Ethan Wong

Discussion 1D

Professor Rich Korf

Assignment 1

I completed this assignment entirely on my own, except for discussions with Franklin Choi and Jiamin Xu.

1. There are 2147483648 positive integers, 0, and 2147483648 negative integers. When you add all these together there are **4,294,967,296** different integers that can be represented. This number is equivalent to 2^{32}, which makes sense because there are 32 bits used in two's complement.

2. The largest integer that can be represented by a 32-bit two's complement is **$2,147,483,647_{10}$** which can also be represented as **$01111111111111111111111111111111_2$**.

3. The smallest integer that can be represented by a 32-bit two's complement is **$-2,147,483,648_{10}$** which can also be represented as **$10000000000000000000000000000000_2$**. This is simply the negative of the largest possible number.

4. One billion is $2^{29} + 2^{28} + 2^{27} + 2^{25} + 2^{24} + 2^{23} + 2^{20} + 2^{19} + 2^{17} + 2^{15} + 2^{14} + 2^{11} + 2^{9}$
   One billion in binary = $0011\ 1011\ 1001\ 1010\ 1100\ 1010\ 0000\ 0000_2$
   Two billion is $2^{30} + 2^{29} + 2^{28} + 2^{26} + 2^{25} + 2^{24} + 2^{21} + 2^{20} + 2^{18} + 2^{16} + 2^{15} + 2^{12} + 2^{10}$
   Two billion in binary = $0111\ 0111\ 0011\ 0101\ 1001\ 0100\ 0000\ 0000_2$
   Adding these two numbers together, the binary result is:
   $0011\ 1011\ 1001\ 1010\ 1100\ 1010\ 0000\ 0000_2$
   $+0111\ 0111\ 0011\ 0101\ 1001\ 0100\ 0000\ 0000_2$
   $=1011\ 0010\ 1101\ 0000\ 0101\ 1110\ 0000\ 0000_2$
   This result, when translated back to decimal, is equal to **$-1,294,967,296_{10}$**. The reason it is not 3 billion is because of overflow. Also, the result is negative because the sign bit (the very first bit) is a 1. In a 32-bit representation there are simply not enough bits to represent the number 3 billion. There is a "0" that is missing in front of the result of our sum, which makes the decimal translation of the binary result appear negative. If we

were using a 33-bit representation (or any representation larger than 32) then we could accurately represent 3 billion. The significance of **-1,294,967,296$_{10}$** is that if you add the magnitude of this number to 3 billion, it equals the total number of possible different integers that can be represented. That is just demonstrating the overflow and wraparound principle when dealing with two's complement.

5. Negative one billion is $-(2^{31}) + 2^{30} + 2^{26} + 2^{22} + 2^{21} + 2^{18} + 2^{16} + 2^{13} + 2^{12} + 2^{10} + 2^{9}$
   Negative one billion in binary = 1100 0100 0110 0101 0011 0110 0000 0000
   Negative two billion is $-(2^{31}) + 2^{27} + 2^{23} + 2^{22} + 2^{19} + 2^{17} + 2^{14} + 2^{13} + 2^{11} + 2^{10}$
   Negative two billion in binary = 1000 1000 1100 1010 0110 1100 0000 0000
   Adding these two binary numbers together, the binary result is :
   ```
     1100 0100 0110 0101 0011 0110 0000 0000
    +1000 1000 1100 1010 0110 1100 0000 0000
    =0100 1101 0010 1111 1010 0010 0000 0000
   ```
   This result, when translated back to decimal, is equivalent to **1,294,967,296$_{10}$**. The reason it is not negative 3 billion is because of overflow. In a 32-bit representation there are simply not enough bits to represent the number negative 3 billion. There is a "1" that is missing in front of the result of our sum, which makes the decimal translation of the binary result appear positive. If we were using a 33-bit representation (or any representation larger than 32) then we could accurately represent negative 3 billion. The significance of **1,294,967,296$_{10}$** is that if you add the magnitude of the number to 3 billion, it equals the total number of possible different integers that can be represented. That is just demonstrating the overflow and wraparound principle when dealing with two's complement.

6. The amount of significant binary digits depends on the mantissa. In our 16-bit floating point representation, the mantissa is **9** bits. That means we will have 9 significant binary digits. Because there are 9 digits in the mantissa, it would translate to 2^9, which equals 512_{10}. Since 512 is three decimal digits, the significant decimal digits would be **3**. Another way to look at this is that the amount of significant decimal digits is equivalent to $\log_2$(# of bits in the mantissa). In this case, it would be $\log_2 9$, which is approximately 2.7_{10}. This will round upwards to 3.

