CSARCH2
S13 - Group 6

IEEE-754 Binary-32 Floating-Point Converter
Analysis Writeup

Project Overview

In this simulation project, we implemented a Binary-32 Floating-Point Converter adhering to the IEEE-754 standard. We opted to implement the program in a web-based application so it can be easily deployed to the internet and become accessible without needing prior local installation. Our converter is designed to accurately process both binary (base-2) and decimal (base-10) inputs, including handling special cases such as Not a Number (NaN), denormalized, infinity, and zero. It outputs data in several formats: a spaced binary representation for clarity (to easily identify the sign bit, exponent representation, and mantissa representation), its hexadecimal equivalent for compatibility, and an option to download this information to a text file for documentation or further analysis.

Challenges and Solutions

1. **Simulating IEEE-754 Binary-32 Format Conversion**
   - Recreating the conversion process was a bit difficult since we were used to doing the conversion by solving them manually. The first thing we did was to break down the process into more straightforward steps that could be easily divided into coded functions. From there, we populated each function and tested them individually to ensure they worked properly.

2. **Selecting Appropriate Data Types for Variable Storage**
   - Since we are responsible for everything related to the project and how the converter executes, this also includes what data types we would use to implement the process. Although most of the operations we would use are generally suited for the numeric data types, other operations, such as slicing the data, are only appropriately handled by strings. This meant that for some of the parts of the code, we had to convert some of the numerical values into strings and vice versa.

3. **Deciding Between Using Existing Libraries or Creating Custom Radix Conversion Functions**
   - When converting inputs of different bases (in our case, decimal, binary, and hexadecimal), most people would generally use the ones provided by programming languages for radix conversions. But in some cases, other inputs

are perceived differently by the libraries than how we wanted them to be accepted, thus resulting in a difference in the outputs. On top of that, since a library handled the conversion, we had no way of checking where the process might have gone differently. To fix this, we implemented our own function that converts the radices so that we have a way of tracing them ourselves and be responsible for our own conversion.

4. **Library Functions Used to Parse/Convert Variable Data Types**
    - Since we had to convert the variables we used in the functions from strings to numeric types and vice versa, we also had to use library functions such as toString(), parseInt(), and parseFloat() to convert numeric data to String, parse a String object to become a variable of Integer data type, and parse a String object into a Float data type, respectively. With this, we encountered minor challenges along the way with using such functions because we noticed some discrepancies between the resulting data and the expected output. Hence, we had to meticulously trace how the input data was represented and processed in each function to identify where the problem was coming from. After that, we spotted the source of the problem and refactored the function to effectively call variables in the right data type they should be in.

5. **Implementation of Accepting and Validating the Input Data**
    - One challenge that we also had to consider was validating the user's inputs. Since the data type of the input field for the number was a String, we had to consider what could only be accepted as valid inputs before starting the conversion process. To solve this, we used a Regular Expression to check if the user would only input a String that could include a sign symbol and numbers valid for their corresponding bases. Hence, the user can opt to include signs with '+' or '-' at the start, and the implementation can easily flag the user if they are inputting numbers that are not within the digit range of the number system, e.g., numbers that are not 0 or 1 for base-2.