计算机组成原理 DataLab Report

徐浩博　软件02　2020010108

1. Bit Manipulations

1.1 bitAnd(x, y)

由逻辑运算的摩根定律，$x \wedge y = \neg(\neg x \vee \neg y)$，转换为数字运算即$x\&y = ((x)|(y))$

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

1.2 getByte(x, n)

首先n是位数，我们通过乘3（右移3位）获得比特数(int shift)，然后将0xff左移shift位与x做AND操作获得mask的所求比特，然后再向右移位获得最终结果。注意最后移位完成后仍需AND 0xff以避免负数算术右移使得高位补1的情况。

```c
int getByte(int x, int n) {
    int shift = n << 3;
    int result_with_shift = x & (0xff << shift);
    // get Byte with shift
    return (result_with_shift >> shift) & 0xff;
    // AND 0xff to prevent arithmetic right shift of negative number
}
```

1.3 logicalShift(x, n)

首先对于正数，逻辑右移和算术右移等价；对于负数，逻辑右移n位实际上将

$$x = -TMAX + \alpha_1 TMAX/2 + \alpha_2 TMAX/4 + \cdots$$

移动成了

$$x_{logical} = (TMAX/2^n) + \alpha_1 TMAX/2^{n+1} + \alpha_2 TMAX/2^{n+2} + \cdots$$

其中$\alpha_i = 0\ or\ 1$，但算术右移实际上是

$$x_{arithmetic} = (-TMAX + TMAX/2 + \cdots + TMAX/2^n) + \alpha_1 TMAX/2^{n+1} + \alpha_2 TMAX/2^{n+2} + \cdots$$

显然给算术右移结果加$TMAX/2^{n+1}$即为逻辑右移结果，而$TMAX/2^{n+1} = 2^{32-n} = 2^{32+(\sim n)+1} = 2^{33+(\sim n)}$，因此我们只需要在判别$n! = 0\&\&x < 0$的基础上加$q = 1 << (33 + (\sim n))$即可.

```c
int logicalShift(int x, int n) {
    int n_not_zero = (n & 1) | ((n >> 1) & 1) | ((n >> 2) & 1) | ((n >> 3) & 1) |
        ((n >> 4) & 1);
```

```
    // = 1, if n != 0; = 0, else
    int n_not_zero_and_x_negative = n_not_zero & (x >> 31);
    // = 1, if n != 0 && x < 0; = 0, else
    int q = n_not_zero_and_x_negative << (33 + (~n));
    // if n != 0 && x < 0, then add 2^(32 - n); else add 0
    return (x >> n) + q;
  }
```

1.4 bitCount(x)

本题的基本思路是先两两计算bitcount，以二进制的方法记录到原位上，然后合并到四位，再合并到八位，以此类推.

为了避免负数的算术右移，我们先提取符号位，并且给x AND 0x7fffffff以保证bitCount计数时是对正数运算，最后答案在加上符号位是否为1即可.

其次有如下观察：对于2位的二进制数，计算bitcount可以用以下方式：

$$Bitcount(a) = BitCount((a_1a_2)_2) = (a_1a_2)_2 - (a_1)_2 = a - (a >> 1)$$

推广到32位int，两两计算BitCount，只需要借助一个$mask = (0101...01)_2 = 0x55555555$，由$x - (x\&mask)$得到. 这样每两位二进制数的1的个数就以二进制数的方式记录到了原位.

再次，我们要将2组2位数的计算拼成1组4位数，这是简单的，我们只需要借助一个$mask = (0011...0011)_2 = 0x33333333$，由$(x\&mask) + ((x >> 2)\&mask)$得到.

然后，我们要将2组4位数的计算拼成1组8位数，借助$mask = (00001111...00001111)_2 = 0x0f0f0f0f$，由$(x\&mask) + ((x >> 4)\&mask)$得到，为了节省运算符，我们可以将其写做$(x + (x >> 4))\&mask$，能够这么运算的原因是8位数最多8个1,1的个数可以用4个二进制数表示，故最终结果只有后四位有效，并不会溢出到高四位上，也因此，可以在加法之后统一AND mask.

