1. **Introduction / Purpose / Intent**

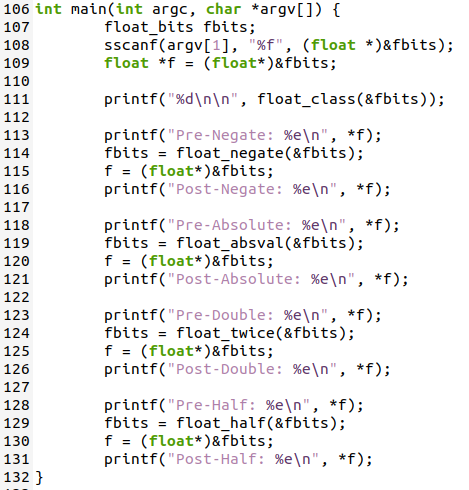
For this assignment I was tasked to complete five function definitions, which solve puzzles, using only a limited set of operators and write a main() function which calls each completed function and demonstrates its functionality. In this lab the functions implement floating-point operations, operating directly on bit-level representations of floating-point numbers and because of this the data type “float\_bits” is defined and provided as part of the lab.

During the lab I am expected to abide by a list of coding rules, the first of which is that each of the functions definition must be written so it contains all declared variables at the beginning of the function, followed by the operations on those variables, and finally the return statement. Within the definitions we are authorized only to use; integer constants 0-255 (0xFF), function arguments and local variables, unary operations NOT (!) and Compliment (~), and binary integer operations AND (&), XOR (^), OR (|), ADD (+), Shift Left (<<), and Shift Right (>>). I am not restricted to one operator per line.

For this lab I am expressly forbidden to utilize a list of items, to include; Global variables, large integer constants, any control constructs (if, do, while, for, switch, etc.), define or utilize any macros, define or utilize any additional functions in the file, call any functions other than to demonstrate their functionality, use any other operators (&&, ||, -, or ?:), utilize any form of casting, and utilize any data type other than and integer (arrays, structs, or unions). There are a few assumptions about the machine I have been instructed to use, including; utilization of 2’s complement, 32-bit representations of integers, right shift is performed arithmetically, and that it has unpredictable behavior when shifting an integer by more than a word size.

1. **Process**

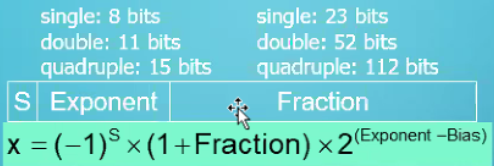
The first step in this lab was to isolate each of the larger tasks into smaller subtasks. To do so I copied each of the 5 function declarations into the provided lab6.c file, along with the English puzzle task as a comment. The 5 functions seemed fairly straight forward, except for the first which puzzled me greatly on how to accomplish without the use of control constructs. With this I decided to organize my main() with a printf() statement before and after every function call. This would allow me to test my program as I moved through each of the functions. Usually this is my method for troubleshooting and isolating issues, but it worked perfectly here as a method to prove functionality.

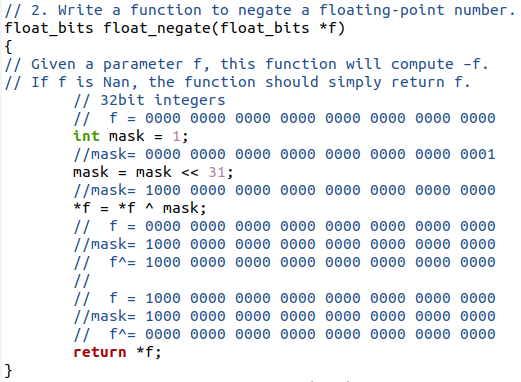


Because the first function seemed harder than the rest, and did little than preform a test of the input, I decided to start with the functions that preformed some sort of task to the input floating point value. These 4 functions all seemed related to another in pairs which would allow me to tackle group 1, which consisted of functions 2 and 3, before morning on to group 2, which consisted of functions 4 and 5.

* **Negate Value, if NAN return input**

I started with the negate function, and wrote what I would need to accomplish in pseudo-code prior to working on the function definition. I knew that in floating point, the 32nd bit is the bit that determines the sign of the float, so my pseudo-code was “flip the 32nd bit of input.” To do this I constructed a mask and assigned it the value of 1, so that only the first bit was on, and shifted it to the left 31 places so it was aligned with the sign bit. An XOR (^) could then be implemented to change the value, in both cases, in the provided value as can be seen below.

