CSARCH2
Simulation Project

Burias, Damuy, Tia, Yap
T3 - AY2324 - S12

IEEE-754 Binary-32 floating point translator

The objective of this simulation project is to create a translator that accepts an 8-digit hexadecimal or a 32-bit binary input and converts it to its corresponding decimal number. Additionally, the user should be able to choose whether the output is to be displayed in fixed point or floating point and have the option to copy and paste this output.

A Binary-32 floating point (also referred to as a single-precision binary floating point) consists of 32 bits–the first bit being the sign bit, the second to ninth being the bits representing the exponent, and the rest of the bits being the mantissa (which is the value to the right of the binary point, i.e., the fractional part of the number).

Converting this binary floating point number to a decimal consists of six steps:

1. Convert the 8-bit exponent to decimal.
2. Subtract the bias from the exponent (127 for single-precision).
3. Convert the mantissa to decimal.
4. Add the mantissa to 1 (to account for the leading 1 of a normalized Binary-32 input).
5. Append the original base (which is 2 for binary) and exponent.
6. Simplify and append the sign.

For example, consider the following input.

Sign	Exponent	Mantissa
1	10000000	11000000000000000000000

Table 1. Sample Binary-32 input

Converting the 8-bit exponent results to a decimal value of 128. Subtracting the bias results in an exponent value of 1.

Binary Value	Multiplier	Decimal Value
1	2^7	128
0	2^6	0
0	2^5	0
0	2^4	0
0	2^3	0
0	2^2	0
0	2^1	0
0	2^0	0
Actual Value		128
Actual Value - Bias (127)		1

Table 2. Exponent conversion

The same is done to the mantissa. However, since this value is the fractional part of the number, its multiplier is 2 raised to -1, then -2, and so on.

Binary Value	Multiplier	Decimal Value
1	2^{-1}	0.5
1	2^{-2}	0.25
0	2^{-3}	0
...	...	...
0	2^{-23}	0
Actual Value		0.75

Table 3. Mantissa conversion

Adding the leading 1 to the mantissa results to a value of 1.75. Appending the original base, exponent, and sign results to a value of -1.75×2^1, which simplifies to -3.5. This is the corresponding decimal number of the given input in Table 1.

The technologies used to develop the project were the following:

- ❖ React and JavaScript (JS) - The group opted to create a JavaScript web application utilizing React due to its compatibility with the necessary packages and features needed to implement the project (e.g. hosting through GitHub Pages, compatibility with Sonner Toast)
- ❖ TailwindCSS - A JS framework that simplifies styling.
- ❖ Sonner Toast - A JS toast package that allows for interactive and customizable toast notifications.
- ❖ GitHub Pages - GitHub's free web hosting service for static applications.

The group was able to implement this project with the following interface.

