解题思路

总操作数：186

1 bitAnd

根据公式`a & b = ~(~(a & b)) = ~(~a | ~b)`即可

```
int bitAnd(int x, int y) {
  // a & b = ~(~(a & b)) = ~(~a | ~b)
  return ~(~x | ~y);
}
```

2 getByte

- 主要利用任何位跟1按位与不变这一性质
- 先将输入x右移指定位数，然后跟1 &

```
int getByte(int x, int n) {
  // any bit & 1 remain unchanged
  int step = n << 3;
  int valid = x >> step;
  return (valid & 0xff);
}
```

3 logicalShift

- 默认提供的>>是算术右移，`x>>n`仅在非负时生效
- 要保证高n位为0，考虑将高位跟0 &
- `((1 << 31) >> n) << 1`得到`11...0..0`，高n位是1，对其取反~，即得到高n位为0

```
int logicalShift(int x, int n) {
  // through ((1 << 31) >> n) << 1 we get a num like 11...0...0, which has n bits
1
  // then through ~ we get 00...1...1
  // and any bit & 1 remain unchanged
  return (x >> n) & ~(((1 << 31) >> n) << 1);
}
```

4 bitCount

- 考虑8位8位地计算其中1的个数（当然也可4位4位或16位16位计算）
- 先通过移位和按位或得到`0000 0001 0000 0001 0000 0001 0000 0001`
- 将输入x与上述数按位与后右移1位，重复7次，得到形如`0000 0110 0000 1001 0000 0101 0000 1101`的数，每8位对应的十进制数即为对应8位中1的个数

- 将上述数不断右移8位与`1111 1111`按位与并相加，得到总的1的个数

```
int bitCount(int x) {
  // 0000 0001 0000 0001 0000 0001 0000 0001
  // count by 8 bits
  int a = 1 | (1<<8) | (1<<16) | (1<<24);
  int c = (x & a) + ((x >> 1) & a) + ((x >> 2) & a) + ((x >> 3) & a)
          + ((x >> 4) & a) + ((x >> 5) & a) + ((x >> 6) & a) + ((x >> 7) & a);
  // amount of 1 in low 8 bits
  int b = 0xff;
  int d = (c & b) + ((c >> 8) & b) + ((c >> 16) & b) + ((c >> 24) & b);
  return d;
}
```

5 bang

- 观察32位有符号数，只有0符合“原码和补码的符号位均为0”这一条件
- 因此符合这一条件为0返回1，不符合为非0返回0

```
int bang(int x) {
  // if x == 0 then return 1
  // else return 0 because any right shift will get 0
  int complement = ~x+1;

  // 0 is the only one 0 | -0 = 0
  int bits = (complement | x) >> 31;
  return bits + 1;
}
```

6 tmin

`1 << 31 = 1000...00`

```
int tmin(void) {
  // tmin = 1000...00
  return (1 << 31);
}
```

7 fitsBits

- 若x能用n位表示，那么至少右起第n-1位（从0开始）就是符号位
- 这时将二进制数左移32-n位再右移回去应该还是原数不变
- 若发生改变，则说明第n-1位左边仍存在有效位，x不能用n位表示

```
int fitsBits(int x, int n) {
  // get two's complement
  // 32 - n
  int shift = 32 + (~n + 1);
  // check is any bit lost
  return !(x ^ ((x << shift) >> shift));
}
```

8 divpwr2

- 直接`x>>n`仅在非负时成立，负数时不会向0舍入
- 利用`x>>31`判断出符号位后，利用`(1<<n)+~0`手动舍入

```
int divpwr2(int x, int n) {
  // x >> n fail when x < 0
  // get sign
  int sign = (x >> 31);
  // remove 1
  int s = sign & ((1 << n) + ~0);
  return ((x + s) >> n);
}
```

9 negate

直接`反码+1`即可

```
int negate(int x) {
  // 0001 <-> 1001 = 1110 + 1
  return ~x + 1;
}
```

10 isPositive

- 考虑直接观察最高位，但是0要特殊处理
- 先用`(x >> 31) & 1`得到符号位
- 上式与`!x`按位或，将x==0的情况与负数归为一类
- 最后再求反即可

```
int isPositive(int x) {
  // find the sign bit
  // 0 is special
  return !(((x >> 31) & 1) | !x);
}
```

11 isLessOrEqual

- 分为两种情况
    - x < 0, y >= 0 `(!y_sign) & x_sign`
    - x、y同好，但x <= y `(!(y_sign ^ x_sign)) & (((x + ~y) >> 31) & 1)`
- 移位获取x和y的符号后将两种情况按位或即可

```
int isLessOrEqual(int x, int y) {
  // get the sign bit
    int x_sign = (x >> 31) & 1;
    int y_sign = (y >> 31) & 1;
    // x and y have the same sign and x <= y
    int z = (!(y_sign ^ x_sign)) & (((x + ~y) >> 31) & 1);
    //y >= 0,x < 0;
    return z | ((!y_sign) & x_sign);
}
```