之后，将2组8进制数的计算结果拼成1组16进制数，这一步可以不需要mask的帮助，原因是如上所述，一组8位数的高4位为0。因此直接可以由$(x + (x >> 8))$得到。

最后2组16进制数拼成1个32进制数，加上第二段所叙述的符号位即为最终结果。

```
int bitCount(int x) {
    int sign, n_0x7fffffff, n_0x55, n_0x33, n_0x0f;

    // process signal of x
    sign = (1 & (x >> 31));
    n_0x7fffffff = (1 << 31) + (~ 1) +1;
    x = x & n_0x7fffffff;

    // 32 group * 1 bit/group -> 16 group * 2 bit/group
    n_0x55 = 85 + (85 << 16); // 85 = 0x55
    n_0x55 = n_0x55 + (n_0x55 << 8); // mask = 0x55555555
```

```
    x = x + (~((x >> 1) & n_0x55)) + 1; // x = x - (x & mask)

    // 16 group * 2 bit/group -> 8 group * 4 bit/group
    n_0x33 = 51 + (51 << 16); // 51 = 0x33
    n_0x33 = n_0x33 + (n_0x33 << 8); // mask = 0x33333333
    x = (x & n_0x33) + ((x >> 2) & n_0x33); // x = (x & mask)+((x >> 4) & mask)

    // 8 group * 4 bit/group -> 4 group * 8 bit/group
    n_0x0f = 15 + (15 << 16); // 51 = 0x0f
    n_0x0f = n_0x0f + (n_0x0f << 8); // mask = 0x0f0f0f0f
    x = (x + (x >> 4)) & n_0x0f; // x = (x + (x >> 8)) & mask

    // 4 group * 8 bit/group -> 2 group * 16 bit/group
    x = x + ((x >> 8));// 16

    // 2 group * 16 bit/group -> result
    return (x & 0xff) + ((x >> 16) & 0xff) + sign;
}
```

1.5 bang(x)

此题与上题类似，可以采用二分法. 将32bit通过高16bit OR 低16bit操作保留16bit，但并不丢失原32bit是否有1这一信息（若高16bit与低16bit对应位有一个1，OR后的数也保留有1）. 以此类推，16bit通过高8bit OR 低8bit保留8bit，直到最终得到1bit，该bit=1则表示原数有1，否则无1. 在这一过程中，需要注意OR采用x — (x >>n)的方式，但这一方式仍然在高位保留了原来的数，因此最终需要给x & 1，得到的就是最终1bit，高位确保全为0. 我们最终需要判断!x，当且仅当无1时!x=1，否则!x=0，因此，只需要(x & 1) ˆ 1就能得到最终结果.

```
int bang(int x) {
    x = x | (x >> 16); // 32 -> 16
    x = x | (x >> 8); // 16 -> 8
    x = x | (x >> 4); // 8 -> 4
    x = x | (x >> 2); // 4 -> 2
    x = x | (x >> 1); // 2 -> 1
    return (x & 1) ^ 1;
}
```

2. Two’s Complement Arithmetic

2.1 tmin()

Tmin即最高位为1，其余为全为0，因此可以通过(1<<31)得到.

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

2.2 fitsBits(x,n)

对于n bits，我们能够表示数的范围是$-2^{n-1} \sim 2^{n-1}-1$，由于正负数不一致，因此我们需要分情况讨论.

对于非负数，我们只需要将其右移(n-1)位，若其不为0，则不能被表示，反之为能表示.

对于负数，我们知道$|x| = -x = (\sim x)+1$，因此$\sim x = |x|-1$. 我们再由n bits能表示的范围$-2^{n-1} \sim 2^{n-1}-1$可知，对于负数x，x能用n位表示当且仅当—x—-1能够被表示，也即$\sim x$能被表示. 因此我们将x按位取反然后按照非负数的操作，右移(n-1)为看其是否为0.

结合以上两点，我们可以先计算出x的符号sign，然后将分情况讨论写成逻辑运算符，负数的时候sign为真，AND上对负数的判断；非负数!sign为真，AND上对非负数的判断；最后将两种结果OR起来即可.

```
int fitsBits(int x, int n) {
  int sign, positive_fitBits, negative_fitBits;
  sign = (1 & (x >> 31));
  // sign = 1, if x is negative; sign = 0, else

  n = n + (~1) + 1;
  // n = n - 1

  positive_fitBits = !((x >> n)) & (!sign);
  // positive_fitBits = 1 iff. x >= 0 and x satisfies fitsBits
  negative_fitBits = (!((~x) >> n)) & sign;
  // negative_fitBits = 1 iff. x >= 0 and x satisfies fitsBits
  return positive_fitBits | negative_fitBits;
  }
```

2.3 divpwr2(x, n)

对于正数，我们直接右移n位即可；而对于负数，算术右移导致向下舍入，并不是我们期望的向上舍入，因此我们需要加一个偏置2^n-1后右移. 下面我们说明这种偏置是正确的.