7. There are **32,257** different numbers that can be represented using the 16-bit floating point representation. This conclusion was reached using this logic:
   - The mantissa has 9 bits. The first bit must always be a ‘1’ unless the entire mantissa is 0. Therefore, there are 2^8, or 256 different possibilities for what the mantissa’s magnitude can be. The mantissa can be negative or positive, which means that there are 512 different possibilities for what the mantissa can be.
   - The exponent has 5 bits. This means that there are $2^{5.}$, or 32 different possibilities for what the exponent’s magnitude can be. The exponent can be negative, positive, or 0, which means the total amount of different possible exponents is 63.
   - Multiplying 512 * 63 yields the amount of nonzero numbers that can be represented. If we add the representation of 0, that gives us a grand total of 32,257 different numbers that can be represented.

8. The largest number in 16-bit binary floating point is **0111111111011111_2**. This is because the mantissa is maxed out at its highest value of $1-2^{-9}$ and the exponent is maxed out at its highest value of 31. When we translate the mantissa into decimal, $1-2^{-9}$ is equivalent to $\frac{511}{512}$. This, when multiplied by 2^{31}, yields a decimal value of **$2{,}143{,}289{,}344_{10}$**.

9. The smallest number (most negative) in 16-bit floating point notation is **1111111111011111_2**. This is because the mantissa is maxed out at its highest magnitude of $1-2^{-9}$ and the exponent is maxed out at its highest value of 31. When we translate this negative mantissa into decimal, $-1-2^{-9}$ is equivalent to $-\frac{511}{512}$. This, when multiplied by 2^{31}, yields a decimal value of **$-2{,}143{,}289{,}344_{10}$**. This is the negative version of the largest number.

10. The non-zero number closest to 0 can be represented in 16 bit floating point as **0100000000111111_2.** The exponent is at its most negative value, -32. This results in 2^{-32}, which is approximately equal to **0.000000000233_{10}**. The negative of this number can also be considered the non-zero number closest to 0 as well because both the positive and negative of this number will be equal in terms of their absolute value.

11. I am interpreting consecutive as two binary numbers, one which directly precedes the other. I am interpreting the difference as the distance between the two numbers on the number line. The smallest difference between two consecutive numbers is, in decimal, **$9.094947 * 10^{-13}$**. The binary representation of this number is **$0.00000000000000000000000000000000000000001_2$.** This was determined by subtracting 0100000001111111_2 from 0100000000111111_2. This binary expression is roughly equivalent to $(0.5 * 2^{-31}) - (0.5 + 2^{-9})*2^{-31}$, which is where the **$9.094947 * 10^{-13}$** came from.

12. I am interpreting consecutive as two binary numbers, one which directly precedes the other. I am interpreting the difference as the distance between the two numbers on the number line. In this case, the largest difference between two consecutive binary numbers is **$4{,}194{,}304_{10}$**, or **0100000000010111_2**. This is the difference between the binary floats 0111111111011111_2 ($2{,}143{,}289{,}344_{10}$) and 0111111110011111_2 ($2{,}139{,}094{,}040_{10}$). These are the two largest numbers that can be written with this 16-bit representation. Because floats lose precision as they get larger, it makes sense that the largest difference between two consecutive numbers could be found by subtracting the second-largest possible number from the largest possible number.

13. The decimal 0.1 can not be represented properly using the 16-bit binary float representation. Attempting to convert it would look something like this:

	0.1	.
*2	0.2	0
*2	0.4	0
*2	0.8	0
*2	1.6	1
	0.6	
*2	1.2	1
	0.2	
*2	0.4	0
*2	0.8	0
*2	1.6	1
	0.6	
*2	1.2	1

It would be non-terminating if you tried to convert it: 0.00011001100_2…, or $0.0\overline{0011}_2$. In the floating point representation, this would be equivalent to $\mathbf{0110011001100011_2}$ as this representation is limited to 16 total bits (instead of having it repeat on forever). This is because the floating point representation can not represent every single value that the decimal system can. Operating on a base-two scale means that there is no way to accurately represent 0.1. Trying to convert this non-terminating binary number back into base-ten will not yield 0.1, but a decimal that is very slightly less than 0.1. This happens because translating the binary would yield something like $2^{-4}+2^{-5}+2^{-8}+2^{-9}+2^{-12}$, which results in $\mathbf{0.998535_{10}}$. It is impossible to have the binary representation translate exactly back into 0.1_{10} as no floating point representation, especially our 16-bit version, will be able to fully encapsulate the endlessly repeating part of the binary representation.

14. A simple rule for determining if a fraction can be represented by a terminating decimal expression is to check the denominator of the fraction. First, make sure that the fraction has been completely simplified. Find the prime factorization of the denominator. First, if the denominator is a prime number (excluding 2 and 5), then the decimal representation will not terminate. The decimal will also not terminate if the prime factorization contains any prime numbers (excluding 2 and 5). If you have a calculator on hand, you could also just type the fraction into the calculator and check if the decimal terminates.