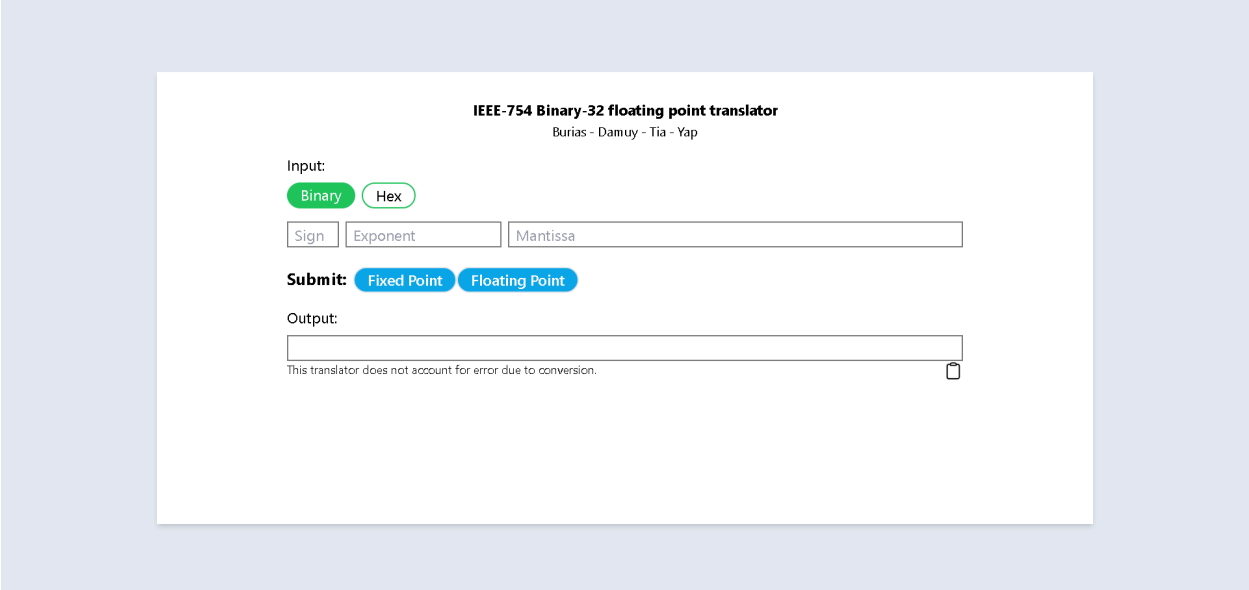

Figure 1. Interface of the converter

There is a toggle where users can select what type of input they intend to submit. To submit these inputs, they can select whether they want a fixed point and floating point and must simply click the corresponding button.

The conversion itself was implemented with 9 functions, which will be discussed one-by-one.

- ❖ *checkSpecialCases(sign, exp, mantissa)*

To account for special cases, the program simply checks each value for the sign, exponent, and mantissa. There are 6 special cases an input can fall into.

Sign	Exponent	Mantissa	Case	Description
0	11111111	00000000000000000000000	+Infinity	Positive Infinity

1	11111111	00000000000000000000000	-Infinity	Negative Infinity
0	00000000	00000000000000000000000	+0	Positive Zero
1	00000000	00000000000000000000000	-0	Negative 0
1/0	11111111	Anything other than 0…0	NaN	Not a number
1/0	00000000	Anything other than 0…0	Denormalized	No leading 1 before the binary point

Table 4. Special cases for Binary-32 floating points

This function accepts the inputted sign, exponent, and mantissa and simply compares it to each possible special case. It returns a string with the corresponding special case to be returned as an output and -1 if the input corresponds to no special case.

- *hexToBinNibble(hex) and hexToBin(hex)*

Since the project is expected to accept binary and hexadecimal inputs, hexadecimal inputs must first be converted to binary before it can be translated to decimal. *hexToBinNibble* is a lookup function that returns the corresponding binary value for each hexadecimal. This helper function is called in *hexToBin* where each character of the hexadecimal input is read and converted. It splits the converted binary value into its three parts: the sign, exponent, and mantissa. This is returned as an array.

Hexadecimal	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010

B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Table 5. Hexadecimal character decimal and binary equivalents

- *binToDec(bin)*

This function accepts a binary and returns a decimal. This is used for the first step of the actual conversion, where the exponent must be converted from binary to decimal. This is a helper function that is called in *getExp*.

- *getExp(bin)*

This function calls *binToDec* and subtracts the bias, as the second step of the conversion. It returns the exponent of the floating point number.

- *getMantissa(bin)*

This function does step 3 and 4 of the conversion process. It accepts the 23-bit mantissa and converts it to its decimal equivalent and adds the leading 1. This value is returned.

- *parseBinFloatFixed(sign, exp, mantissa)*

This function combines all the previous helper functions to do the actual conversion. It accepts the sign, exponent, and mantissa. It checks for special cases by calling *checkSpecialCases*. If a special case is returned, that is returned as the output. If not, it proceeds with the conversion. It calls *getExp* to get the actual exponent, and *getMantissa* to get the decimal value of the mantissa plus the leading 1. To get the magnitude of the number, this is simplified by multiplying the current value of the mantissa to 2 raised to the exponent. In the return statement, the sign is checked and the appropriate sign is appended to the value being returned. This is the final output for a fixed point value.

- *parseBinFloatFloating(sign, exp, mantissa)*

To add functionality to the floating point option for the project, this function simply calls *parseBinFloatFixed* with the JavaScript function *toExponential* appended to convert it into floating point notation. The “e” in this output is replaced with “*10^” to specify the base and for better readability. This

- *parseHexFloat(input, isFloating)*

This function accepts the hexadecimal input and a Boolean value which tells if the expected output is in floating point or not. It converts the input to binary with *hexToBin* then calls the appropriate functions–*parseBinFloatFixed* if the expected output is fixed point, and *parseBinFloatFloating* if the expected output is floating point.

