移位运算说明文档

朱子晨　2013013297

1. int bitAnd(int x, int y)

   在没有&操作的情况下实现两个数得按位与操作，根据公式　x & y = ~(~x | ~y)　易得结果。例如：0x8000FFFF & 0x77777777 = ~(0x7FFF0000 | 0x88888888) = ~0xFFFF8888 = 0x00007777

2. int getByte(int x, int n)

   在 x 的二进制表示从后往前中抽出第 n 个字节的数，由于一个字节对应 8 个比特，而 16 进制表示中每一位数字表示的是 4 个比特，所以一个字节对应的是 16 进制中的相邻两位。如第 0 个字节，就是 16 进制的后两位；第 1 个字节就是 16 进制的倒数第三位和倒数第四位。为了抽出 16 进制中的两位数字，只需要将对应数字右移至末尾，再与 0xFF 作&操作。而为了将对应数字移至末尾，我们需要右移 n 个字节，即 8*n 个 bit 位，也就是 n << 3 。

   例如：x = 0x77778FFFF， n = 1 时，n << 3 = n*8 = 8, x >> 8 = 0x0077778FF， 0x0077778FF & 0xFF = 0xFF 。

3. int logicalShift(int x, int n)

   逻辑右移，由于逻辑右移在符号位为 0 时与算术右移无差别，而在符号位为 1 时右移　时补足的位仍然是 0, 所以需要根据符号位进行判断。大致思路是，首先提取出 x 的符号位，保持符号位的位置不变，其他位置为 0。然后对 x 进行算术右移，需要进行操作的是算术右移时补足的位，右移 n 位，补足的位数是 n – 1 位, 再使表示符号位的数右移 n – 1 位，（为了避免 n = 0 时产生特殊情况，右移 n – 1 位的实现方法为右移 n 位再左移 1 位），两个数再取异或。对 0 取异或为本身，对 1 取异或为取反，所以如果补足的符号位是 1 时，符号位右移 n – 1 时补足的位仍然为 1，这样取异或时，可以使补足的 1 变为 0；如果补足位为 0 时，与 0 取异或仍然是本身。对于非补足位，与 0 异或仍然是本身。

   例如：x = 0x8000FFFF，n = 4 时，((x >> 31) << 31) = 0x80000000，(x >> 3) = 0xF8000FFF，(0x80000000 >> 4) << 1 = 0xF0000000, 0xF8000FFF & 0xF0000000 = 0x08000FFF 。

4. int bitCount(int x)

   计算二进制表示时 1 的个数。为了在 40 个操作数之内完成，所以不能采用移一位算一位的方法。所以需要设计一些方法使原来的方法得到简化。而我这里设计了 0x11111111 这样一个数，可以使得四个比特位能够同时比较，也就是用 0x11111111 与 x 进行与操作，然后将 x 右移一位，再与 0x11111111 进行与操作，总共右移三次，然后将得到的数字相加，这样每四位的 1 的个数就存在结果的十六进制表示时的八个数字上，接下来只需要将这 8 个数字相加就能得到结果。因为 16 进制表示时每位上的数字不会超过 4 ，所以我们可以先让这个结果右移 16 位和原来得结果相加，这样得到的新结果的后 4 位上的数字之和为整个 x 二进制 1 的个数。同理，每个位数上的数字不会超过 8。但是同样的方法不能够再使用了，因为如果再将新结果右移 8 位相加后，每个位数上的数字表示的是原二进制 16 个位数上的 1 的个数，这个个数是会超过 15 的，这样的话一个位置的 16 进制的数字是没有办法表示 16 的。所以这里我们需要换一种方式来相加，我们设计了数字 0x00000F0F，新结果与这个数字进行与操作，新结果再右移 4 位再和 0x00000F0F 与操作，两者的与结果相加，这样两个 byte 表示的数字之和就是 x 二进制的 1 个数，此时一个 byte 表示的数字是可以大于等于 16 的，最后的只需要将两个 byte 相加再和 0x3F(因为 1 的个数不会超过 32，保证二进制第六位及一下都是 1，所以设计的是 0x3F)进行与操作就能得到最后的答案。后两个 byte 位的相加也很简单，只需要将数字右移一个 byte 然后和原数相加那么得到最后一个的 byte 就是最后答案，由于 1 的个数不会超过 32 个，所以放心的使用 byte 来存储吧！