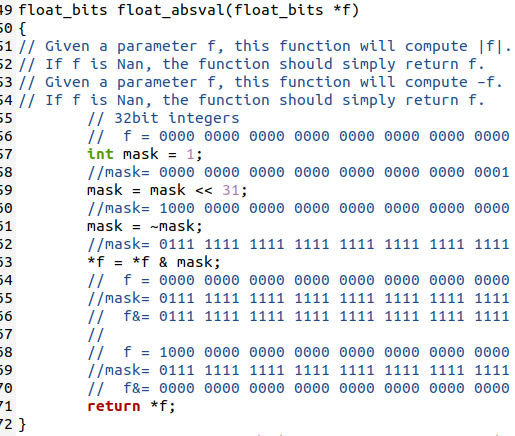






* **Absolute Value, if NAN return input**

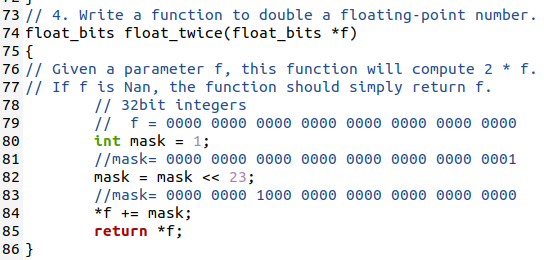
Following the last function, and part of the first group of related operations, the logic behind this puzzle was simple as well. Instead of indefinitely flipping the 32nd bit, the task would be to ensure that the bit was always off. To accomplish this I used the same logic and process as before, to place a bit in a mask and move it to the sign-bit position, but this time I used the NOT (~) of the mask. This would allow me to retain all of the bits throughout the rest of the number and eliminate the sign bit with an AND (&) as shown below.





* **Double Value, if NAN return input**

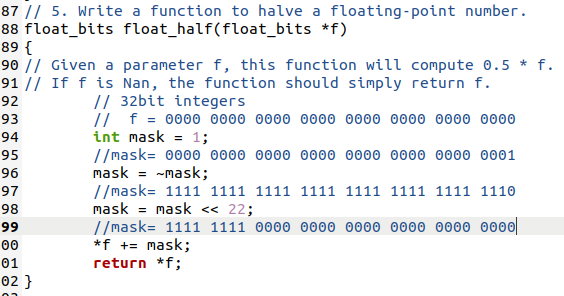
To begin the next group of functions, I realized I would be operating in the exponent portion of each of the floating point numbers, rather than with the sign bit. These bits are the 8 that precede the sign bit and represent an n value 0 – 255 for the equation 2^n as shown above. Because these bits represent the exponent of a 2, mathematically all that would be required for a double of the value is to increase the exponent by 1. I wrote this as my pseudo-code; “Add 1 to the exponent bits” and began to construct the function definition. Similarly to my other functions I generated a mask with a value of 1 and shifted it into place, at the least significant exponent bit. I added this mask to the input and returned the input which tested successful in doubling the value provided. In the special case of a NaN or Infinite where the exponent would be all 1s to begin, this would overflow into the sign bit and zero out the exponent.





* **Half Value, if NAN return input**

This function utilized the same logic as the prior in the second group, however due to the restriction of operators I cannot simply reduce the exponent with a subtraction of 1. To effectively produce this result, I begin the same mask as I had in the doubling function and NOT (~) it, placing a zero at the least significant bit of the float and all 1s elsewhere. This mask is then shifted into the least significant bit of the exponent, filling the entire fraction portion of the masks with 0s, and representing a -1 in the exponent field. By adding this value to the input, I successfully reduce the n value for 2^n by 1 and half whatever the input was for the function.

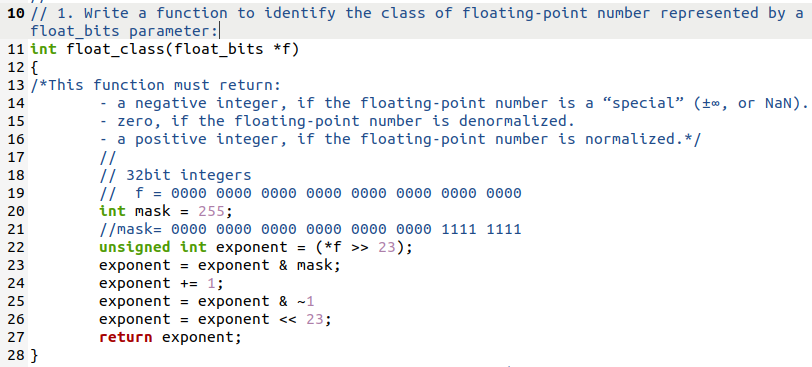


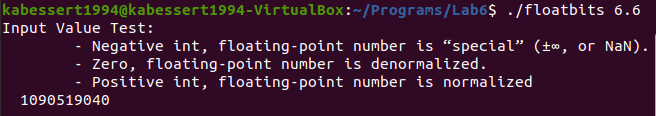


With this, the 4 functions that manipulated the input are completed, leaving only the first and most challenging function of the lab that tests the input values to go.

