + b) + 127 = E(a) + E(b) - 127

- 乘法的尾数即为补全 1 后两浮点数尾数值相乘，但由于一个浮点数的尾数（此处加上最前面的1）有24位，两个浮点数相乘时可达48位，导致发生溢出。所以，程序中只取两个尾数相乘后的前24位作为新的尾数而舍弃了后24位。

浮点数乘法后可能会产生类似加法一样的进位问题，所以在得到结果后也要进行规格化处理。

```
dwrd fmul(dwrd a, dwrd b)
{
    if(a == 0 || b ==0)  return 0;
    dwrd ea = (a>>23)&0xff;
    dwrd eb = (b>>23)&0xff;
    dwrd ma = (a & 0x7fffff) | 0x800000;
    dwrd mb = (b & 0x7fffff) | 0x800000;
    dwrd mr = 0;
    ea = ea + eb - 127;
    while(mb != 0)
    {
        mr = (mr>>1) + (mb&1) * ma;
        mb >>= 1;
    }
    while(mr & 0xfff000000)
    {
        mr>>=1;
        ea++;
    }
    return ((a^b) & 0x80000000) | (ea<<23) | (mr & 0x7fffff);
}
```

除法

- 根据异或值法则，浮点数除法的符号位即为若两数最高位异或值：
- 除法的阶码运算法则，设 E(a) 为 阶 码 ，a 为阶：

> E(a) = a + M E(a/b) = (a - b) + M = E(a) - E(b) + M

- 对尾数做除法时，为了处理最高位商0导致的移位失败，我们在开始时通过不断的移位处理使得 ma > mb ，这样保证了最高位能够商1，除此之外，该操作也能够保证除法后不需要再进行借位规格化处理。此外，该操作也能排除左移导致的进位规格化处理的问题，证明如下：设最后一次移位前 a / b = x ,由移位条件得 x ∈ [1/2, 2) ，当再进行移位后，可以得到 x1 ∈ [1/2, 2) ，因此商必介于1与2之间，不会产生进位。每次除法时，如果ma < mb，则说明不够减，此时商0，否则将 ma 减去 mb 并且商1，最后每次减法后将被除数左移，直到商的第24位为1为止。

为了便于对算法的理解我们以4 / 10为例进行演示：

> 4.0在浮点数中表示为：0 10000001 00000000000000000000000
>
> 10.0在浮点数中表示为：0 10000010 01000000000000000000000

阶码计算：

> 10000001 - 10000010 + 01111111 = 0111110 (126)

对尾数做除法：

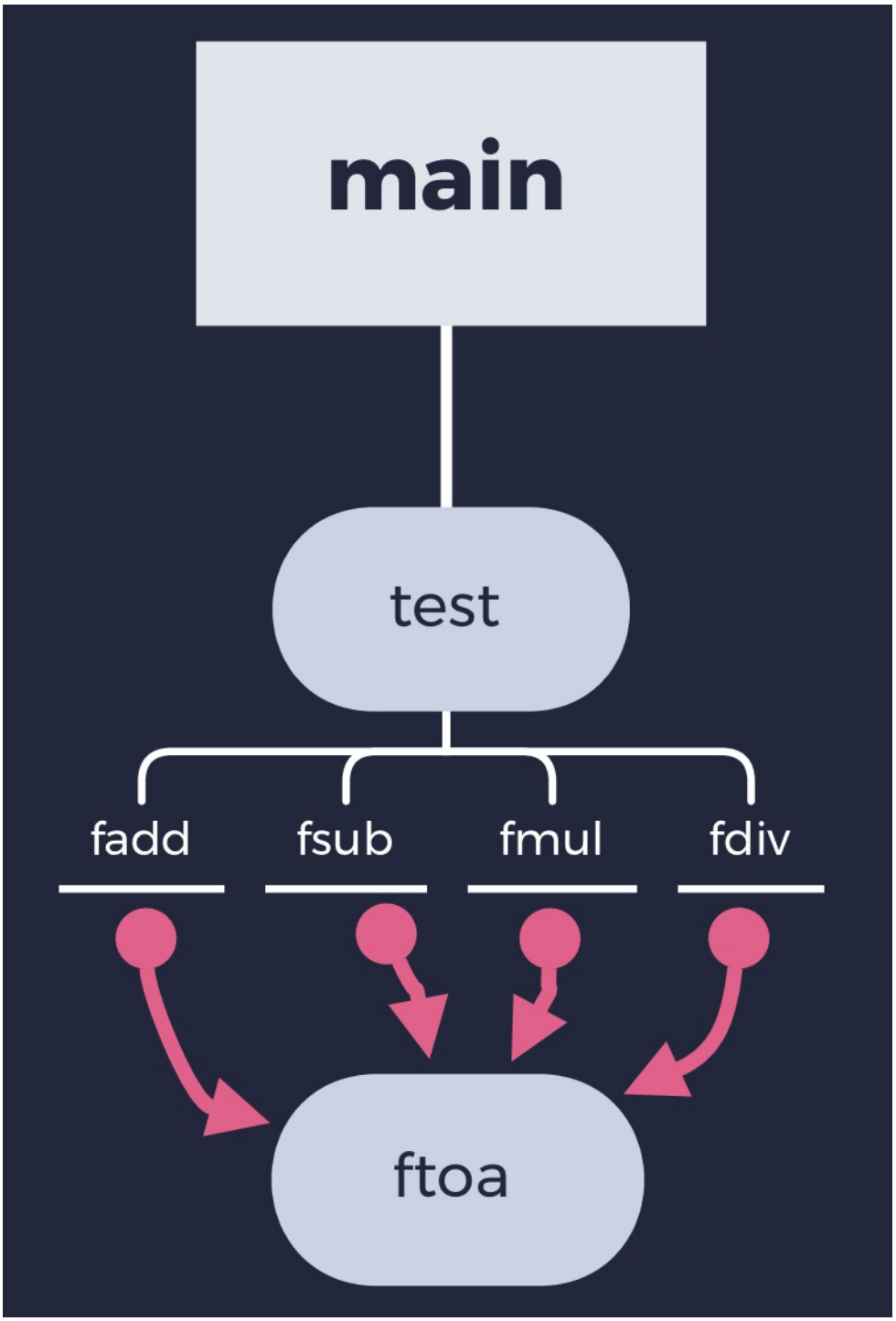
> 10000000000000000000000 / 10100000000000000000000

