HW3.1. Floating Point Theory

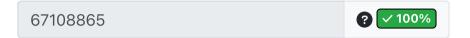
Feel free to check out the guide that we have prepared to help you in this problem.

For the following question, we will be following the IEEE-754 Floating Point standard *but with different numbers of bits:* with 6 bits of exponent, an exponent bias of -31, and 25 bits of mantissa. Due to PrairieLearn restrictions, please submit your answer as the full number.

Floating point representations present a trade-off between range and accuracy. Due to the limited significant digits that you can represent, there will be a point where integers cannot be exactly represented anymore.

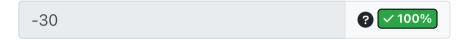
Q1.1: What's the smallest non-infinite positive integer (whole number) this representation CANNOT represent?

Hint: Think about how your floating point value changes when you change the mantissa by 1.



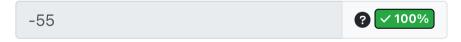
The floating point standard optimizes the representation of numbers by not including the implied 1 for the 'normal' binary representations.

Q1.2: What power of 2 is the smallest representable positive normalized number? Submit the exponent only.



Floating point also allows for representations of numbers even smaller than the smallest normalized number. Denormal numbers utilize an implied 0, instead of an implied 1.

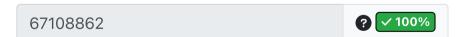
Q1.3: What power of 2 is the smallest representable positive value? Submit the exponent only.



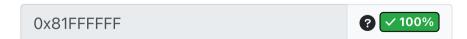
Floating point also has support for special values that can be returned as the result of invalid operations.

Q1.4: How many NaNs can be represented with this floating point system?

Hint: Recall how NaNs are represented following the floating point standard.

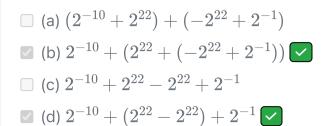


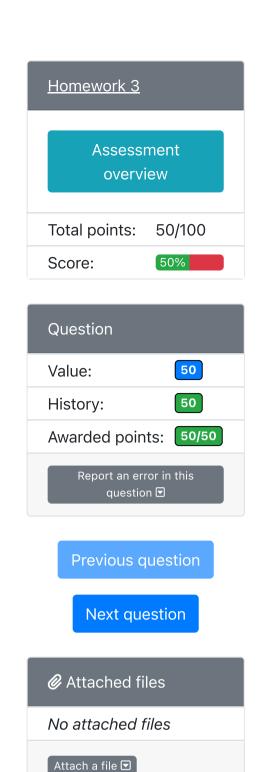
Q1.5: What's the most negative possible denormalized number representable in this system? Write the value in IEEE-754 hexadecimal notation (with uppercase letters), including the "0x" prefix.



The next few questions explore some quirkier aspects of floating-point numbers.

Q2.1: Addition and subtraction on unsigned and two's complement numbers is both commutative and associative. (You can arbitrarily re-order a series of additions and subtractions and get the same result.) In contrast, floating point addition is **not** associative, because between each substep, the result loses a small amount of precision. Given the floating point encoding from earlier, which of the following sequence of floating point additions and subtractions returns the correct result?





Attach text 모

100%

Q3.1: An IEEE-754 **single-precision** floating point number is 32 bits, consisting of 1 sign bit, 8 exponent bits, and 23 mantissa bits. Can this floating-point encoding represent the space of all integers a 32-bit, two's complement number can represent?

- (a) No
- (b) Yes

~ 100%

Try a new variant

Correct answer

67108865

Q1.1: Precision is limited by our mantissa, which has 25 bits. This means we can represent integers up to 2^{26} , which would be $2^{26}*1.0000\,$ 0000 0000 0000 0000 0000 0. $2^{26}+1$ would be $2^{26}*1.0000\,$ 0000 0000 0000 0000 01, requiring us to put a 1 in the non-existent 26th bit of the mantissa. So the smallest positive integer we CANNOT represent is $2^{26}+1=67108865$. (The next largest number we can represent after 2^{26} is $2^{26}+2$, which would be $2^{26}*1.0000\,$ 0000 0000 0000 0000 1).

