4. bitParity(x)

The key to this problem is recognizing that if we have c = a^b, then c has the same parity as number of bits in a plus number of bits in b. So if we were to write something like x = x ^ (x >> 16), this will ensure that the lower 16 bits of x has the same parity as x did originally. We can keep squeezing this parity into fewer and fewer bits, until we have squeezed it into one bit. We can then mask this with 1 and return the result. The mask is important because the higher bits will contain junk.

8. isPositive(x)

x is not negative if the most significant bit is 0. So we can use the mask 1 << 31 to check if this is true. This is how we get notNegative = x & (1<<31).

Now we need to ensure x is not 0. !!x will be 0 only when x is 0. Requiring both of these expressions to yield 1 will give us the desired result.

9. subOK(x, y)

We first recognize that if x and y have the same sign, we cannot have overflow. We can see whether x is non-negative with

xNotNeg = !(x>>31);

and similarly with any other signed int.

Supposing x and y do have different sign, we will know that overflow has occurred if the sign of our result is the same as the sign of y. For example, if x is positive and y is negative, overflow would cause the result to be negative. Similarly, if x is negative and y is positive, underflow would cause the result to be positive.

The result of our subtraction can be computed with

result = x + (~y + 1).

The sign can be found the same as before.

So now !(xNotNeg ^ yNotNeg) will return 1 if y and x have the same sign. rNotNeg ^ yNotNeg will return 1 if the result and y have a different sign. If either of these gives 1, then overflow did not occur. So we simply bitwise or these expressions and return.

10. howManyBits(x)

First, suppose we are looking only at positive values. Since we are representing x in two’s complement, we would need to find the position of the most significant 1 in the bit string, then add 1 because we would need a leading 0 for the sign bit. For example, 12 in decimal is 1010 in binary, but in two’s complement would need to be 01010, giving 5 bits.

For negative values, notice we are looking for the position of the most significant 0 then adding 1 for the leading 1 sign bit. For example, -5 in decimal is 1011 in two’s complement with 4 bits needed.

Suppose we were to develop an algorithm that would find the most significant 1 and add 1. This would also find the number of bits needed for negative values if we instead worked with ~x when x is negative. The difference between negative and positive number is the sign bit, so if we were to write

sign = x >> 31;

sign would be all 1’s if x is negative and all 0’s if x is positive. So if we say

x = (sign & ~x) | (~sign & x);

then x will be ~x when x is negative and x when x is positive. Notice that now right shifting will always be logical as the sign bit will never be 0.

Our algorithm will now execute something like binary search. Notice we can write

need16 = !!(x >> 16);

To determine if there are any 1’s in the top 16 bits of x. If so, we should shift x to the right by 16 bits. Otherwise, we should leave x alone. We can write the amount we want to shift by as

shift = need16 << 4;

Since need16 is 1 when x needs at least 16 bits and 0 otherwise, shift will be 16 if x has any 1’s in the top 16 bits and 0 otherwise. So now we just write

x = x >> shift;

to shift x. We also want to keep a running score of the number of bits needed, so we update needed variable with

needed += shift;

We continue in the same way for need8, need4, need2, need1, and need0 in the same way. We finally add 1 to needed for the extra sign bit and return.

12. float\_twice(uf)

We want to return any specialized case to return the original value, so we use the mask

expMask = 0x7F800000;

unsigned exp = uf & expMask;

If the exponent is all 1’s, we want to return the value, so

if(exp == expMask) will catch all specialized cases, and simply returns uf.

Else if (!exp) will catch all denormalized cases. In this case, we need to double the fraction field. We can erase the fraction field with

Uf & (~fracMask)

Then double fraction field with frac + frac. This takes care of the denormalized cases.

All other cases are normalized, and here we can add one to exponent with

Uf + 0x00800000;

13. trueFiveEighths(int x)

There are two keys to this problem. The first is in saving the remainder that will be lost when x is divided by 8 and regained when x is multiplied by 5. The other is that negative numbers will round to negative infinity. We want them to round toward 0, so we must correct in this case.

xDiv8 = x >> 3;

This expression divides x by 8.

Remainder = x & 0x7;

Remainder is used to save the lowest 3 bits which are lost when dividing by 8.

xDiv8Mult5 = (xDiv8 << 2) + xDiv8;

This multiplies xDiv8 by 5, not yet taking into account the remainder.

xIsNeg = x >> 31;

This will be all 1’s if x is negative, and all 0’s if x is positive.

remainderMult5 = (remainder << 2) + remainder;

Here we multiply remainder by 5.

carry = (remainderMult5 + (xIsNeg & 0x7)) >> 3;

Here we see that xIsNeg & 0x7 is 0 when x is positive and 111 when x is negative. This result is then added to remainderMult5, then the whole thing is divided by 8. Basically, as we shift the remainder right by 3, we are okay with any bits being lost if x is positive, but we want to round up if x is negative. Adding 111 to the bottom three bits in the case of x being negative before the shift ensures this happens.