The following test cases were used to check the functionalities of the project.

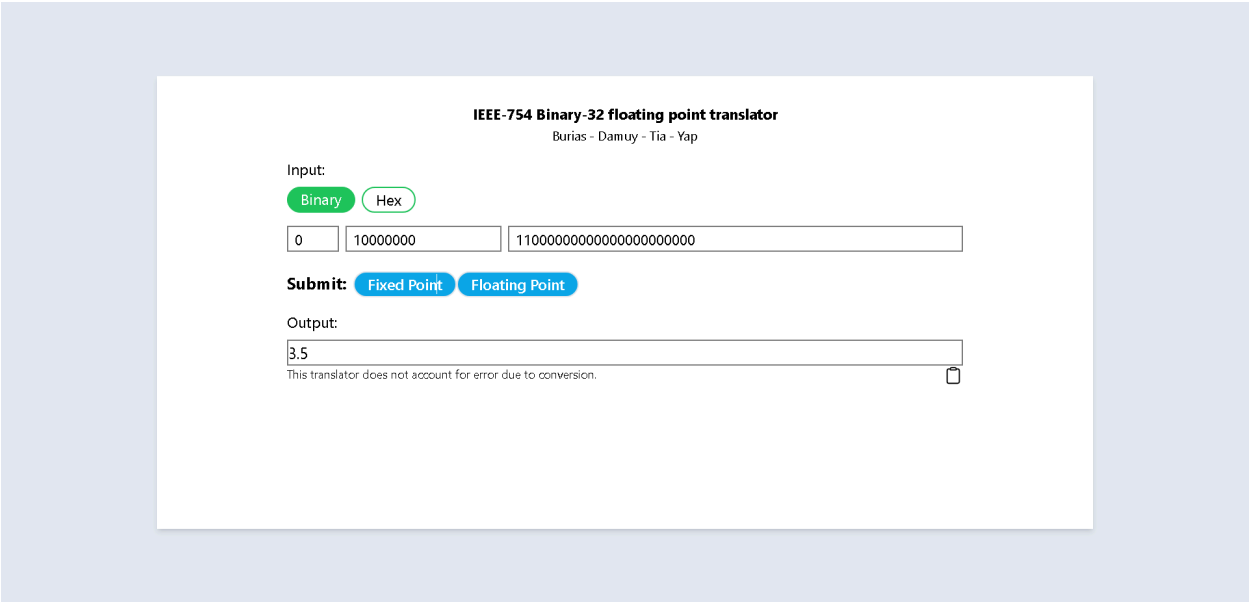

Figure 2. Binary input to fixed point output

