The assigned problem to our group was IEEE-754 binary-32 floating point operation, which involves adding two binary-32 floating point numbers and displaying all input/output and the computation process via a GUI. To meet the GUI requirement, we decided to go with a web application using JavaScript and Vue.js. As for the operation itself, we modeled our calculation from the following steps:

1.) Take the user input
2.) Normalize the input by aligning the exponents
3.) Round to the supported number of digits
4.) Addition proper
5.) Normalize the result
6.) Round the result to the supported number of digits
7.) Output the result.

Described below are the functions we created to achieve each step:

After the user inputs the two operands, they are sent to the alignExponent() function to normalize the two operands of the input and to ensure their exponents are equal.

alignExponent():
This function takes as input the two operands needed for the computation, normalizes them, and returns as object the normalized operands.

- Since the whole number part of the magnitude is not passed to the function, it is assumed that the whole number of both operands are equal to '1'.
- The difference is then calculated by taking the absolute difference of the two exponents.
- The main operations are done depending on which operand has the smaller exponent:
    - Its mantissa is shifted to the right. The offset of the shift is based on the value of the difference that is computed earlier. The actual shifting is handled by the shift() function, which takes as input the mantissa part of the magnitude, the offset of the shift, the length of the mantissa, and the whole number part of the magnitude.
    - Its exponent is made equal to the exponent of the other operand.
    - Its whole number part of the magnitude is set to 0, as it is assumed that the whole number part of the magnitude will always be equal to 0 when shifted.
- The magnitudes of both operands are calculated by doing a simple string concatenation of the whole number part, the decimal/radix point, and the mantissa part.

After the initial normalization, the operands are sent to either useGRS() or RoundRTNTE() depending on if the user opts to use guard/round/sticky bits or not.

useGRS(): If the user opts to use G/R/S, this function processes the normalized operands to find the guard, round, and sticky bits, in relation to the number of binary digits supported selected by the user.

- The operand length must be at least digits supported + 3 (G/R/S bits) to ensure the operation supports the selected number of digits.

- If the operand length is less than the number of supported digits + 3 (G/R/S bits), zeroes are appended to compensate.
- If the operand length equals or exceeds the number of supported digits + 3 (G/R/S bits), the guard and round bits are copied. The sticky bit remains 0 until it encounters a one from the remaining bits, after which it's set to 1.

The operands with G/R/S bits are then sent to addOperands() for addition proper.

RoundRTNTE(): If the user opts not to use G/R/S, then the normalized operands go through this function to be rounded (using round to nearest, ties to even) to the number of binary digits supported selected by the user.

- If the inputted operand has less binary digits than the amount supported, it gets padded with 0s until the length matches
- If the operand has the same number of digits as the amount supported, it is unchanged (no rounding necessary)
- If the operand has more digits than is supported, then rounding occurs:
    - For rounding, the function breaks the operand into multiple substrings; the "supported" portion and "extra" portion (which is used to round). The extra portion is itself broken down into the MSb and the remaining digits.
    - If the MSb of the "extra" portion is 0, it means either do nothing (ex. 000) or round down (ex. 010). In this case, the supported portion is left as is.
    - If the MSb is 1 and remaining digits include a 1 anywhere, then it is round up. The first 0 starting from the LSb of the supported portion is located, changed to 1, and all succeeding digits are changed to 0.
    - If the MSb is 1 and remaining digits are all 0, then it ties to even. If the LSb of the supported portion is 0, then the whole portion remains unchanged. If not, it performs the round up.

The rounded operands from RoundRTNTE() are then sent to addOperands() for addition proper.

addOperands(): The function adds the normalized operands, considering the chosen rounding method.

- The binary floating-point values are converted to decimal in preparation for the operation
    - If both operands have the same sign, they are added together, keeping the sign
    - If the operands have different signs, the smaller number is subtracted from the larger value, and the sign of the more significant number is kept
    - The resulting sum is then converted back to binary floating-point value

Once the sum has been obtained, it is sent to normalizeSum().

normalizeSum(): The sum from addOperands() then goes through here to be normalized.

- Calculates the offset needed to convert into the format 1.xxx
- Converts the sum into the proper format
- Adjusts the exponent by the amount offset

Once normalized, the sum goes through RoundRTNTE() to ensure that it conforms to the maximum number of digits supported. Once this has been completed, we have obtained the final answer which is displayed on the GUI.