第一题:

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

用非和或实现与的功能，德摩根，取反之后与，得到与的非，再非即可。

第二题:

```
int getByte(int x, int n) {
    return (x >> (n << 3)) & 0xFF;
}
```

从 x 中拿到对应字节，即让 x 右移 8*n 个位，在和 0xff 与即可。

第三题:

```
int logicalShift(int x, int n) {
return ~(((1 << 31) >> n) << 1) & (x >> n);
}
```

实现逻辑右移，让右移之后左边的位变为 0 即可。

第四题:

```
int bitCount(int x) {
int t1 = (0x55 << 8) + 0x55;
int t2 = (0x33 << 8) + 0x33;
int t3 = (0x0f << 8) + 0x0f;
int t4 = 0xff;
int t5 = (0xff << 8) + 0xff;
t1 = t1 + (t1 << 16);
t2 = t2 + (t2 << 16);
t3 = t3 + (t3 << 16);
t4 = t4 + (t4 << 16);
x = (x & t1) + ((x >> 1) & t1);
x = (x & t2) + ((x >> 2) & t2);
x = x + (x >> 4) & t3;可直接右移，再相加
x = x + (x >> 8) & t4;
x = x + (x >> 16) & t5;
return x;
}
```

算出数中的一的个数，采用分治的思想，位数为 2 时，((x>>1)&1)+(x&1)就可以得到两位上一的和（即第一个位数的值加上第二个位数的值），当位数为 4 时，将 4 平均分成两份，分别计算每份的一的个数再相加即可，由此在位数增加时，可像此一样向上递归，完成运算。再算完分成 4 位的时候（如上方红色标记所示），该做法是由于即使位数全是一，加两次后为 0100，在发生右移时不会出现多余的一，故直接右移相加。

第五题:

```
int bang(int x) {
```

```c
int t = (x >> 31) & 1;
return (((((~x + 1) >> 31) & 1) ^ t) | t) ^ 1;
}
```

实现非，对于大多数，取反加一符号位必改变（除 0 和 0x80000000），利用这个，将变换后的符号位与之前的符号位异或再取反即可，但由于 0x80000000 的干扰，需要让异或的结果与自身或，把 0x80000000 纳入到异或结果中，同时由于我们要返回 0 和 1，再拿出符号位之后，其他位都是 0，因此不能直接取反，要采用异或的方式（0x00000001），任何数与 0 异或任为其本身，与 1 异或为其反，我们只要符号位取反，故与 0x00000001 异或。

第六题:

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

返回最小数，即返回 0x80000000。

第七题:

```c
int fitsBits(int x, int n) {
    int sign = ~n + 33;
    return !(x ^ (x << sign >> sign));
}
```

X 是否能被 n 位补码表示，即判断符号位及之前所有位是否一致，故用 32-n 算出符号位及之前所有位的个数，再将 x 先左移再右移（构造一个属于 n 位补码的数），判断是否相等（与自身异或再取反）即可。

第八题:

```c
int divpwr2(int x, int n) {
int bias = (1 << n) + ~0;
return x + ((x >> 31) & bias) >> n;
}
```

实现2的幂除法，由于右移是向下取整，需要一个偏置值（2^n-1），在x为正时，利用符号位为0的特性，右移31位与偏置值与，得到0，在x为负时，利用符号位为1的特性，右移31位与偏置值与，得到偏置值，在将与的结果和x相加得到最终值，之后右移n位即可。

第九题:

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

求相反数，取反加 1 即可。

第十题:

```c
int isPositive(int x) {
    return !(x >> 31) & (!!x);
}
```

判断是否为正，由于!!0 等于 0，而任意非零整数的!!x 为 1，故取符号位后与上!!x,排除 0 的情况。

第十一题:

```
int isLessOrEqual(int x, int y) {
int t = x >> 31;
int t1 = t & 1;
int r = y >> 31;
int r1 = r & 1;
int issame = !(t1 ^ r1);
return !(((y + ~x + 1) >> 31) & issame) & !(r & !t1);
}
```

是否 x<=y,当 xy 符号相同时，无需管溢出的情况，当符号不同时，可根据符号直接返回结果，故分成两块（一块计算 y-x 的符号值，另一块在二者符号不同时计算结果），将符号位判断是否相同的结果（即 issame 这个变量）与 y-x 的结果与，再非，即可保证两块分别工作，后边一块同理。

第十二题:

```
int ilog2(int x) {
int y = !!(x >> 16) << 4;
y = y + (!!(x >> 8 + y) << 3);
y = y + (!!(x >> 4 + y) << 2);
y = y + (!!(x >> 2 + y) << 1);
y = y + !!(x >> 1 + y);
return y;
}
```

求以 2 为底 x 的对数,即找到最大位数的 1，采用二分法，先将数分成两个平均的部分，用!!x 判断该部分是否为 0，再将该数左移 4，即下一次要将数分成的长度，若该部分为零，则说明较小部分的位数中有 1，下次需右移该次右移的一半，否则，仍需右移之前的长度，故要<<4,<<3,<<2，以此类推，求得最大位数的 1

第十三题:

```
unsigned float_neg(unsigned uf) {
if ((uf & 0x7fffffff) > 0x7f800000)
return uf;
return uf ^ 0x80000000;
}
```

取相反数，判断传进来的数是否为 NaN 即可（将符号位置 0，在判断是否大于 0x7f800000）。

第十四题:

```
unsigned float_i2f(int x) {
    int frac = 0;
    int exp = 0;
    int k = 1;
    int count = 0;
   int max = 0;
    int fu = 0x80000000 & x;
    int flag = 0;
```

```
    int tt = 0;
    if (fu) { x = -x; }
    if (!x)return 0;
    while (1)
    {
        if (k & x)max = count;
        if (k & 0x80000000)break;
        k = k << 1;
        count++;
    }
   tt = x << (32 - max);
    if ((tt & 0x1ff) > 0x100)
        flag = 1;
    else if ((tt & 0x3ff) == 0x300)
        flag = 1;
    frac = (tt >> 9) & 0x007fffff;
    exp = max + 127;
    return (fu | (exp << 23) | frac) + flag;
}
```

返回 int 的 float 形式，当 x 为 0 时，直接返回 0，当 x 不为 0 时，找到最大位数的 1（while 循环），得到阶码，在考虑小数位舍入的情况（最高位的 1 是否大于 24）即可。

第十五题:

```
unsigned float_twice(unsigned uf) {
    int exp = (uf & 0x7f800000) >> 23;
    int frac = uf & 0x007fffff;
    unsigned y = uf & 0x80000000;
    int j = exp + 1;
    if (uf == 0 || uf == 0x80000000)return uf;
    else if (j > 255)return uf;
    else if (j == 1)return y | (frac << 1);
    return y | (j << 23) | frac;
}
```

返回两倍的 uf，若 uf 时+0 或-0 或 NaN 或 inf，返回 uf（利用数本身与阶码（exp）进行判断），对于阶码为 0 的数，由于其偏置值为 1-Bias，故直接左移一位即可，对于规格数，阶码加一即可。