1）假设x能够被2ˆn整除，那么$x/2^n = \lfloor x/2^n \rfloor = \lfloor (x+2^n-1)/2^n \rfloor$，因此偏置并不会改变最终右移n位的结果. 在这里注意n=0时实际上偏置也等于0，因此也成立.

2）假设x不能被2ˆn整除，我们所求的结果$\lceil x/2^n \rceil = \lfloor x/2^n \rfloor + 1$，而不能整除意味着它的余数必然大于1，加上$2^n-1$会导致除以$2^n$的时候整数部分变大1，因此$\lfloor (x+2^n-1)/2^n \rfloor = \lfloor x/2^n \rfloor + 1$，从而我们所求的结果也可以用$\lfloor (x+2^n-1)/2^n \rfloor$表示. 综合以上两点，我们就证明了这种做法的正确性.

回到运算当中，我们先用sign记录x的正负（负为1），然后将sign右移n位减sign计算出add偏置量：当x非负时sign等于0，实际上偏置量等于0，而x为负数时偏置量为$2^n - 1$. 最后加上偏置量右移n位即可.

```
int divpwr2(int x, int n) {
    int sign = (x >> 31) & 1;
    // sign = 1, if x is negative; sign = 0, else
    int add = (sign << n) + (~sign) + 1;
    // plus 2 ^ n - 1
    return (x + add) >> n;
    // right shift n bits
  }
```

2.4 negate(x)

实际上这道题考察原码和补码的相互转换，假设x是原码，x'是补码，那么有x+x'=0xffffffff，则有x+x'+1=0，因此$-x = x' + 1 =\sim x + 1$，且有$-x' = x + 1 =\sim x' + 1$. 因此无论是对负数还是对正数，取相对数都只需要求$\sim x + 1$.

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

2.5 isPositive(x)

我们首先看x的符号位sign = 1 & (x >>31)，!sign=1当且仅当x非负，但这里还需要考虑x=0的情况. !!x=1当且仅当x非0，因此结合二者，(!sign) & (!!x)当且仅当x大于0.

```
int isPositive(int x) {
    int sign = 1 & (x >> 31);
    // sign = 1, if x is negative; sign = 0, else

    return (!sign) & (!!x);
    // !sign = 1, if x >= 0; sign = 0, else
    // !!x = 1, if x != 0; !!x = 0, else
    // (!sign) & (!!x) iff. x > 0
  }
```

2.6 isLessOrEqual(x, y)

x <= y 有两种情况，我们根据他们的正负号分别讨论：

1）x <0 ，在这种情况下要满足less or equal只能有x为负数但y非负，我们可以通过查看x，y的二进制数符号位很方便地获知；

2）x >= 0 ，在这种情况下y只能非负，而且y >= x. y非负很好判断，我们只需要判断x - y <= 0，方法是查看x - y的符号位（由于没有减号，故只能手动补码运算），注意在x，y都为非负数的情况下，减法不会超出int的范围.

以上两点包含了x <= y的所有可能，只需满足一点即可得知isLessOrEqual(x, y) = 1，反之，不在以上情况内必有isLessOrEqual(x, y) = 0.

```c
int isLessOrEqual(int x, int y) {
    // case 1:
    int sg1 = (1 & (x >> 31));
    // sg1 = 1, if x is negative; sign = 0, else
    int sg2 = (1 & (y >> 31));
    // sg2 = 1, if y is negative; sign = 0, else
    int case1 = sg1 & (!sg2);
    // x < 0 && y >= 0

    // case 2:
    int sub = x + (~y) + 1;
    // sub = x - y
    int sg_sub = (1 & (sub >> 31));
    // sg_sub = 1, if sub is negative; sign = 0, else
    int case2 = (!(sg1 ^ sg2)) & (sg_sub | (!sub));
    // x, y have same sign, and (x - y < 0 or x - y == 0)
    return case1 | case2;
  }
```

2.7 ilog2(x)

此题我们可以采用二进制表示数的方法，我们已知ilog(x)不超过32，可以用5位二进制数表示，而**二进制数的表示方法唯一**，这是我们解题的关键点.

