Sam Dinkelman

11/17/2018

CIS 247

Lab Report 6

Introduction:

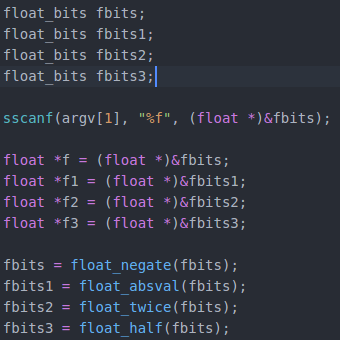
The intent of this lab was to get us to work arithmetically with floating point numbers in order to understand how they operate. We were tasked with completing five functions for this lab, one to return the class of the floating point number, one to negate the number, one to return the absolute value, one to double the value, and one to half the value.

Process:

In order to do this lab correctly we were given some code to properly implement floating point numbers. The code we were given defines a type of float\_bits to be unsigned.

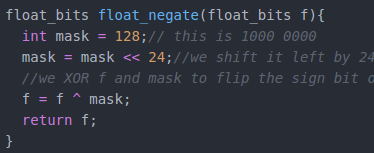


We were also given some of the functions and what parameters we pass to them in the lab description. Down in main we were also given some code for defining these types and calling functions as well as grabbing user input. Here are those lines:

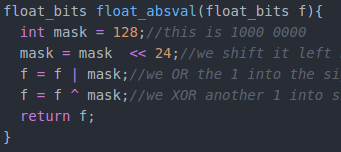


From this start we really only had to complete the functions and get them working with the values. This proved to be a big challenge for some of the functions while some were not too hard. The first two I started on were the negate and absolute value functions.

For my negate function I figured that creating a mask of 128 which in binary is 1000 0000, shifting it over so that the most significant bit(MSB) is aligned with the signed bit of the floating point number and XORing it into the value would be the easiest way to complete this. Here is that function:



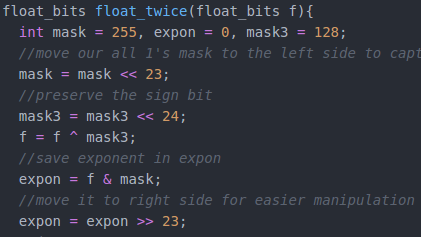
Next on my absolute value function I decided that doing a similar operation would achieve the results that I required. I started with the same mask of 128 and shifted it over to the signed bit in the floating point number. Then I OR the floating point value and the mask together, so if the floating point number is negative it will be positive and if its positive it will be negative. Then we XOR the floating points signed bit with the same mask so if the result of the previous operation turned the sign bit to 1 it will turn it to 0. This ensures that the signed bit always evaluates to positive since absolute value is just the distance from zero.



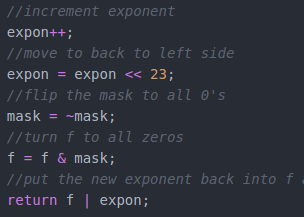
The next two functions I tried proved very difficult and I was stuck for a very long time on each of them. The doubling and halving functions proved very tricky. I was able to get my ideas down for each of them, but essentially when I would do certain value ranges my function would perform regularly and certain value ranges wouldn’t work. I was able to get one value range working at a time but I couldn’t get all the ranges to work together.

I ended up asking some classmates and tutors to help me with the math for these two functions. So while I wrote the code myself the algorithms to perform these operations were worked out with assistance from classmates and tutors.

So in my twice function I start by creating two masks and a blank integer for the exponent. The two masks are 255 which in binary is 1111 1111 and 128 which is 1000 0000 in binary. I then take my 255 mask and shift it to the left by 23 to align it over the exponent part of the floating point number. I also shift the 128 mask to the left 24 and XOR it with the floating point value to preserve the sign I found that if I didn’t do this I would receive negative values always. Then I take my blank exponent integer and save the exponent of the floating point number by ANDing it with the 255 mask. Then I shift this exponent number to the right by 23 for easier manipulation of the exponent.

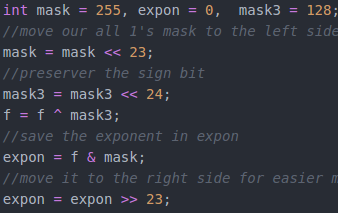


Next I increment the exponent and shift it back into place by shifting it left by 23. Then I take my mask and flip it so that my all 1’s mask becomes all 0’s. Then I take my floating point number and AND it with the mask of all 0’s to turn floating point into all zeroes. Then I take the floating point number and OR it with the exponent to return the new value of the floating point number.