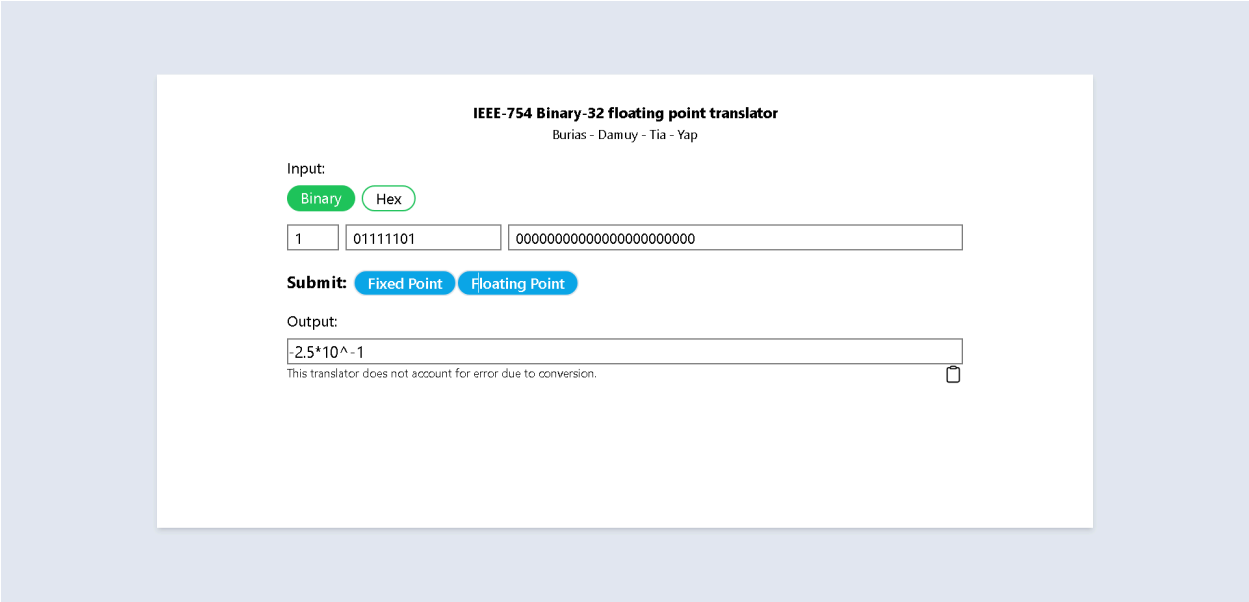

Figure 3. Binary input to floating point output

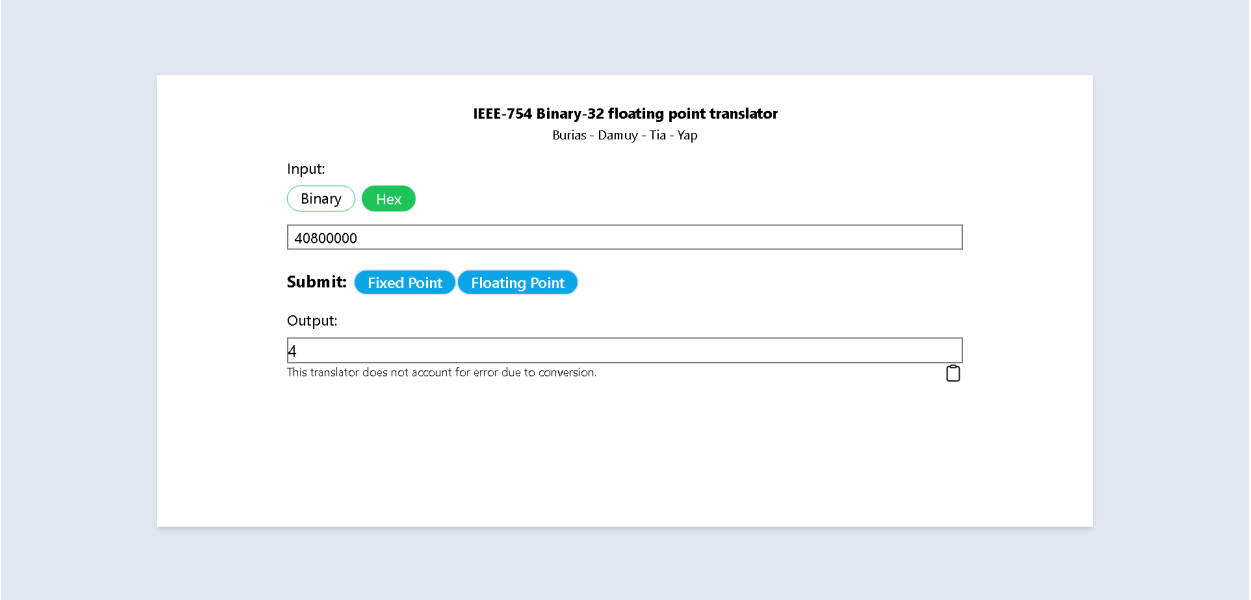

Figure 4. Hexadecimal input to fixed point output

