(Note: this is written for norm floating point numbers. For denorm, the exponent will always be exp - bias + 1, and we do NOT add $2^{exp - bias + 1}$ in the mantissa.)

Floating point to decimal

Assuming 1 sign bit, 8 exponent bits, 23 mantissa bits, and a bias of -127.

Example: 0x70DA0000

1. **Convert to binary**

0x70DA0000 → **0 11100001 1011010000…**

2. **Find the exponent**

11100001 → 225, so 225 - 127 (bias) = **98**

3. **Convert mantissa**

Write out the mantissa bits with numbers underneath them. These numbers will start from the exponent we just calculated minus one, so 98 - 1 = 97. Like this:

1	0	1	1	0	1	0	0	…	0
97	96	95	94	93	92	91	90	…	75

Now, every bottom number that has a 1 on top will be included in our final answer:

2^98 + 2^97 + 2^95 + 2^94 + 2^92

Our original exponent of 2^98 will always be included in this summation as well (but NOT for denorm).

4. **Don't forget the sign bit**

If the sign bit is 1, don't forget to multiply by -1. In this case our sign bit is 0, so our number is positive, so we don't have to do anything here.

Final answer: 2^98 + 2^97 + 2^95 + 2^94 + 2^92

Decimal to floating point

Assuming 1 sign bit, 8 exponent bits, 23 mantissa bits, and a bias of -127.

Example: 9.75

1. **Find sign bit**

In this case, **0**, since we have a positive number.

2. **Write out number in exponents of 2**

9.75 = **2^3 + 2^0 + 2^-1 + 2^-2**

3. **Find exponent bits**

Our leading exponent will be the exponent we want to represent here. In this case, our leading exponent (the biggest one) is 3. So we do, 3 + 127 (bias) = 130 → **10000010**

4. **Find mantissa bits**

This process is similar to what we did in step 3 from converting floating point to decimal. Starting from the leading exponent minus one, so 3 - 1 = 2, if the exponent is present in our summation, we write a 1, otherwise we write a 0. Again, this is better seen in an example, so here it is. The bottom numbers are the exponents, the top are the mantissa bits.

Our original summation: **2^3 + 2^0 + 2^-1 + 2^-2**

0	0	1	1	1	0	0	…	0
2	1	0	-1	-2	-3	-4	…	-20

5. **Convert to hex**

0 10000010 001110000… → **0x411C0000**

Final answer: 0x411C0000

Intuition

So how does this all work? Well let's first forget about floating point for a sec.

If I asked you in a math class to do 100 * 1.305, you would probably just multiply them. But, if you wanted to, you could do something like

$$10^2 * (1 * 10^0 + 3 * 10^{-1} + 5 * 10^{-3}) = \mathbf{10^2 + 3 * 10^1 + 5 * 10^{-1}}$$
$$= 130.5$$

It's the same idea with floating point, but in this case thinking about it using the second method is easier (for me at least). Since we're in base 2 now, we replace the 10's with 2's.

So 1000 * $1.1001100\ldots_2$ in decimal would be

$$2^3 * (1 * 2^0 + 1 * 2^{-1} + 1 * 2^{-4} + 1 * 2^{-5}) = 2^3 + 1 * 2^2 + 1 * 2^{-1} + 1 * 2^{-2}$$
$$= \mathbf{2^3 + 2^2 + 2^{-1} + 2^{-2}}$$
$$= 12.75$$

Helpful Tips

So besides conversion, how is this useful? Well, for example, what if I asked you for the smallest integer unrepresentable by the standard 1 sign, 8 exponent, 23 mantissa bit floating point representation? You've probably already seen this problem, but this method gives an intuitive way to think about it.

Consider the example from the first page with **2^98 + 2^97 + 2^95 + 2^94 + 2^92**. Could I represent **2^98 + 1**? Nope, because **2^98 + 2^0** would be impossible to encode. (We know that to increase a floating point number by the smallest amount possible, we add 1 to the last bit, so…) Since there are only 23 mantissa bits, the lowest possible increase (by adding 1 to the last bit) would be **2^75**. (Because in this case, a '**1**' in the last bit corresponds to adding **2^75** to our answer!)

So to find the smallest integer unrepresentable, we just need to find when the last bit corresponds to adding **2^1** rather than **2^0**. (Adding **2^0** means adding **1**, which hits every integer, but adding **2^1** means adding **2**, which skips every other integer.) When the last bit corresponds to **2^1**, our exponent is **2^24**. Therefore, the smallest integer unrepresentable is **2^24 + 1**. (Try representing **2^24 + 1** using this method, and you'll see why it's impossible. If my explanations make no sense, I'm sorry, but pls try doing this out, I think it'll help a lot.)

All in all, hopefully the logic makes sense, and you can try applying it to other floating point questions. This is how I do every floating point question :)

good luck!