Binary-32 Floating Point Converter Analysis Writeup

Project Overview

In this simulation project, we developed a Binary-32 Floating Point Converter adhering to the IEEE-754 standard. Our converter is designed to accurately process both binary and decimal inputs, including handling of special cases such as Not a Number (NaN). It outputs data in several formats: a spaced binary representation for clarity, its hexadecimal equivalent for compatibility, and an option for exporting these representations to a text file for documentation or further analysis.

Challenges and Solutions

1. Simulating IEEE 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 simpler steps that could be easily divided into coded functions. From there we populated each function and tested them individually to make sure they worked properly.

2. Selecting Appropriate Data Types for Variable Storage

    - Since we are responsible for all things related to the project and how the converter functions, this also includes what data types we would use to code 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 handled properly 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 the conversion was handled by a library, 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 not only do we have a way of tracing them ourselves, but also be responsible for our own conversion.