* **Float class**

I enlisted the help of multiple students who were stuck with the logic behind how to accomplish this task, whom are credited in the last section of this report, and began brainstorming. The logic derived from this was to shift the input value to the right so that the least significant bit of the exponent was in the least significant bit of the declared integer variable. A mask is generated with a value of 255, all 8 bits in the first octet are 1s, then AND (&) with the exponent variable to isolate the exponent bits inside the exponent variable, removing the sign bit from the original input. 1 is added to these bits so that in the event of a NaN or infinite as input, which has its entire exponent bits set to 1, would overflow into the sign bit. The exponent variable is then AND (&) with NOT (~) 1 to remove the bit that we just added in, but retain the result of the added bit. The result is then shifted back into place, filling the fraction portion with all 0s. Because we do not care about the actual value being manipulated, other than its sign, this is effective in producing the results we desire and will be further covered in the testing portion of this report.

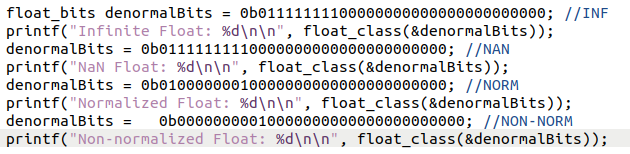




The completion of the first function marks the end of defining the functions for the lab, and with an operating main() that demonstrates each of the functioning definitions, I am ready to move onto the next phase of the lab which is testing. During this report I have not mentioned the returning of the input value for functions 2-5 in the case of a NaN, which is intentional and will be covered in a later section of this report.

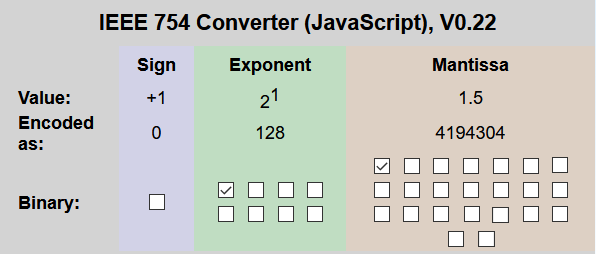
1. **Testing**

All throughout the coding process many “printf();” statements were used to output variables in various locations, such as in functions, before and after assignment statements, and within main() to ensure proper use of variables, expected behavior of the program, and manipulation of data. These statements used for troubleshooting have been shown in the screenshots provided up to this point of the report, but have been removed from the final revision of the submitted floatbits.c code that was submitted alongside this report.

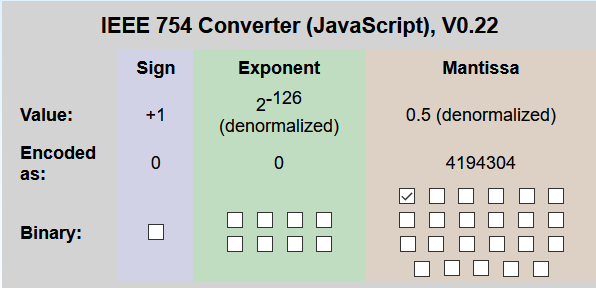


The above screenshot is also intended to demonstrate the method utilized for testing of the first function, float\_class(), by providing the function call values of known inputs. The first 2 cases supplied are the special cases of positive/negative infinity and Not a Number (NaN). The first is where the entire exponent field is filled and no data is in the fraction portion of the float, which indicates infinite, and the second is where the entire exponent field is filled and some data exists in the fract, which indicates a NaN. These 2 cases, as directed, should result in some negative integer when called by the float\_class() function, which our logic in the process portion of this report supports.

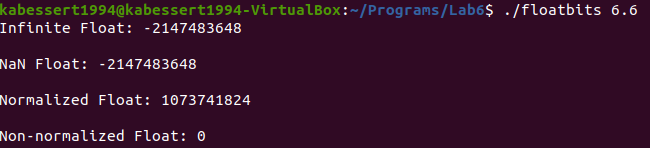
The third condition is a normalized float value, where it has a fraction value and has only a single digit at the left of the decimal point. I chose to store the number 3 by turning on the 128 bit in the exponent field, which when the bias is subtracted results in an n value of 1, and the .5 bit in the fraction portion. The .5 is added to the implicit 1 for the fraction portion and then multiplied by 2^n, n=1, resulting in a 3. Because only a single digit is to the right of the the decimal point for the value 3 it is normalized. I expect to receive a positive integer as a return from the float\_class() function, which our design supports.



