Homework 2

BY 陈牧歌
1500012702

1 习题2.86

1.1 题意简述

Intel兼容cpu支持一种叫做『 extended precision』的浮点格式，这种格式用80个二进制bit来表示一个浮点数，其中包含一个符号位，$k = 15$个指数位，以及一个**整数位**(integer bit)，和$n = 63$个小数位(fraction bits)。

整数位可以看成我们讲IEEE浮点标准中省略的那一位的明确表示，即它对正规化的浮点数为1，非正规的为0。

用extended precision格式填写下表：

Description	Value	Decimal
Smallest positive denormalized		
Smallest positive normalized		
Largest normalized		

1.2 解答

1.2.1 Smallest positive denormalized

显然前17位均为0, fraction bits仅最后一位为1。

那么二进制表示为

$$\underbrace{00...00}_{79个0}1$$

精确十进制值为

$$2^{-16382-63} \approx 3.64519953188247460252840593361941981639905081569356334372098 \times 10^{-4951}$$

$C/C++$于ICS9服务器上运算结果为

$$3.6451995319 \times 10^{-4951}$$

1.2.2 Smallest positive normalized

显然前16位仅最后一位为1, integer bit也应该为1, 其它应该均为0。

那么二进制表示为

$$\underbrace{00...00}_{15个0}1\underbrace{00...00}_{63个0}$$

精确十进制为

$$2^{1-16383} \approx 3.362103143112093506262677817321752602598079344846 5^{-4932}$$

$C/C++$于ICS9服务器上运算结果为

$$3.3621031431 \times 10^{-4932}$$

1.2.3 Largest normalized

指数位应为$2^{15}-2$, frac部分应该全为1。

于是二进制表示为

$$0\underbrace{11\ldots1}_{14个1}0\underbrace{11\ldots11}_{64个1}$$

精确十进制为

$$(2^{65}-1)\times 2^{2^{15}-2-16383-64}\approx 1.18973149535723176505351\times 10^{4932}$$

$C/C++$于ICS9服务器上运算结果为

$$1.1897314954\times 10^{4932}$$

1.3 解题相关代码

```
union {
    short a[5];
    long double b;
} tmp;
int main() {
    tmp.a[0] = 1;
    printf("Smallest positive denormalized %.10Le\n", tmp.b);
    tmp.a[0] = 0;
    tmp.a[4] = 1;
    tmp.a[3] = 1 << 15;
    printf("Smallest positive normalized %.10Le\n", tmp.b);
    memset(tmp.a, -1, sizeof tmp.a);
    tmp.a[4] ^= 1 << 15;
    tmp.a[4] ^= 1;
    printf("Largest normalized %.10Le\n", tmp.b);
}
```

2 习题2.95

2.1 题意简述

实现函数`float_bits float_half(float_bits f);`

要求严格按照书上的位运算编程规则。

2.2 解答

本题实际上也是我们的datalab中的一个subtask。

一个朴素的想法是, 取出exp位, 如果全为1, 那么原样返回(∞或者NaN)。

如果大于1, 那么exp减1返回。

如果exp恰好等于1, 那么就要完成normalized到denormalized的转换。

如果exp等于0, 那么就是frac除2后取整。

值得注意的是, 取整需要按round to even规则。

按照这个朴素思路, 容易写出代码：

```
float_bits float_half(float_bits uf){
    int tmp = uf & 0X7f800000;  //取exp
    int tmp2 = uf & (0x7fffff); //取frac
    if (tmp ^ 0X7f800000) {     //判断是否为NaN/Inf
        if (tmp > 0x800000) {   //exp大于1
            return uf - 0x800000;
        } else if (tmp == 0) {  //Denorm
            uf = uf ^ tmp2 ^ (tmp2 >> 1);
        } else {                //Norm -> Denorm
            uf = (uf ^ tmp2 ^ (tmp2 >> 1)) - 0x400000;
        }
        if ((tmp2 & 3) == 3) uf++; //round2even
    }
    return uf;

}
```

这样需要15个操作, 完成了float_half。

利用一些trick, 我们可以优化到7个操作。

(driver.pl对switch的次数计算并不靠谱)

```
//代码来自郭天魁,在driver.pl上测试结果ops为7
unsigned float_half(unsigned uf) {
    unsigned sign_exp = uf & 0xFF800000;
    unsigned frac;
    unsigned mask = 0x7FFFFF;
    unsigned shift = 1;
    switch (sign_exp) {
        case 0x7F800000:
        case 0xFF800000:
            return uf;
        case 0:
        case 0x80000000:
            break;
        default:
            switch (sign_exp -= 0x800000) {
                case 0:
                case 0x80000000:
                    mask = 0xFFFFFF;
                    break;
                default:
                    shift = 0;
            }
    }
    frac = uf & mask;
    frac = frac >> shift;
    switch (uf & 3) {
        case 3:
            frac += shift;
    }
    return sign_exp | frac;
}
```

3 习题2.97

3.1 题意简述

按照位运算代码规则，实现`float_bits float_i2f(int i);`

即int转到float。

3.2 解答

同样也是lab代码，一个很朴素的想法即先判断正负，因为int如果非0的话，必定是normalized的，所以找出最大的k，满足$2^k \leqslant i$，然后将i减去2^k，再稍微做做对应的shift，就完成了。

写出代码即为

```
float_bits float_i2f(int x) {
    int sig = x & 0x80000000; //判断符号位
    int cnt = 31;             //用于找出被减去的2的幂次
    int tmp;
    if (sig) x = -x;          //负的取反
    if (x == 0) return x;
    if (x < 0) x = -x;        //这时候还为-的说明x为-2^31
    while (!(x & (1 << cnt))) cnt--;//暴力查找
    x -= 1 << cnt;
    if (cnt > 23) {
        //需要右移舍入
        int rig = cnt - 24;
        tmp = x >> (rig + 1);
        if (((x >> rig) & 1)) {
            int mask = (1 << rig) - 1;
            if ((x & mask) || tmp & 1) tmp++;
            //判断什么时候round2even生效
        }
    } else {
        int lef = 23 - cnt;
        tmp = x << lef;
    }
    cnt += 0x7F;
    //加上bias
    return sig + (cnt << 23) + tmp;
}
```

这样操作数比较多，为29次。

同样，利用一些trick可以把它优化到7次。

但是因为我能找到的实现都太丑陋，可读性太差，所以就不贴了。