   例如：x = 0x8000FFFF，0x8000FFFF&0x11111111 = 0x00001111，再将 0x8000FFFF 依次移位再与 0x11111111 进行与操作，得到结果分别为，0x00001111，0x00001111，0x10001111，相加之后的结果为 0x10004444，右移 16 位之后和本身相加为，0x10005444，和 0xF0F 与操作之后是 0x404，右移 4 位之后和 0xF0F 与操作的结果为 0x504，相加之后的结果为 0x908，在右移 8 位之后与本身相加 0x11，再和 0x3F 与操作之后为 0x11，十进制结果为 17 。

5. int bang(int x)

判断 x 是否为 0，0 返回 1，否则返回 0。我使用的大致判定规则是一个数的相反数的符号和自己的符号是否是一样的，得到相反数的方法就是取反加一，而得到符号的方法更简单，就是右移 31 位，除了 0x80000000 这个特殊的数之外其余的数都能用这个规则判定。为了也能判定这个特殊的数，我们只需要稍微改进一下判定规则即可。我们注意到 0 的符号位是 1 而这个特殊的数的符号位为 1 所以我们得到某个数的符号位和其相反数的符号位之后，进行或操作，这样这个数和其相反数保证必须同时为 0，最后取反，那么最后一位的值必然是我们的结果，最后和 0x1 进行与操作去掉没有用的其他位的数字，就能得到最后需要的结果。

6. 例如：x = 0x80000000，x >> 31 = 0x1，~x + 1 = 0x80000000，(~x + 1) >> 31 = 0x1，0x1|0x1 = 0x1，~(0x1) = 0xFFFFFFFE，0xFFFFFFFE&0x1 = 0x0，返回 0 。

7. int tmin(void)

返回 32 位 int 中的最小值。很明显，最小值的表示方式是 0x80000000，只需要将 0x80 左移 24 位就能得到 int 中的最小值。

(0x80) << 24 = 0x80000000，此结果即为 32 位 int 的最小值。

8. int fitsBits(int x, int n)

计算 x 是否能被 b 位二进制表示，判断机制十分简单，无论 x 的正负，为了判断 x 是否能被 n 位二进制表示，我们将 x 左移 32-n 位然后再右移 32-n 位，得到的结果与 x 作比较，如果一致的话，说明在 x 的 32 位二进制表示之中前 32-n 位是没有用的，换句话说就是 x 能够被 n 位二进制数表示。根据这样的原理，我们只需要判断两个数是否相同就能判断 x 是否能被 n 位二进制表示。因为 32-n 是方便表示的，相当于 32 加上 n 的相反数，而 n 的相反数即为(~n+1)。觉我们判断两个梳是否相同时就可以使用异或的操作，如果相同则结果全为 0，如果不同则结果中至少含有一个 1。这时我们可以使用!操作判断是否为 0。但是在操作过程中，函数 n=32 时有报错，我想可能时移位操作时有些小 bug，于是在程序的结果中添加了 n=32 的判断，如果 n=32，x 是一定能被表示的。所以这里只需要判断 temp 是不是 0，得到结果再和之前的异或取反结果取或操作就能解决这个小 bug 了。

例如：x=32，n=6，我们可以得到 32-n = 0x21 + (~n) = 0x21 + 0xFFFFFFF9 = 0x1A， 而(x << 26) >> 26 = 0x0，而 0x0^0x20=0x20，!0x20 = 0x0， !0x1A = 0x0，0x0|0x0 = 0，32 不能被 6 位表示。

9. int divpwr2(int x, int n)

计算 x/(2^n)，其实处理这个运算并不算麻烦，右移 n 位能得到结果。但是比较麻烦的事情是这里需要向 0 取整，x 为正数(x >> n)的结果和向 0 取整是一致的，但是负数时(x >> n)的结果会比向 0 取整的数少 1 ，这时我们需要判断 x 是否为负数如果 x 为负数，那么 x 右移 n 位之前需要补上(2^n- 1)，这样能在右移 n 位之后保证得到的数是向 0 取整的。判定 x 是否为负数很简单，只需要把 x 右移 31 位得到数要么是 0x0 要么就是 0xFFFFFFFF，把这个数和(2^n- 1)作与操作，就可以得到 x 再右移之前需要相加的数了，因为如果 x 是正数，那么相加的数为 0，如果 x 是负数那么相加的数为 2^n- 1。

例如：x = -33，n = 5，x >> 31 = 0xFFFFFFFF，(1 << 5 + 0xFFFFFFFF) = 0x0000001F，0xFFFFFFFF& 0x0000001F = 0x0000001F，x + 0x0000001F = 0xFFFFFF1F + 0x0000001F = 0xFFFFFF30，0xFFFFFF30 >> 5 = 0xFFFFFFFF = -1 。