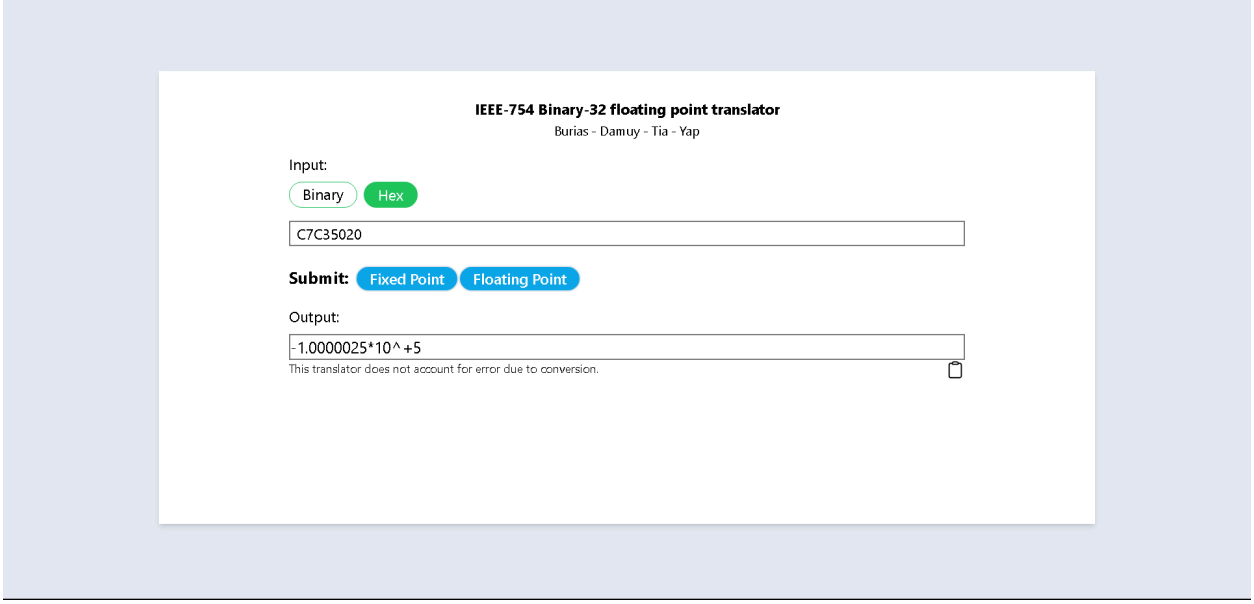

Figure 5. Hexadecimal input to floating point output

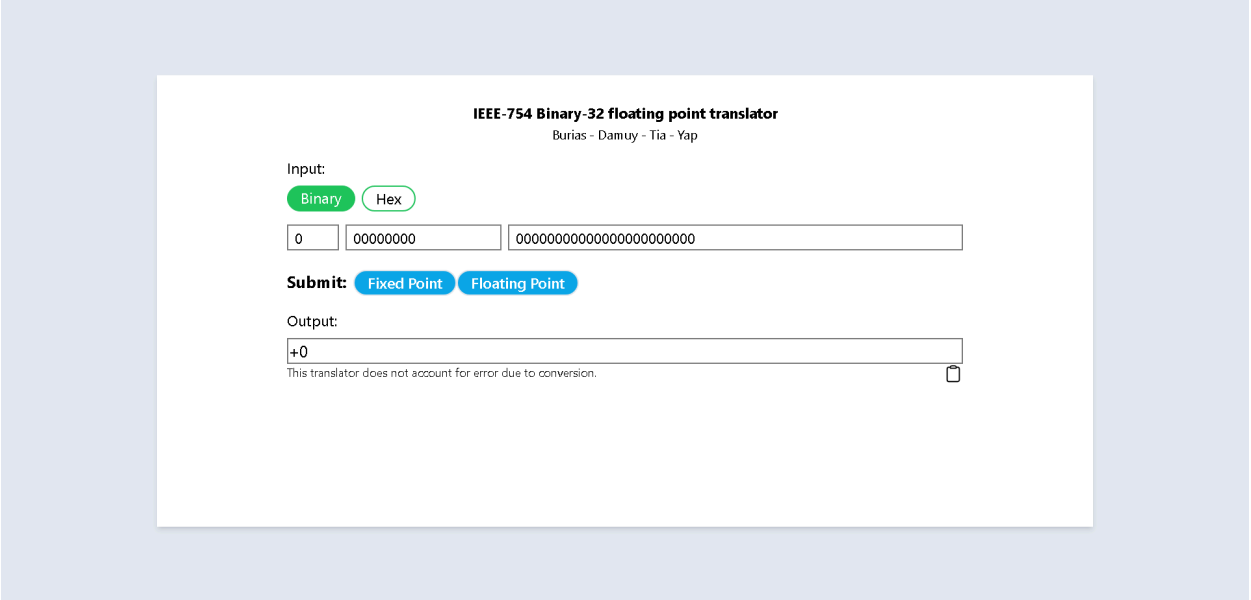

