DATALAB 实验报告

黄泽文 2022013014 未央-软件21

Points	Rating	Errors	Points	Ops	Puzzle
1	1	0	2	4	bitAnd
2	2	0	2	3	getByte
3	3	0	2	6	logicalShift
4	4	0	2	37	bitCount
4	4	0	2	6	bang
1	1	0	2	1	tmin
2	2	0	2	7	fitsBits
2	2	0	2	12	divpwr2
2	2	0	2	2	negate
3	3	0	2	4	isPositive
3	3	0	2	15	isLessOrEqual
4	4	0	2	27	ilog2
2	2	0	2	8	float_neg
4	4	0	2	27	float_i2f
4	4	0	2	10	float_twice
41	41	0	30	169	TOT

1. Bit Manipulations

1.1 bitAnd

考察两个数的任意一位a_i, b_i，$a_i \& b_i = 1$当且仅当$a_i = b_i = 1$，不难发现这与$\sim (\sim a_i | \sim b_i)$等价。

```
int bitAnd(int x, int y)
{
   return ~(~x | ~y);
}
```

1.2 getByte

获取一个Byte，相当于获取8个位的值。将x右移$8 \times n$位后，最低8位即为所需的结果。

```
int getByte(int x, int n)
{
   int mask = 0xff;
   int shift = n << 3;
   return (x >> shift) & mask;
}
```

1.3 logicalShift

对于逻辑右移而言，最终结果的高n位一定为0，使用位运算将算术右移的结果高位置0即可。

```
int logicalShift(int x, int n)
{
   int mask = ~(((1 << 31) >> n) << 1); // make use of m-rightShift
   return (x >> n) & mask;
}
```

1.4 bitCount*

Hamming Weight问题。为了满足操作数的限制，这里使用swar算法。Swar算法使用了二分的思想，即，将32个数码作为一棵完全二叉树的所有叶子节点，从叶子向根一层层更新，将每个父节点的值置为其所有子节点的值的和。

具体实现上，可以通过掩码$mask = \{01010101..., 00110011..., 00001111...\}$实现。每层的更新使用$x = (x\&mask_i) + ((x >> 2^i)\&mask_i)$，这相当于通过掩码将两个子节点的值取出并相加（左侧的节点移动到右侧），并将每个父节点的值原位储存。

```
int bitCount(int x)
{
   // swar
   int mask1 = 0x55, mask2 = 0x33, mask3 = 0x0f;
   mask1 = mask1 + (mask1 << 8) + (mask1 << 16) + (mask1 << 24);
   mask2 = mask2 + (mask2 << 8) + (mask2 << 16) + (mask2 << 24);
   mask3 = mask3 + (mask3 << 8) + (mask3 << 16) + (mask3 << 24);
   x = (x & mask1) + ((x >> 1) & mask1);
   x = (x & mask2) + ((x >> 2) & mask2);
   x = (x & mask3) + ((x >> 4) & mask3);
   x = x + (x >> 8) + (x >> 16) + (x >> 24); // combine the rest value
   return x & 0xff;
}
```

1.5 bang

对于非0数而言，其自身与其相反数的最高位必然存在一个1，换句话说，$x == 0$当且仅当x与-x最高位都是0。

```
int bang(int x)
{
   // if x == 0, the highest digit of both x and -x is 0,
   // otherwise, 1 and 0.
   return 1 ^ (((x | (~x + 1)) >> 31) & 1);
}
```

2. Two's Complement Arithmetic

2.1 tmin

Two's中，最小的负数除了符号位外都为0。

```
int tmin(void)
{
   return 1 << 31;
}
```

2.2 fitsBits

只保留x的低n位，判断x的值是否发生了变化。这里使用位运算实现目标。

```
int fitsBits(int x, int n)
{
   int shift = 32 + ~n + 1; // = 32 - n
   int y = x << shift;
   y = y >> shift;
   return !(x ^ y);
}
```

2.3 divpwr2

对于正数而言，直接右移n位即可。对于负数而言，直接右移会导致取整的方向错误，需先加上$2^n - 1$的偏置值。利用这个偏置值，可以将精确计算结果的范围从$[ans - 1, ans)$改变到$[ans, ans + 1)$，确保取整正确。

```
int divpwr2(int x, int n)
{
   int mask = x >> 31;
   // when x is neg, add a bias = (1 << n) - 1
   return (mask & (x + (1 << n) + ~1 + 1) >> n) | (~mask & (x >> n));
}
```

2.4 negate

负数采用补码表示，结果等于其绝对值按位取反再加1. 验证可以发现，将补码转换为原码的操作完全相同。

```
int negate(int x)
{
   return ~x + 1;
}
```

2.5 isPositive

对于正数和负数而言，判断其符号位即可。需要注意的是，0在这种判断下会被视为正数，但应该返回0，需要特殊处理。

```c
int isPositive(int x)
{
   return !((x >> 31) | !x); // |!x check 0
}
```