10. int negate(int x)

取 x 的相反数，由于 x 与 x 的相反数之和为 0，根据这样的性质，我们可以想象 x 先与~x 求和得到的结果必然是 0xFFFFFFFF，再这个基础上再加上 1 就能得到 0x0 这样一个理想的结果，所以 x 的相反数是~x+1。

例如：x = 0xFFFFF00F，x 十进制上面表示的是-4081 这个数字，~x + 1 = 0x00000FF0 + 1 = 0x00000FF1，而 0x00000FF1 表示的数字刚好是 4081 。

11. int isPositive(int x)

判断 x 是否为正数，方法十分简单只要判断 x 的符号位是否为 0 即可，!(x >> 31)就能知道 x 的符号位是 0 还是 1，但是我们遇到一个问题，就是在判断 0 的时候出现了 bug，因为 0 的符号位也是 0，但是 0 并不是正数。我们需要额外判断 x 是否为 0，判断 x 为 0 就更加简单，!x 的操作就能完成。现在只需要把逻辑理清楚就能写出表达式，当且仅当 x 不为 0 且 x 的符号位为 0 的时候 x 为正数，那么(!(!x))&(!(x >> 31))的表达式自然就出来了。

例如：x = 0x8FFFFFFF，!!x = 1，!(x >> 31) = 0，所以 0x1&0x0 = 0，这个 x 不是正数。

12. int isLessOrEqual(int x, int y)

判断 x 是否小于等于 y，我们可以分两种情况进行讨论，一种是符号位相同的结果一种是符号位不同的结果，两个结果取或操作就能得到我们最终的比较结果。符号不同时，判断 x 与 y 的大小关系是容易的，符号位不同时当且仅当 x 符号位为 1 而 y 符号位为 0 的时候 x 小于等于 y，那么提取出符号位 x >> 31，类似的有 y >> 31，那么根据刚才叙述的逻辑，符号不同时的判断表达式为(x >> 31)&(!(y >> 31)) 。而符号相同时，判断的条件是什么呢?我们可以直接使用 y – x = y + ~x + 1 这样的式子来判断结果的符号位，如果符号位是 1，y < x，如果符号位是 0，则 y >= x，符号位是 0 时是我们想要的结果。所以符号相同时，当且仅当 y + (~x) + 1 的结果符号位是 0 时 x 小于等于 y 。但是在计算 y 和 x 的差时，必须保证相同符号，不然符号不同的话，一个负数减去一个正数在 32 位二进制中可能得到的是一个负数，所以事先需要判断是否为相同符号，根据之前取出的 x >> 31 和 y >> 31 可以有异或操作来判断符号位是否一致。符号位一致的情况是，当且仅当(x >> 31) 和(y >> 31)的两个结果相同，且 y + ~x + 1 的结果的符号位为 0 时有 x 小于等于 y，那么就有如下表达式((!(signY^signX))&(!signY_X))，其中 signX = x >> 31，signY = y >> 31，signY_X = (y + (~x + 1)) >> 31 。最后两个情况的结果再取或操作。

例如：x = 0x0000001F，y = 0x0000000F，signX = x >> 31 = 0x0，signY = y >> 31 = 0x0，signY_X = (y + (~x) + 1) >> 31 = (0x0000000F + 0xFFFFFFE0 + 1) >> 31 = 0xFFFFFFF0 >> 31 = 0xFFFFFFFF，signX & (!signY) = 0x0 & 0xFFFFFFFF = 0x0，((!(signY^signX))&(!signY_X)) = ((!(signY^signX))&(!0xFFFFFFFF)) = ((!(signY^signX))&0x0) = 0x0，0x0 | 0x0 = 0，!!0 = 0 。

13. int ilog2(int x)