Figure 6. Special case positive zero

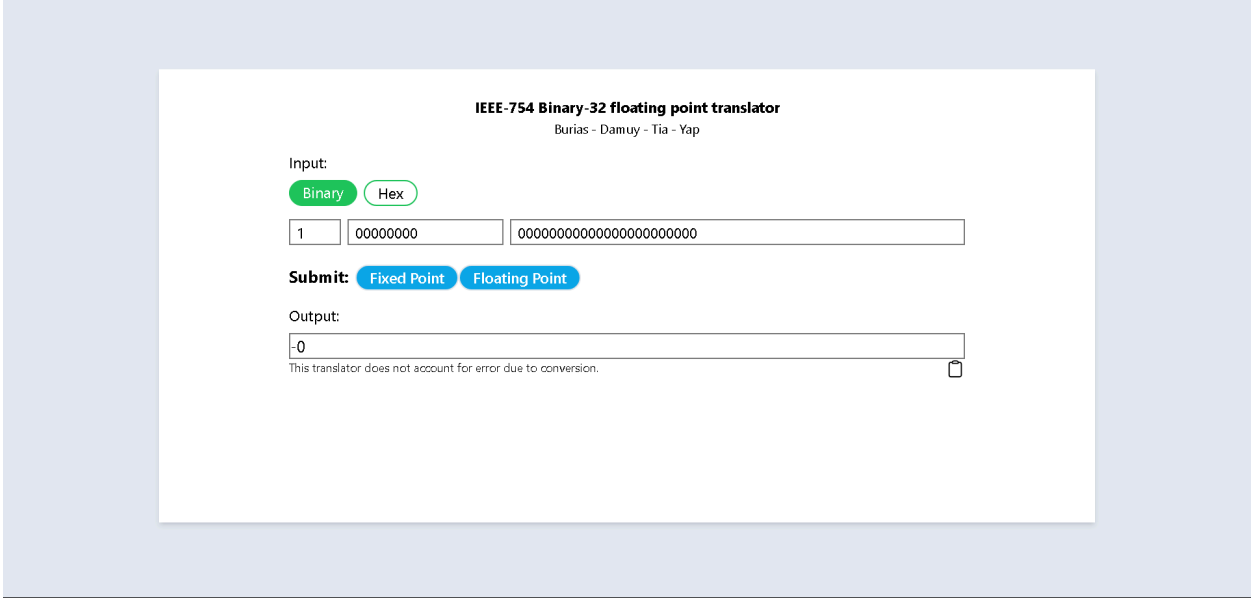

Figure 7. Special case negative zero

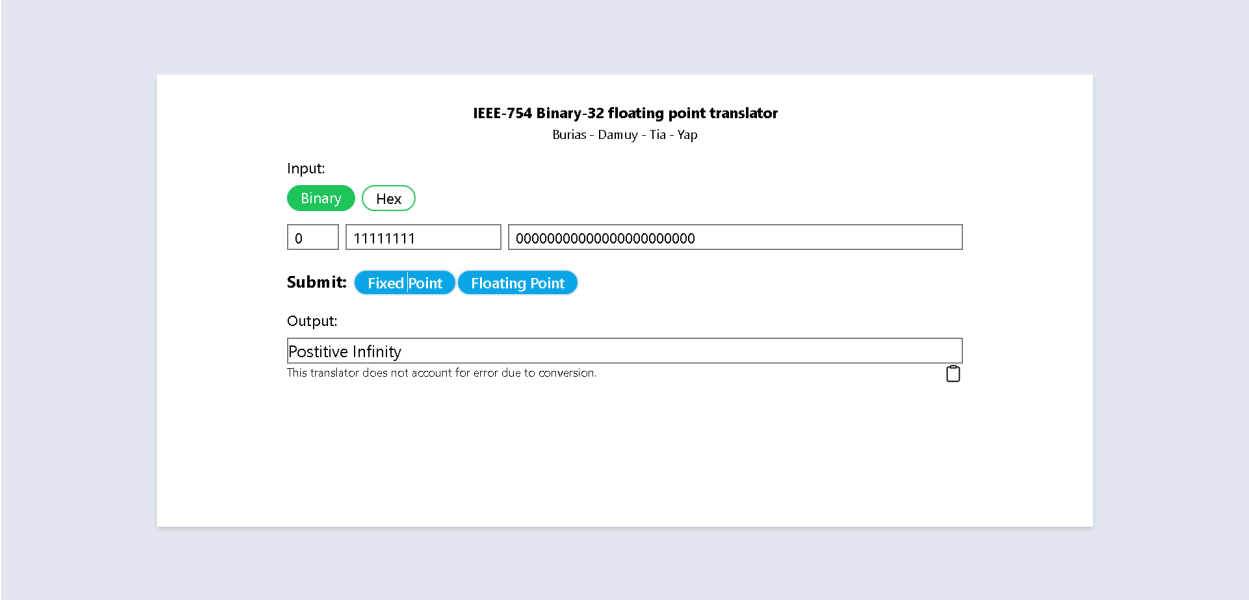