Finally the last condition to test for the float\_class() function is that of a non-normalized number. I disabled the 128 bit in the exponent portion to produce a non-normalized 0.5, which is required to return a 0 and is supported by the design of the function in the above section of this report.



All values chosen and discussed here in this report tested successfully as intended and required. This proves that the logic and method behind the first function is solid, but requires further testing. Additional tests with varying values for all 4 cases were run, mostly focusing on the infinite and NaN conditions.



This additional focus is because, as discussed previously, of the inability of functions 2-5 to perform the required action of returning the original value supplied if that value is a NaN. The logic behind the functioning float\_class in this case is as follows;

Take any NaN or Inf

0 11111111 1000000000000000000000

>> 23

0 00000000 00000000000000011111111

+1

0 00000000 00000000000000100000000 // Add is carried out of exponent, into sign bit

& ~1

1 11111111 11111111111111111111110

0 00000000 00000000000000100000000

=

0 00000000 00000000000000100000000 // Removes added 1 in non-NaN values

<< 23

1 00000000 00000000000000000000000

The returned value is negative because of the observed overflowed in the addition step, and now that overflowed 1 is in the sign bit. I have tried to write pseudo-code to adapt my functions to perform this test but I am unsure how to implement it effectively while still performing the original operation. It was suggested to place the input value into a temp variable and conduct this test to see if the sign bit flips, however again I am unsure of what to do to combine that result with a working function. Theoretically, if the value in the temp variable did not overflow, leaving the sign bit off, and all other bits were removed, then that sign bit was added to the original input, no change would occur, unless the value was negative to start which would flip the sign. Then the operations of the negate function would return the same value for negative numbers and the opposite for positive numbers. I have reached out and attempted to work with other students to resolve this missing functionality, however no progress has been made. In the efforts of submitting this on time, I have chosen to keep my working functions as is for the time being.

1. **Results**

The results of this lab are a mixed, with a majority success and a single task failure. I have accomplished all that was required inside the lab assignment with the exception of returning the original input value, in functions 2 – 5, if the value supplied to the function was a NaN as discussed in the testing portion of this report. Aside from this all requirements have been met. I have completed the 5 function puzzles and written a main() which calls these completed definitions an displays their functionality. All provided code was utilized and implemented successfully.

I have ensured that I have written this program to abide by the full list of coding rules, declaring my variables at the beginning of functions and using my expressions on those variables underneath before utilizing a return statement. In these functions I ensured that I utilized only integer constants 0-255, function arguments and local variables, unary operations NOT (!) and Compliment (~), and binary integer operations AND (&), XOR (^), OR (|), ADD (+), Shift Left (<<), and Shift Right (>>). I took care to avoid the expressly forbidden items such as global variables, large integer constants, control constructs (if, do, while, for, switch, etc.), define or utilize any macros, define or utilize any additional functions in the program, call any functions other than to demonstrate their functionality, the use of any other operators (&&, ||, -, or ?:), the utilization of any form of casting, and utilizing any data type other than integer (arrays, structs, or unions). I also followed all assumptions of my machine while programing.

1. **Conclusions**

Based on the results and intent of this assignment I conclude that I have met the objectives of the assignment, which were; practice in bit manipulation of integers in C and enhance understanding of IEEE floating point format. Though the lab requirements were not fully met, and I could continue to further my knowledge with the floating point format, I would argue that my understanding of the IEEE floating point format has been enhanced from the beginning of this lab. I had previously no understanding of this format and to be able to have achieved what I have indicates some form of comprehension. The continued practice of bit manipulation of integers is present as well, which I demonstrate in every function definition.

1. **References / Acknowledgements**

C Programming Language, B. W. Kernighan & D. M. Ritchie, 2nd Edition, Prentice Hall, 1988.

C Programming: A Modern Approach, K.N. King, Norton, 2008.

[IEEE-754 Floating Point Converter](https://www.h-schmidt.net/FloatConverter/IEEE754.html)

[Bitwise Operators in C Programming](https://www.programiz.com/c-programming/bitwise-operators)

[6.4.2 Logical Negation](https://www.cs.auckland.ac.nz/references/unix/digital/AQTLTBTE/DOCU_061.HTM#log_negative_sec)

[Floating Point Numbers 0612TV](https://www.youtube.com/watch?v=gc1Nl3mmCuY)

[Single –Precision Floating-Point Format](https://en.wikipedia.org/wiki/Single-precision_floating-point_format)

Logan Lipke, Jared Larson, Nathan Burns, Keegan Giles, and Calvin Hewitt for;

* Various floating point aspects comprehension
* Assistance in generation of pseudo-code
* Assistance troubleshooting
* Logic checking
* Brainstorming group wide defects in understanding