求以 2 为底数的 x 的对数，向下取整，由于 x 必须大于 0，所以这道题其实就是求正数 x 在二进制表示时数值为 1 的最高位 bit 位位数再减去 1，由于一共有 32 个 bit 位，如果一位一位的比较，势必会超过 90 个操作数，所以我们必须用其他更为简便的查找方法。我们这里采用的是二分查找方法，先判断 1 的最高位数在前 16 位还是后 16 位，判断方法非常简单，将 x 数右移 16 位，如果最高位数在后 16 位，那么右移 16 位之后得到的数字必然不是 0，而如果最高位数在前 16 位，右移 16 位后 x 将全部变成 0 。接下来要解决怎么把这样得信息传递到下次判断之中，因为是二分查找，所以下次查找是查找 8 位，可是怎么知道是在前 16 位查还是在后 16 位查。首先，得知最高位数后 16 位数，那么结果是必然大于等于 16，类似的，如果是在后 8 位，一定会大于等于 8，我们现在用一个变量来存储这样一个必然大于等于的值，因为是二分查找，有边界条件，不断找到找到这样一个值之后，就能得到一个精确的结果。这样一个临时变量也就是我们需要返回的结果。如果我们能判断最高位数在前 16 或者后 16，我们可以用这样一个 0 或 1 判断结果右移 4 位，就会得到 16，而之后的结果就在这个 16 的基础上相加。那么问题来了，怎么让下次 8 位判断知道在前 16 后 16 位查找呢?其实有了临时变量后十分简单，直接可以让 x 右移这个临时变量的值就行，接下来就可以重复类似判断前 16 和后 16 的操作。因为知道了 x 最高 1 位数大致位置，就可以通过右移，去掉一些无用的位数，就能更方便的进行运算。不过一定记住，这个临时变量一定是累加的，中途的右移操作也是依托于这个变量，而且这个变量会记录最终的结果。可以通过例子更加明白二分查找的过程。

例如：x = 0x00110201，temp = (!!(x >> 16)) >> 4 = (!!(0x00000011)) << 4 = 1 << 4 = 0x10000 = 16，temp = temp + ((!!(x >> (16 + 8))) << 3) = temp + ((!!0x0) << 3) = temp = 16，temp = temp + ((!!(x >> (16 + 4))) << 2) = temp + ((!!0x01) << 2) = temp + 4 = 20，temp = temp + ((!!(x >> (20 + 2))) << 1) = temp + ((!!0x0) << 1) = temp = 20，temp = temp + ((!!(x >> (20 + 1)))) = temp + (!!0x00) = 20，最高为 1 的位数 21，最后结果为 20 。

14. unsigned float_neg(unsigned uf)

浮点数求相反数，浮点数取反其实非常简单，只要把将符号位取反就行，而不影响其他位的数值即可，那么只需要和 0x80000000 取异或操作就能够实现，因为和 1 取异或是取反而和 0 取异或则是取本身。但是题目里面有额外的要求，当参数是 NaN 的时候，应该返回原参数，不做符号位的改变。所谓的 NaN 也就是，表示 exp 的八个 bit 位都是 1 并且表示 frac 的 bit 位不全为 0 的情况下，我们说这样的浮点数表示的是 NaN，允许 if 操作的话，我们其实只需要比较不考虑符号位的情况下，浮点数解析成 int 的值是否大于(0xFF << 23)我们就能知道这个数是不是 NaN，因为除开符号位，如果这个数大于(0xFF << 23)的话，那么对应的 exp 的八个 bit 位必然全部为 1，而且 frac 部分必然是大于 0 的。所以我们只需要判断非符号位与(0xFF << 23) 的大小，然后根据情况返回结果。如果大于这个数，那么返回本身，否则的话，符号位取反，就是需要返回的结果。

例如：uf = 0x10001111，(uf&(~(0x80 << 24)) = 0x10001111 & 07FFFFFF = 0x0001111 < 0x7F80000 。返回 uf^(0x80 << 24) = 0x10001111 ^ 0x80000000 = 0x00001111 。

15. unsigned float_i2f(int x)

int 类型转 float 类型，我们可以理解 int 是没有 exp 部分的，而 frac 部分 int 有 31 位，而 float 只有 23 位，所以很有可能会丢失精度，这时必须要有心理准备的，如果有精度丢失，我们需要根据针对浮点数的"四舍五入"，来进行进位。我们要有准备应对这个情况，但是这只是属于细节，真正重要的是怎么尽量减少丢失的精度。为了方便，我们只讨论 int 为正数的情况，因为如果 int 为负数的话，我们可以转化为正数后，得到 float 结果后，再将符号位变为 1 。我们只需要在转化 int 之前把符号位提取出来最后再加上去就行。

首先，我们该如何减少精度的丢失，这里其实很简单，我们忽略掉 int 二进制表示时(从左往右数，第一个出现的 1 之前的 0，这样在浮点数表示时，我们就能尽量减少精度丢失)。采用循环的方法，从左往右遍历直到出现第一个 1 为止，用一个变量记录(0 的个数加一)，变量初始值为 1，循环停止之前每经过依次循环就自增 1，结束条件就是(x & 0x80000000)为真，否则 x 就左移一位需要提醒的事情是，如果输入为 0，程序可能进入死循环，那么程序需要在最开始进行判断，如果 x 为 0 就返回 0，这样就避免了死循环，根据这个先决条件，我们能够确定最后的浮点数表示中 frac 部分至少存在一个 1。还要额外提醒一下这个(从左往右数)第一个 1 的作用，这个 1 是不会显示在最后的 frac 部分的，因为一般情况下浮点数解析时个位数会自动变为 1，除非 exp 部分全为 0，所以实际上 frac 部分显示的只需要原来的 32 - temp2 位，这就是为什么这个变量储存的是(从左往右数)第一个 1 之前 0 的个数加一，而不是 0 的个数。