Figure 8. Special case positive infinity

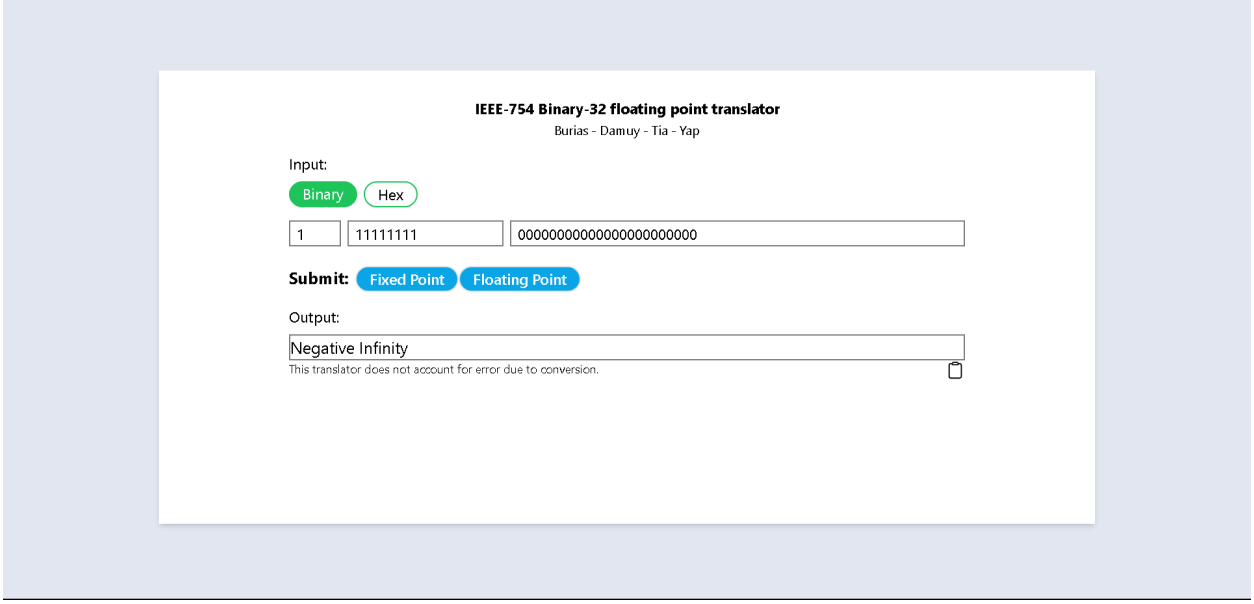

Figure 9. Special case negative infinity

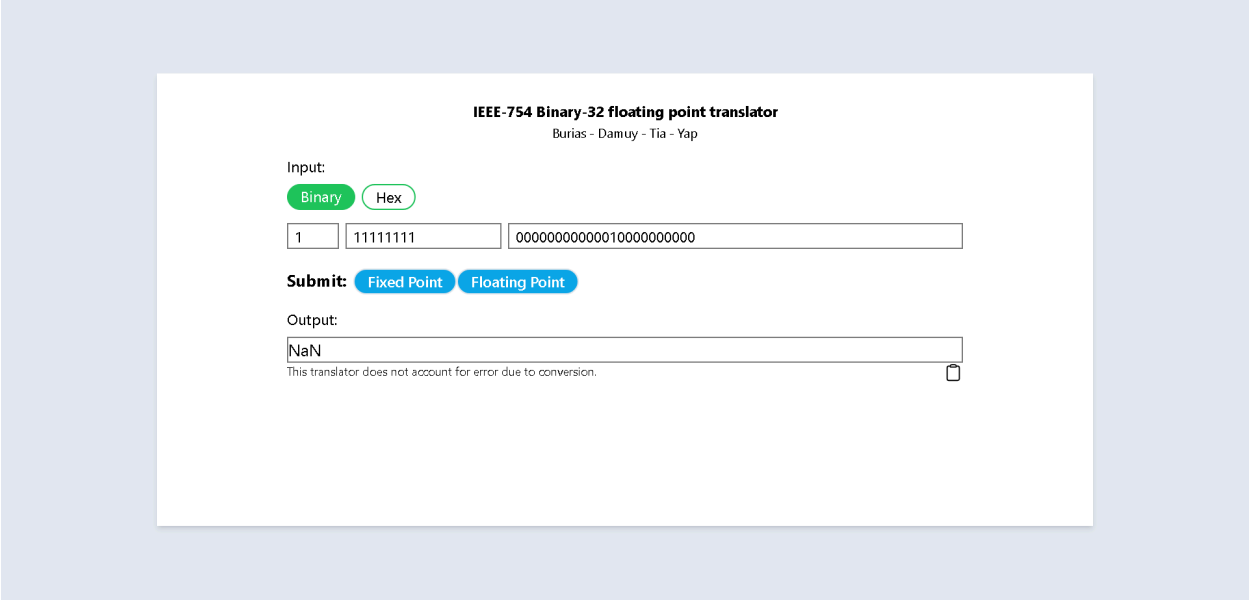