-30

Q1.2: Exponent field: 000001

The number cannot be a denorm, so the exponent must be nonzero.

So exponent is 1 - 31 = -30

 2^{-30}

-55

Q1.3: The floating point values closest to 0 are denorms. Denorms in this representation have an implicit exponent of -30. We set the mantissa as small as possible to $00\dots01$, meaning we multiply $2^{-30}\cdot0.00\dots01_2$ to give us the smallest positive value.

$$= 2^{-30} \cdot 2^{-25}$$

$$=2^{-55}$$

67108862

Q1.4: There's $2^{25}-1$ possible positive NaNs (we subtract one for infinity), and there's $2^{25}-1$ possible negative NaNs (we subtract one for infinity again). Combined, that gives us $2^{26}-2$ possible NaN values, which equals 67108862.

0x81FFFFFF

Q1.5: The most negative denormalized number must have a sign bit of 1, an exponent value equaling 0, and all 1s to fill out all of the significand bits. This thus produces 0b 1 000000 111...1, which can be written in hex as 0x81FFFFFF.

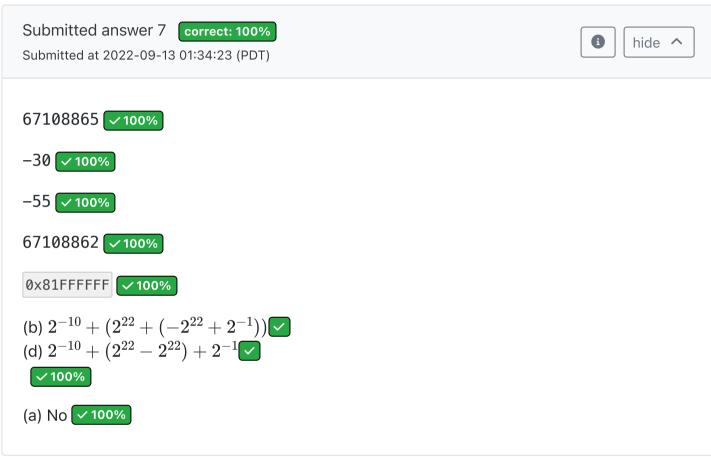
(b)
$$2^{-10} + (2^{22} + (-2^{22} + 2^{-1}))$$
 (d) $2^{-10} + (2^{22} - 2^{22}) + 2^{-1}$

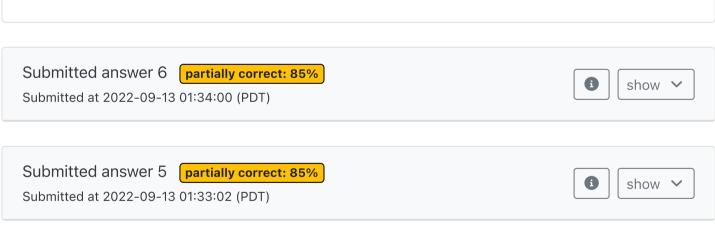
Q2: Unlike in the two's complement and unsigned representations we studied, adding a small magnitude floating-point number (e.g., 2^{-10}) to a significantly larger number (e.g., 2^{22}) will yield the larger magnitude number unchanged (e.g., $2^{-10} + 2^{22} = 2^{22}$). Since we have 25 bits of mantissa, our encoding correctly computes $-2^{22} + 2^{-1}$ and so we need not worry about doing the right-most addition first. (This would not be the case for representations

with fewer than 23 bits of significand). This is because in order to make sure 2^{-1} gets properly represented with an exponent value of 22, the signficand would have have all 0s and a 1 only in the 23rd bit since 22-(-1)=23. With fewer than 23 bits used, that -1 gets lost.

(a) No

Q3: It cannot. Here's one intuitive proof: both this floating-point and two's complement encoding can represent at most 2^{32} unique values, since they are 32 bits long. We know that the two's complement number represents exactly 2^{32} distinct integers. Thus, if there exists any floating point number that is **not** one of these integers, it cannot possibly represent all of the same values. There are many such values: oddities like NaNs, -0; integers that are larger than $2^{31}-1$; integers smaller than -2^{31} ; and of course, non-integer, but rational numbers.





Show/hide older submissions >