Project Introduction

The project assigned to this group was the IEEE-754 Binary-128 floating point converter also known as quadruple-precision floating point. IEEE-754 represents numbers in 3 different sections: a sign bit for the sign of the number, an exponent to multiply to the mantissa, and the mantissa itself representing 1.x where x is the binary digits stored in the mantissa. The bias also is an additional number added to the exponent to allow storing negative exponents without needing an additional bit. For the case of Binary-128, it stores 64 bits with 1 for the sign-bit, 11 for the exponent, and 52 for the sign-bit. The bias in this version is 1023.

Two different versions of the input were allowed. The first would be an input of a Binary number and a corresponding exponent alongside it ($z_2 \times 2^y$). The second would be a Decimal number and exponent as well ($z_{10} \times 10^y$). The outputs would be the same either way, the resulting IEEE-754 representation in both full binary (split into the three sections) as well as its hex representation. An option to export by text should also be allowed according to the specifications.

Implementation

The project is divided into two steps: (1) conversion to a normalized binary number, and then (2) encoding into IEEE-754 format. The normalized binary number is stored as a combination of the mantissa and the exponent, with no constraints for both. For decimal (base-10) inputs, multiplying/dividing by 2 to obtain the normalized binary digits (as discussed in class) was used. For binary inputs, it is more straightforward as the input is essentially a non-normalized binary number, only needing normalization by "moving" the dot and adjusting the exponent.

The encoding process takes the unbounded mantissa-exponent combination and ensures that they fit within the constraints of the Binary-128 format, performing adjustments such as reducing the number of bits in the mantissa by rounding, conversion to subnormal representations, and conversion to infinity if necessary.

Validation

In order to validate the values found within the program, automated testing was performed with a limited set of decimal numbers, obtained from the list of all Binary-16 numbers that are not NaN/infinity, which can be easily calculated as there are only 63,488 of them. The script to accomplish this can be found in test_binary16.js within the tests folder.

Additional test cases were applied to test different kinds of values from normal values to the special cases. The special cases that were tested were the denormalized value, invalid inputs, zero and infinity.

Issues Encountered and Limitations

One major issue in developing the project was the lack of ways to check if the application's results were correct. Compared to single-precision and double-precision floating-point formats, computing with quadruple-precision floating-points is much more uncommon. Only a few specific programming languages have explicitly defined the implementation of a 128-bit floating-point, and often, only select compilers do so with some even treating the 128-bit floating-point data type the same as a double-precision floating-point. Other programming languages do not support quad-precision floating-points natively and only do so limitedly using external libraries and packages.

Fortunately, the group found two online JavaScript-based calculators that supported converting decimal values to its 128-bit floating point representation. They were primarily used for checking if the calculator's results were correct. The results were still checked manually, however, as the two calculators were also limited in their functionality in terms of the input formats they accepted and the special cases they handled. For example, one calculator did not implement the denormalized special case, opting instead to convert the values to 0. The same calculator also only outputted quiet NaN for all NaN values while the other did not handle NaN at all. The implementation the group used tried to closely follow the specifications defined in IEEE 754 and the conversion process used for single-precision and double-precision floating-point numbers that were discussed in class.

Furthermore, Binary128 can store numbers up to $1.19 \cdot 10^{4932}$ in value, while JavaScript's native numerical data types could only hold up to around 1.797^{308} which meant that either the conversion needed to be done by the string while the characters were analyzed one by one, or an external library would be used in order to help with the calculations. The group decided to move forward with using a library, one that could store the large numbers needed for Binary128. decimal.js is a JavaScript library that allows storing numbers far larger than capable, thus making it possible to calculate using values up to 4932 digits which would be needed.