Figure 10. Special case not a number

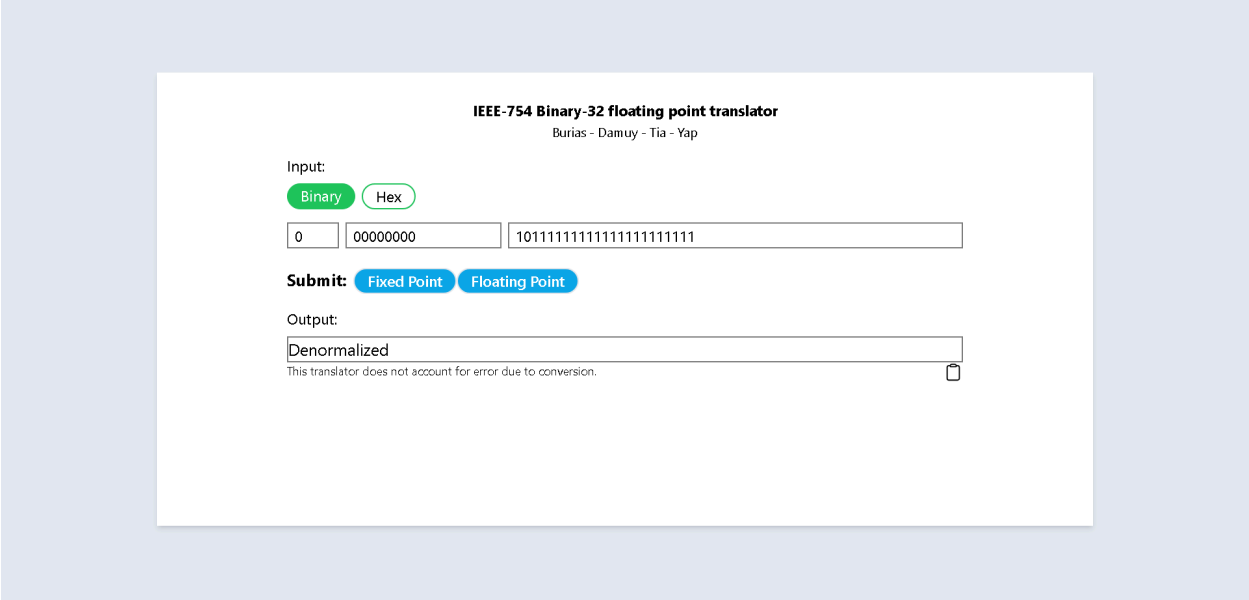

Figure 11. Special case denormalized input

Based on these test cases, the translator works as expected.