现在假设我们知道 x 中第一个 1 之前 0 的个数(temp2 - 1)，怎么得出最后的 float 数字呢?我们现在 x 已经左移到符号为 1 的情况，如果右移 8 位，注意是 8 位而不是 9 位，因为第一个 1 之后的数字才有用,第一个 1 是在浮点数解析时会自动补上的，也就是说我们不必在乎这个原始 x 从左往右数的第一个 1 在浮点数二进制中的表示。另外，由于是 unsigned 类型，右移时符号位会一直补足 0。我们已经得到 float 中 frac 部分的大致数字，然后只需要加上 exp,符号位，和最后丢失精度的进位就是最后的结果。

我们知道最高位 1 之前的 0 的个数加 1 为 temp2，又由于之前右移 8 位时留了一个 1 在从左往右数的第 9 位上，所以计算 exp 部分时可以少算一个 1 。那么根据 exp 部分的计算公式，我们可以得到 exp 部分应该表示的数为 127+(31 – temp2) = 158 – temp2 。将这个数左移 23 位到 exp 的位置，我们得到转为 float 之后的 exp 部分其他部分全为 0 。

现在还有一个需要计算，就是当我们舍去某些位时，会导致精度的丢失，而此时我们需要根据针对浮点数的"四舍五入"，判断 flag 是否为 1 。现在提取出会被舍去的部分，也就是 temp0 的(从左往右看)后 9 位，可以用和 0x01ff 的与操作，提取出来，如果这一部分是大于 256(0x0100)的，那么应该有进位操作，如果这一部分是等于 256 且 temp0 的(从左往右看)倒数第 10 位上是数字 1 的话，也应该有进位，这种情况的判断是和 0x03ff 与操作，看结果是否与 0x0300 相等。根据 if 判断在这两种情况下都使得 flag=1，最后可以将求得的符号部分，exp 部分，frac 部分，和最后的进位部分相加就能得到最后的结果。

例如：x= 0xFFFF8888，由于 x < 0，那么求得绝对值 abs = -x = 0x00007778，符号位 sign = 0x80000000，temp0 表示需要右移的数，也就是 x，进入循环判断，循环过程略去，循环跳出时，tmp=0xEEF00000，temp2 = 18，temp0 = 0xDDE00000，根据 if 判断得到 flag = 0 。那么最后的答案将会是 sign + (tmp >> 8) + ((158 – temp2) << 23) = 0x80000000 + 0x00EEF000 + (140 << 23) = 0x80000000 + 0x00EEF000 + (0x0000008C<< 23) = 0x80000000 + 0x00EEF000+0x46000000 = 0x80000000 + 0x46EEF000 = 0xC6EEF000

表示浮点数的 0xFFFF8888 。

16. unsigned float_twice(unsigned uf)

求浮点数的两倍 ，需要考虑的情况分为，当 exp 部分全为 0 时，因为此时表示的 frac 部分加入小数部分时，非小数位会变为 0 而不是 1(0.frac 这样的形式)，如果这样的数乘以 2，我们只需要把 x 的后 23 位左移一位，然后符号位不变即可，因为如果小数第一位为 1，那么左移一位，刚好有一个 1 能够加入 exp 部分，此时的 frac 加入数值的小数部分时，非小数位由于 exp 不全为 0 会自动变为 1(1.frac 这样的形式)，如果小数的第一位为 0，那么左移一位正好也是乘以 2 而不影响其他部分的位置，实现方法为先提取符号位 uf&0x80000000，然后提取出后 23 位 uf & 0x007FFFFF，然后左移一位之后与之前的符号结果做或操作；当输入为 NaN，返回参数本身；除开这两种情况，exp 部分加 1(实现方法 uf + 0x00800000)就能够返回了。

例如：uf = 0x0070000F，uf & 0x7F800000 = 0，(uf & 0x007FFFFF) << 1 =0x 00E0001E，uf & 0x80000000 = 0，0x00E0001E | 0 = 0x00E0001E，返回 0x00E0001E 。