我们先查看ilog的第5位二进制数，方法是将x右移$2^5 = 16$位，若x仍不为0，说明ilog大于$2^5 = 16$，因此ilog的第5位二进制数必为1（二进制数的表示方法唯一）. 注意，如果第5位数为1，则需令$x >>= 16$，如果第5位数不为1，则需令$x >>= 0$，即不变，综合起来可以表示为$x >>= (16\&((\sim exist) + 1))$，其中后面一项exist代表ilog的第5位二进制数，$(\sim exist) + 1 = 0xffffffff, if\ exist = 1$；$(\sim exist) + 1 = 0, else$.

与之类似，对于ilog的第4位二进制数，将x右移8位查看，方法同上；以此类推，直到ilog的第1位二进制数.

最后，我们根据每一位二进制数，将各位乘以权重，最后就可以得到ilog的值了.

```c
int ilog2(int x) {
    int x_16, x_16_exist;
    int x_8, x_8_exist;
    int x_4, x_4_exist;
```

```
    int x_2, x_2_exist;
    int x_1, x_1_exist;

    x_16 = x >> 16;
    x_16_exist = !!(x_16);
    // test whether x >> 16 == 0, which is equal to the fifth bit number of ilog2
    x = x >> (16 & ((~x_16_exist) + 1));
    // x = x >> 16 if x >> 16 != 0, else x = x

    x_8 = x >> 8;
    x_8_exist = !!(x_8);
    // test whether x >> 8 == 0, which is equal to the fourth bit number of ilog2
    x = x >> (8 & ((~x_8_exist) + 1));
    // x = x >> 8 if x >> 8 != 0, else x = x

    x_4 = x >> 4;
    x_4_exist = !!(x_4);
    // test whether x >> 4 == 0, which is equal to the third bit number of ilog2
    x = x >> (4 & ((~x_4_exist) + 1));
    // x = x >> 4 if x >> 4 != 0, else x = x

    x_2 = x >> 2;
    x_2_exist = !!(x_2);
    // test whether x >> 2 == 0, which is equal to the second bit number of ilog2
    x = x >> (2 & ((~x_2_exist) + 1));
    // x = x >> 2 if x >> 2 != 0, else x = x

    x_1 = x >> 1;
    x_1_exist = !!(x_1);
    // test whether x >> 2 == 0, which is equal to the second bit number of ilog2

    return x_1_exist + (x_2_exist << 1) + (x_4_exist << 2) +
     (x_8_exist << 3) + (x_16_exist << 4);
    // get value of ilog2 by bit numbers
  }
```

3. Floating-Point Operations

3.1 float_neg(uf)

此题主要分三种情况讨论：

1）NaN不合法：当exponent=0xff（加上偏移量为0x7f800000）且fraction!=0，直接返回原数uf;

2）uf非负：uf最高位为0，则直接改变最高位为1，方法是给原数OR 0x80000000;

3）uf负：uf最高位为1，则直接改变最高位为0，方法是给原数AND 0x7fffffff;

```
unsigned float_neg(unsigned uf) {
    unsigned fraction = uf & 0x007fffff;
    unsigned exponent = uf & 0x7f800000;
    unsigned sign;

    // NaN
    if (fraction != 0 && exponent == 0x7f800000)
      return uf;
    sign = (uf >> 31) & 1;

    // not negative
    if(sign){
      return uf & 0x7fffffff;
    }
    // negative
    else{
      return uf | 0x80000000;
    }
  }
```

3.2 float_i2f(x)

首先特判x=0，直接返回0即可.

然后我们可以观察，x除去符号位的二进制表示事实上通过简单的移位和舍入即可作为fraction part，其中移位的过程记录下来就可以变成exponent part. 具体来说，我们提取x符号位，并且将x统一转化为正数形式，这里注意特判x=0x800000000，-x也等于0x800000000，事实上通过移位，最终结果也是正确的，因此我们对于负数x统一取x=-x.