2.6 isLessOrEqual

朴素的想法是计算$y-x$，判断其是否非负即可。但如果x与y异号，减法可能溢出，导致结果错误，需要对这种情况额外分析。

```c
int isLessOrEqual(int x, int y)
{
   // when x-y+, return 1; x+y-, return 0; otherwise isPositiveOrEqualzero(y-x)
   int xt = (x >> 31) & 1;
   int yt = (y >> 31) & 1;
   return ((xt ^ yt) & xt) | (!(xt ^ yt) & !((y + ~x + 1) >> 31));
}
```

2.7 ilog2*

最高的1所在的位置即为结果。为了满足操作数的限制，利用二分查找的思想做优化。

1. 对于32个数位而言，判断高16位中是否存在1。
2. 如果不存在1，则最终结果小于16，这个数可以直接被视为16个数位组成的数。
3. 如果存在1，则最终结果大于等于16，后续分析只需保留高16位。最后结果是这个16位数的log值加上16。
4. 重复1-3步，对于2、3步给出的16个数位而言，判断其高8位是否存在1. 不断重复这个操作即可。

```c
 int ilog2(int x)
 {
    // find the pos of highest 1
    // use binary search to opti code

    // check ilog2(x) >= 16 or not
    int check = !!(x >> 16);
    int mask = check << 4;
    int ans = mask;
    // if so, x >> 16. we only consider the highest 16 digits.
    // if x is not changed, then ans < 16, in other words we still only need consider the lowest 16 digits.
    x = x >> mask;

    // check ilog2(new_x) >= 8 or not
    check = !!(x >> 8);
    mask = check << 3;
    ans = ans + mask;
    x = x >> mask;

    check = !!(x >> 4);
    mask = check << 2;
    ans = ans + mask;
    x = x >> mask;

    check = !!(x >> 2);
    mask = check << 1;
    ans = ans + mask;
```

```
    x = x >> mask;

    check = !!(x >> 1);
    mask = check;
    ans = ans + mask;
    return ans;
}
```

3. Floating-Point Operations

3.1 float_neg

最高位取反即可，注意特殊判断NaN的情况。

```
unsigned float_neg(unsigned uf)
{
    int mask = ~0x7f800000; // 1 00000000 111...
    if (!(~(uf | mask)) && uf << 9)
    {
        return uf;
    }
    else
    {
        return uf ^ (1 << 31);
    }
}
```

3.2 float_i2f*

先对x做特殊判断和预处理：x为0时结果为0，x为负数时需先取绝对值得到原码。

核心的计算过程为：

1. 为了使用科学计数法，先寻找x最高的1所在的位置，记为topPos。
2. 左移x，将topPos移到最高位。现在可以将x分为3部分：最高位为原数中最高的1，这个1在float表示中应该被省略；中间23位是需要存入float的frac值，但还未考虑取整；后8位是无法表示的数位，如果其非0，则应该判断其是否会导致进位。
3. 如果原先的topPos>23，则说明后8位可能是非零的，需要进行取整分析。如果超过一半，直接向上取整；如果不低于一半且中间23位的最低位为1，也向上取整。
4. 向上取整的操作是：先对frac的结果+1，再判断最高位是否发生了进位，即，结果是否超过23个数位。如果超过，需要重新计算科学计数法，对exp+1，将frac右移一位（需要注意的是，右移后的最高位一定是1，但这个1会与float表示中被省略的1相加，再次进位，所以应该被抹去）。
5. 合并sign，exp和frac即可。

从计算的角度讲，这里的第2步左移不是必要的。但它可以通过固定topPos的位置，简化后续计算的表达式，从而减少操作数。

```
unsigned float_i2f(int x)
{
    int neg = x & 0x80000000, topPos = 31, frac;
    if (x == 0)
        return 0;
    // if neg, change to pos and mark
    if (neg)
    {
```

```
        x = -x;
    }
    // get the highest 1's pos
    while (!(x >> topPos))
        topPos = topPos - 1;
    // printf("%d %d\n", x, topPos);
    // here standardize the x to decrease ops
    x = x << (31 - topPos);
    frac = 0x007fffff & (x >> 8);
    if (topPos > 23)
    {
        // round
        if ((x & 255) > 128 || ((x >> 7) & 3) == 3) // > half or = half to even
        {
            // printf("%d round!\n", x);
            frac = frac + 1;
            if ((frac >> 23) & 1)
            {
                frac = (frac >> 1) & 0x003fffff;
                topPos = topPos + 1;
            }
        }
    }
    return frac | ((topPos + 127) << 23) | neg;
}
```

3.3 float_twice

对于normalized，exp位加一即可。对于denormalized，需分类讨论0和极小数、NaN和inf。

```
unsigned float_twice(unsigned uf)
{
    // check 0
    if (!(uf << 1))
    {
        // printf("%u zero\n", uf);
        return uf;
    }
    // check nan or inf
    if ((uf & 0x7f800000) == 0x7f800000)
    {
        // printf("%u nan of inf\n", uf);
        return uf;
    }
    // denormalized small
    if (!(uf & 0x7f800000))
    {
        // printf("%u denorm\n", uf);
        return (uf << 1) | (uf & 0x80000000);
    }
    // normalized
    return uf + 0x00800000;
}
```