12 ilog2

- 求2的对数等价于找到二进制表示下最高位的1的位置
- 考虑利用前面题bitCount计算最高位1
- 首先不断将x与x右移的结果按位或，将最高位1的右边位全部置为1，得到形如`00...011.11`的数
- 然后利用bitCount计算1的个数
- 得到的结果-1即为最终结果（-1利用negate即可）

```
int ilog2(int x) {
  int a, b, c, d;
  // set the left bit of the most left 1 all to 1
  x = x | (x >> 1);
  x = x | (x >> 2);
  x = x | (x >> 4);
  x = x | (x >> 8);
  x = x | (x >> 16);

  // get the number of bit 1
  // uses bitCount
  a = 1 | (1<<8) | (1<<16) | (1<<24);
  c = (x & a) + ((x >> 1) & a) + ((x >> 2) & a) + ((x >> 3) & a)
          + ((x >> 4) & a) + ((x >> 5) & a) + ((x >> 6) & a) + ((x >> 7) & a);
  // amount of 1 in low 8 bits
  b = 0xff;
  d = (c & b) + ((c >> 8) & b) + ((c >> 16) & b) + ((c >> 24) & b);

  // return count - 1
  return d + (~1 + 1);
}
```

13 float_neg

- 先求出单精度浮点数的exp和frac
    - `int exp = (0xff << 23) & uf`
    - `int frac = 0x7fffff & uf;`

- NaN则直接返回，即`exp == 0x7f800000 && frac`
- 正常情况直接修改符号位即可`uf ^ (1 << 31)`

```
unsigned float_neg(unsigned uf) {
  // 1 8 23
  int exp = (0xff << 23) & uf;
  int frac = 0x7fffff & uf;
  // NaN  exp is all 1 and frac is not all 0
  if (exp == 0x7f800000 && frac)
    return uf;
  return uf ^ (1 << 31);
}
```

14 float_i2f

- 先将0和Tmin这类特殊情况单独处理掉
- 其余情形
  - 先获取符号位，并将所有数取绝对值处理（利用negate）
  - 获取exp。根据浮点数`V = (-1)^x * 2^E * M`,将x不断除2直到0，所用次数为E，通过bias得到exp
  - 获取frac。frac需要将x四舍五入移位，小则移位，大则舍掉

    ```
    if ((x & 0x80) && ((frac & 1) || ((x & 0x7f) > 0)))
        frac++;
    ```

  - 最后按照浮点数定义相加即可`sign + ((e + bias) << 23) + frac`

```
unsigned float_i2f(int x) {
  int sign = x & (1 << 31);
  int e = 0x1f;
  int bias = 0x7f;
  int exp, frac;
  if (!x)
    return 0;
  // tmin -2^31
  if (x == 0x80000000)
  // 1 e=exp-bias=31 f=0
    return 0xcf000000;

  // abs(x)
  if (sign)
    x = ~x + 1;
  // get exp
  exp = e;
  while (!(x & 0x80000000))
  {
    x = x << 1;
    exp = exp - 1;
  }
  e = exp;
  // get frac
```

```
  frac = (((0x7fffffff) & x) >> 8);
  if ((x & 0x80) && ((frac & 1) || ((x & 0x7f) > 0)))
    frac++;
  return sign + ((e + bias) << 23) + frac;
}
```

15 float_twice

- 先获取exp，`exp = (uf >> 23) & 0xff`
- NaN或INF仍返回NaN和INF，即`exp == 1111 1111`
- 非规格化数，直接左移即可
- 规格化数，存在溢出可能
    - `exp == 1111 1110`时，2倍会溢出，因此返回INF
    - 其他情况，直接给exp+1即可

```
unsigned float_twice(unsigned uf) {
  // 1 8 23
  int exp = (uf >> 23) & 0xff;
  // exp == 1111 1111 NaN or INF
  if (exp == 0xff)
    return uf;
  // exp == 0000 0000
  if (!exp)
    return (uf & 0x80000000) | (uf << 1);
  // exp == 1111 1110, 2*f will overflow
  if (exp == 0xfe)
  {
    exp = 0xff;
    return (uf & 0x80000000) | (exp << 23);
  }
  // common case
  return uf + (1 << 23);
}
```

感想

- 从位级编码的角度思考设计这些函数确实极具挑战性，有些问题一时半会儿很难想到纯用位操作的解决方法
- 有些解法看似解决了问题但一些边界条件特别是Tmin实则存在问题，需要寻找通解
- 对计算机编码有了更加深入的理解