1. bitAnd

可以通过 x & y = ~(~(x & y)) = ~(~x | ~y) 实现。

2. getByte

在十六进制中从最低有效位往最高有效位取第 n 个 byte，也就是在二进制下，将数字右移 8 * n 位，再取最后 8 位。前者可以通过 x >> (n << 3)，后者可以通过和 0xFF 进行位与操作来实现。

3. logicalShift

显然，将 x 移动 n 位，再将结果与 0x0...FFF 进行位与操作即可得到结果。考虑 0x0...FFF 的构造，可以采用(1 << (32 + ((~n) + 1))) + (~0)，但要注意对 x 等于 0 时结果等于 0，该方法并不适用，因此需要单独对 0 通过((~(!n)) + 1)构造出 0xFFFFFFFF。

4. bitCount

先得到掩码 A：0101 0101 0101 0101 0101 0101 0101 0101；B：0011 0011 0011 0011 0011 0011 0011 0011；C：0000 1111 0000 1111 0000 1111 0000 1111；D：0000 0000 1111 1111 0000 0000 1111 1111；E：0000 0000 0000 0000 1111 1111 1111 1111。通过 x 与 A 的位与再加上 x 右移一位后与 A 的位与，可以每两位计算出 1 的个数。用同样的方法换用 B/C/D/E 操作，可以计算出每 4/8/16/32 位的 1 的个数，也就是在最后得到了用二进制表示的 x 中 1 出现的次数。

5. bang

也就是对不为 0 的 x 返回 0，对为 0 的 x 返回 1。考虑到为补码存储，右移 31 位后，小于 0 的数一定是 1111 1111 1111 1111，大于等于 0 的数一定是 0000 0000 0000 0000，加一后分别为 0 和 1，只要找到合适的对应关系。对为 0 的 x，其有特殊的性质为取反加一后仍然是其本身，而其他数则为它的相反数。此时若再与其本身进行位或操作，也即 x | ((~x) + 1)，非 0 数必定为第一位一 0 一 1，位或后第一位为 1；0 操作后第一位仍然为 0，再右移 31 位后加 1 即可得到。

6. tmin

最小的二进制补码整型数字，也就是 1000 0000 0000 0000，可通过将 0000 0000 0000 0001 左移 31 位得到。

7. fitsBits

需要先对 x 右移 n – 1 位，再判断余下的数字是否全为 0 或者全为 1 即可得到。其中 n – 1 可以通过~((~n) + 1)得到，而!temp | !(temp + 1) 会对全为 0 或全为 1 的数返回 1，否则返回 0。

8. divpwr2

可以先判断是不是负数，如果是负数就构造 0xF...FFF，再通过构造~((~0)<<n)，来做到对负数的向上取整。而正数不需要额外操作，直接右移 n 位即可。

9. negate

要取 x 的相反数，且 x 以补码方式存储，只要对 x 取反加一，可通过(~x) + 1 得到。

10.isPositive

与第 5 题类似，右移 31 位后，小于 0 的数一定是 1111 1111 1111 1111，大于等于 0 的数一定是 0000 0000 0000 0000，但此处若直接加 1，当 x = 0 时返回 1，不符合要求且较难处理。考虑到只有!0 = 1，可以利用该性质，先取!((x >> 31) + 1)，此时若 x >= 0，返回 0；若 x < 0，返回 1。将结果与!0 进行位或操作，则若 x != 0，值不变；若 x = 0，返回 1。再对整体进行!((!((x >> 31) + 1)) | !x)，即可得到。

11.isLessOrEqual

可以先通过 sign_x = (x >> 31) & 0x01 来得到 x 与 y 的符号，若 x 大于等于 0，结果 0，否则为 1；若 x 与 y 同号，则 sign_x ^ sign_y 为 0，否则为 1；将同号与异号的情况分别讨论，异号时只需要判断 x 是否小于 0，也就是通过 isSameSign & sign_x，异号且 x 小于 0 时返回 1，否则返回 0，以此防止负数减正数时的溢出问题；同号时判断 y – x 是否大于等于 0，与之前的方法相同，对!(((~x) + 1 + y) >> 31)，若 y – x 小于 0 则得到 0，否则得到 1。如此可以判断 x 是否小于等于 y。

12.ilog2

因为 x 长 32 位，通过 x >> 16，可以判断只取前 16 位是不是仍然大于 0，若大于 0，!(x >> 16)返回 0，再次进行非运算后是 1，然后左移 4 位，到 a 的位置，就说明这个数大于 16，二进制中的第一个 1 肯定出现在前 16 位中，也就是函数的返回值一定大于 1。再用相同方法将需要判断的长度缩短到前 8 位，判断前 8 位是否大于 0。如此缩小范围可以逐步判断最高位的 1 在哪个位置，也就是可以得到 log(x)。