通过程序算得结果后进行移位操作以保持尾数性质。

```
dwrd fdiv(dwrd a, dwrd b)
```

```
{
    if(a == 0)  return 0;
    if(b == 0){
        cout << "ERROR" <<endl;
        return (a & 0x80000000) | 0x7f800000;
    }
    dwrd ea = (a>>23)&0xff;
    dwrd eb = (b>>23)&0xff;
    dwrd ma = (a & 0x7fffff) | 0x800000;
    dwrd mb = (b & 0x7fffff) | 0x800000;
    dwrd mr = 0;
    ea = ea - eb + 127;
    while(ma < mb)
    {
        ma<<=1;
        ea--;
    }
    while((mr & 0xff800000)==0)
    {
        if(ma < mb) mr <<= 1;
        else{
            ma -= mb;
            mr <<= 1;
            mr += 1;
        }
        ma <<= 1;
    }
    return ((a^b)&0x80000000) | (ea << 23) | (mr& 0x7fffff);
}
```

程序框图



特殊处理

通过辅助函数将可能存在溢出的计算转换为字符串的形式表示，由于mod函数计算不会发生溢出，可使用dwrd返回值

```
string divide(string s, dwrd b)
{
    if(s == "0")    return "0";
    dwrd r = 0, i = 0;
    string res;
    for(i = 0; i < s.size(); i++)
    {
        r = r*10 + s[i]-'0';
        if(r>=b)
        {
            res += r/b + '0';
            r %= b;
        }
        else res += '0';
    }
```

```
    while(res[0] == '0' && res!="0")
    {
        res.erase(res.begin());
    }
    return res;
}


dwrd mod(string s, dwrd b)
{
    if(s == "0")    return 0;
    dwrd r = 0, i = 0;
    string res;
    for(i = 0; i < s.size(); i++)
    {
        r = r*10 + s[i]-'0';
        if(r>=b)
        {
            res += r/b + '0';
            r %= b;
        }
        else res += '0';
    }
    while(res[0] == '0' && res!="0")
    {
        res.erase(res.begin());
    }
    return r;
}

string multi(string s, dwrd b)
{
    dwrd carry = 0;
    string res;
    for(int i = s.size()-1; i >= 0; i--)
    {
        res += ((s[i]-'0') * b + carry) % 10 + '0';
        carry = ((s[i]-'0') * b + carry) / 10;
    }
    while(carry != 0)
    {
        res += carry % 10 + '0';
        carry /= 10;
    }
    reverse(res.begin(), res.end());
    return res;
}

string add(string x, string y)
{
    string res;
    dwrd carry = 0;
    reverse(x.begin(), x.end());
    reverse(y.begin(), y.end());
    while(x.size() < y.size())
        x += '0';
    while(y.size() < x.size())
        y += '0';
    for(int i = 0; i < x.size(); i++)
    {
        res += (x[i] - '0' + y[i] - '0' + carry)%10 + '0';
        carry = (x[i] - '0' + y[i] - '0' + carry)/10;
    }
    if(carry != 0)
        res += carry + '0';
    reverse(res.begin(), res.end());
    return res;
}
```

测试实例分析

case1:可准确转换

```
Float 1: 2.5
Float 2: -2
Float1 is represented as :40200000
float2 is represented as :c0000000
2.5 + -2 = 0.5
2.5 - -2 = 4.5
```

```
2.5 * -2 = -5.0
2.5 / -2 = -1.25
```

case2:不可准确转换

```
Float 1: 2.3
Float 2: -1.4
Float1 is represented as :40133333
float2 is represented as :bfb33333
2.3 + -1.4 = 0.887898005369522840725
2.3 - -1.4 = 3.68799771916733682875
2.3 * -1.4 = -3.1198877788193840501725
2.3 / -1.4 = -1.631856073739639928125
```

case3:大数运算（不溢出）

```
Float 1: 1234567890
Float 2: 9876543210
Float1 is represented as :4e932c06
float2 is represented as :50132c06
1234567890 + 9876543210 = 11111110658.0
1234567890 - 9876543210 = -8641976322.0
1234567890 * 9876543210 = 12193262897479494656.0
1234567890 / 9876543210 = 0.125
```

case4:大数运算（溢出）

```
Float 1: 123456789012345678901234567890.123456789
Float 2: -3.222222222222
Float1 is represented as :6e1f906d
float2 is represented as :c04e38e4
123456789012345678901234567890.123456789 + -3.222222222222 = 123456791572657037544618791040.0
123456789012345678901234567890.123456789 - -3.222222222222 = 123456791572657037544618791040.0
123456789012345678901234567890.123456789 * -3.222222222222 = -397805210730830337686609379840.0
123456789012345678901234567890.123456789 / -3.222222222222 = -383141737434834764124345297 6.0
```

分析： 由于采用了高精度算法，多余或不足位数的舍入，因此在 10 的正负 38 次方内均可以使用。但是由于atof函数在转换输出时存在一定的误差，因此针对不能完全转换成二进制浮点数的数的输出会存在精度丢失的问题。