15. A simple rule for determining when a fraction can be represented exactly by a terminating binary expression is seeing if the fraction is a power of two. If it is, then it can be represented as a terminating binary expression. Another way to determine this is to find the prime factorization of the completely simplified fraction's denominator. The prime factorization should only include 2's for the fraction to be capable of being represented as a terminating binary expression.

16. All terminating binary fractions can be represented in decimal. In base 10, literally any terminating fraction we can think of can be represented. Thinking about it logically, terminating binary fractions just represent a small subset of terminating base 10 fractions because binary fractions can't represent every fraction that we can conceive (like 0.1 for instance). This proves that every terminating binary fraction has a terminating decimal representation. Not all terminating decimal fractions will terminate in binary. For instance,

a simple decimal in base 10 would be 0.1. However, if you tried to represent this in binary, it would not terminate and become 0.0011001100110011001100110011…, as was explained previously in question #13.

17. The decimal 0.129 can not be represented with our 16-bit float with complete accuracy. The closest representation that can be achieved is 0100001000100010_2. This is because $(2^{-1}+2^{-6}) * 2^{-2} = .12890625_{10}$. I came to this conclusion by following the same process I used in question #13 (it's really space-consuming and clutters the work so I worked it out on scratch paper and chose to just simplify it to this).
The decimal 0.121 also can not be represented with our 16-bit float with complete accuracy. The closest representation that can be achieved is 0111101111100011_2. This is because $(1-2^{-5}) * 2^{-3} = 0.12109375_{10}$. I came to this conclusion by following the same process I used in question #13.

Subtracting the second number from the first yields:
  $1.1000010000 * 2^{-2}$
$+0.100001000 * 2^{-2}$
$= 0.0000100001 * 2^{-2} = 2^{-7}+2^{-11} =$ **0.00803008_{10}**

This is equal to **0100001000100110_2** in floating point.
This is equal to **0100010000100110_2** in floating point.

The decimal representation for 0.08_{10} can not be represented with our 16-bit float with complete accuracy either. The closest floating-point representation that can be achieved is **0100000110100110_2**. When translating this back into decimal, the result is **0.0079956_{10}**.
Again, the result is slightly off because of the fact that 0.008 can't be represented accurately in binary in the first place as 0.008 is non-terminating in binary.
After the binary subtraction, we end up with the result **0.00803008_{10},** which is slightly different from the expected answer of 0.008. This is also different from the translation of **0.0079956_{10}.** For the subtraction, there is a larger margin of error than in comparison of the direct translation of 0.008 to 16-bit float. This is most likely because of the lack of a perfect binary representation for 0.129 and 0.121 to begin with. As more processes were

executed on these numbers (the subtraction), the amount of error grew. The small amount of error in the initial translation is what threw off the actual answer from the expected answer.

18. 1.0_{10}, when translated into our 16-bit representation, is 0100000000000001_2
$.002_{10}$, when translated into our 16-bit representation is 0100000110101000_2

<u>First Method: (1.0+0.002) + 0.002</u>

$1.0 = .1 * 10^1$

$0.002 = .10000011 * 10^{-8} = .0000000010000011 * 10^1$

Adding (1+0.002):

$0.1 * 10^1$

$+0.0000000001000011 * 10^1$

$= 0.1000000001000011 * 10^1 = \mathbf{0100000000000001_2 = 1.0_{10}}$

By looking at this sum, we can see that the mantissa is going to be the same as the mantissa for our representation of 1_{10}. This happened because the 0.02 is too small in comparison to the 1.0 so it got lost once the two numbers were added together. This is a common occurrence when adding two numbers where one is many magnitudes larger than the other. Once we see this, it's clear that adding the second 0.002_{10} is not going to change the value. The end result will still just be $\mathbf{1.0_{10}}$.

<u>Second Method: (0.002 + 0.002) + 1.0</u>

$1.0 = .1 * 10^1$

$0.002 = .10000011 * 10^{-8} = .0000000010000011 * 10^1$

Adding (0.002+0.002):

$.0000000010000011 * 10^1$

$+.0000000010000011 * 10^1$

$= .0000000100000110 * 10^1$

Adding (1+(0.002 + 0.002)):

$0.1 * 10^1$

$+ .00000000100000110 * 10^1$

$=.10000000100000110 * 10^1$

This would translate to 0100000001000001_2, which translates to $(2^{-1}+2^{-9}) * 2^1$. This equals 1.0039. This number is remarkably close to the expected answer 1.004, but it is slightly off due the fact that the 9-bit mantissa wasn't able to hold all the values from the addition performed earlier. This worked out better than the first method because it is better to add the smaller numbers together first. This is because once you add the two 0.002_{10}'s together, their sum will be great enough so that it can get added to the 1.0_{10} without getting drowned out by the larger number.