下面我们需要统计二进制数种最高位1的位置，这里通过一个while循环来实现. 对于最高位1低于或等于23位的，即不足fraction部分23位的，我们需要给末位补零；而高于23位的，我们需要进行舍入. 我们首先获得舍入的边界情况（即code中的limit_case），该情况中整数部分与x右移结果一致，而小数部分为1000...，即为十进制中的.5. x与边界情况比较. 向上舍入只有两种情况：1）若大于边界情况，则小数部分大于¿.5，向上舍入；2）若等于边界情况，且整数部分为奇数（右移后二进制末位为1），则满足向偶数的向上舍入. 除此之外，均为向下舍入，直接保留整数部分. 特别注意，向上舍入的时候fraction part可能会恰好溢出23位，此时应该令fraction part=0（即直接保留后23位）并给E加一.

最后，exponent=bias+E=127+E

```
unsigned float_i2f(int x) {
```

```
unsigned sign, temp, limit_case, E, fraction, exponent;
// if x == 0
if(!x)
    return 0;

sign = (x >> 31) & 1;
// sign = 1, if x is negative; sign = 0, else
if(sign)
    x = -x;
// if x is negative, turn it to positive
temp = x;
E = 0xffffffff; //E = -1

// test the position of highest 1
while (temp){
    temp = temp >> 1;
    E = E + 1;
}

// if the position is lower than 24, left shift
if(E <= 23){
    temp = x << (23 - E);
}
// if the position is higher than 24, right shift
else{
    temp = x >> (E - 23);
    limit_case = ((temp << 1) | 1) << (E - 24); // limit case of rounding (0.5)
    temp = temp & 0x7fffff; // reserve the fraction part
    if (x > limit_case) // if greater than limit case (> 0.5)
    temp = temp + 1;
    else if(x == limit_case && (temp & 1)) // if equal to limit case AND satisfying Ties To Even (= 0.5)
    temp = temp + 1;
    if (temp & 0x800000) // if fraction part overflow, then div by 2 and exponent ++
    E++;
}

fraction = temp & 0x7fffff; // get fraction part
exponent = 127 + E; // get exponent part
```

```
    return (sign << 31) + (exponent << 23) + fraction;
    }
```

3.3 float_twice(uf)

我们可以将uf分为以下情况讨论：

1）NaN以及inf，此种情况exponent=1111 1111，带shift为0x7f800000，乘2将返回原数；

2）乘2溢出，此种情况exponent=1111 1110，带shift为0x7f000000，溢出时符号位不变，exponent位全1（OR 0x7f8000000），fraction位全0（AND 0xff800000）；

3）非归约形式，此种情况exponent=0000 0000，考虑到非归约形式和归约形式之间的gap是连续且等距的，因此保留符号位，直接将exponent和fraction部分合并起来左移一位即可；

4）归约形式，此种情况直接将exponent加一即可，考虑到带shift，因此将exponent左移23位加一再右移23位，和fraction、sign拼合即可.

```
  unsigned float_twice(unsigned uf) {
    unsigned fraction = uf & 0x007fffff; // get fraction part
    unsigned exponent = uf & 0x7f800000; // get fraction part with shift
    unsigned sign = uf & 0x80000000; // get sign part with shift

    if (exponent == 0x7f800000)
        return uf; // NaN & inf

    if (exponent == 0x7f000000)
        return (uf | 0x7f800000) & 0xff800000; // mul 2 then overflow, return inf

    if (exponent == 0)
        return sign | (uf << 1); // De-normalized value

    return (sign + (((exponent >> 23) + 1) << 23) + fraction); // Normalized Value
}
```

4. Summary

我完成了本次实验的全部内容，总共用了220个operators,并在自动脚本测试中获得了全部的Corr和Perf分数.

在本次实验中，我对于整型、浮点数有了更深入的了解，尤其是浮点数部分，通过一系列的lab，我对于其表示、几种特殊情况、舍入等等有了更全面更细致的掌握. 尤其是限定运算符号集合以及使用个数以后，我更娴熟地掌握了各种运算符的运用，也对于算术右移等操作符有了进一步的理解.

除此之外，我还充分利用了Mask、二分、二进制表示等运算/算法技巧. 总体来说，题面易于理解，形式也很新颖，但有些题目仍有不小难度，解决不定令人抓耳挠腮，忽然突破又让人会心一笑，别有趣味.