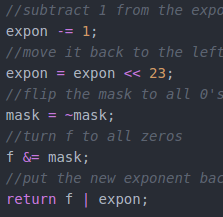


In my next function I also had a lot of difficulty with it and had to receive some help but once I had worked out the above function for doubling the value, I was able to complete this function by reversing only one step.

So in my halving function I start similarly to the doubling function by declaring some masks and a blank integer for the exponent. I create a mask for 255 which in binary is 1111 1111 and a mask for 128 which is 1000 0000. Then I do the same operation of shift the mask over to align it. I also do the same with the 128 mask to preserve the sign bit in my calculations. I then store the value of the floating point number into the exponent integer by ANDing it with the 255 mask. Which I then shift to the right by 23 in order to easily manipulate it.



Then this time instead of incrementing the exponent integer I subtract one from it and then shift it back to the right by 23. Then I flip the all 1’s mask to all 0’s in order to AND it with the floating point value f and turn f into all 0’s. Then I OR the floating point value and the exponent in order to store the value of the updated exponent into the floating point value.

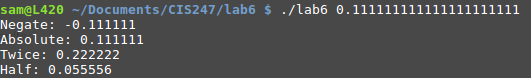


I didn’t really attempt class at all since I seemed very difficult and I ran out of time attempting the other two. I thought that class seemed very difficult without control statements and I couldn’t think of any ideas for performing the calculations.

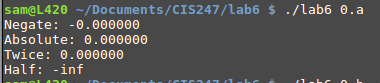
Testing:

When testing my code I took into account the fact that I didn’t have a class function so I was unable to test for that. Also I was unable to return NaN for any of the functions if they were indeed Not a Number.

Testing with really large mantisa’s truncated my print statements down since they are printed as floats.



Testing with letters instead of numbers gave me all 0’s for most of the functions and returned infinity for the half function.

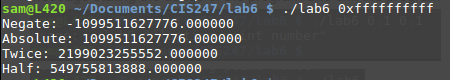


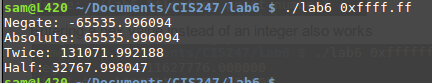
If I enter more than two or less than two arguments for my program it will throw an error message stating the usage of the program and exit the program



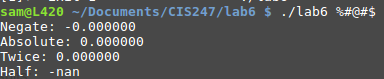


Entering a hex value instead of an integer also works



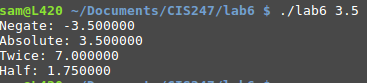


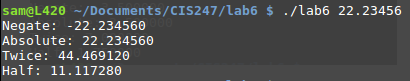
Entering characters again returns zero’s for all the functions except half which returns NaN



Results:

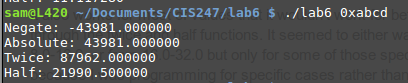
My results for the functions I was able to complete were very good and the functions themselves performed very well against almost all test cases.





It also works for most of values that it wouldn’t work for before in my first round of going through the twice and half functions. It seemed to either want to work for 2.0-8.0 or for 8.0-16.0 or from 16.0-32.0 but only for some of those specific ranges. This was likely because I was programming for specific cases rather than all cases as a CS mentor pointed out to me.

My results for hex are also valid although it will only print as a floating point integer not a floating point hex.



As you can see from my results my program accomplishes at least four of the functions required for the lab. I was unable to complete my class function and I was unable to return f for values that were NaN.

Conclusions:

Overall this lab was very difficult and made me dislike floating point number quite a bit. However I do now have a much greater grasp of their functions and limitations than I previously held. I found that manipulating the bits in the way that we did without control statements was good practice for test questions. This lab was very difficult but I felt like I have a more solid understanding of floating point numbers and how to work with them.

References and Acknowledgements:

I would like to acknowledge two classmates, John Traner and Conner Mangarelli for helping me with the math regarding the twice and half functions when I was stuck. I would also like to acknowledge the CS mentors who helped me at various points throughout this lab and offered guidance. I also used the site which I will list below very extensively and I found it to be very useful to visualize my operations.

https://www.h-schmidt.net/FloatConverter/IEEE754.html