Presented to the College of Computer Studies

De La Salle University - Manila

Term 2, A.Y. 2023-2024

In partial fulfillment of the course

CSARCH2 S14

IEEE-754 Binary-16 floating-point converter

Group No. 5

Submitted by:

Dela Cruz, Frances Julianne

Gamboa, Mikkel Dominic

Muriel, Gabriel

Sang, Nathan Immanuel

Yu, Hanz Patrick

Submitted to:

Sir Roger Uy

March 23, 2024

IEEE-754 Binary-16 floating-point converter

In Half-precision floating point representation, also known as binary16 format, here the team breaks down each component of its sign, exponent, and mantissa allocated bits

1. Sign Bit (S):
    a. ***1 bit*** represents the sign of the number, indicating whether it's positive or negative.
2. Exponent (E):
    a. ***5 bits*** are used for the exponent. By allocating bits to the exponent, the range of representable exponents becomes negative 14 (-14) to positive 15 (+15) when biased since its offset by bias of 15 allows for both positive and negative exponents.
3. Mantissa (M):
    a. ***10 bits*** are used for the mantissa as it holds the significant digits of the number and determines the precision of the floating-point representation.

Range of Numbers

Largest Positive and Negative Numbers:

- The largest positive and negative numbers in half-precision format are represented as $1.1111111111\ x\ 2^{15}$ and $-\ 1.1111111111\ x\ 2^{15}$ respectively. These values approximate $6.5504\ x\ 10^{4}$ and $-\ 6.5504\ x\ 10^{4}$ respectively.

Smallest Positive and Negative Numbers:

- The smallest positive and negative numbers in half-precision format are represented as $1.0\ x\ 2^{-14}$ and $-\ 1.0\ x\ 2^{-14}$ respectively. These values approximate $6.103515625\ x\ 10^{-5}$ and $-\ 6.103515625\ x\ 10^{-5}$ respectively.

Special cases

1. Greater than 2^{15}
    a. In half precision format, the maximum representable exponent is 2^{15} with a bias of 15. Any number greater than this value results in overflow. In this case the exponent would be filled with the maximum representable value in binary "11111". The mantissa would be filled with all 0s, representing the maximum possible value within the constraints of the format which is 10 bits for the mantissa. The sign bit could either be 0 or 1, depending whether the number is positive or negative. However, since the number is

effectively representing infinity, the sign bit would typically be 0 for positive infinity or 1 for negative infinity.

Example:
Sign bit: 0
Exponent: 11111
Mantissa: 0000000000

2. Less than 2^{-14}
   a. In this case, the exponent would be filled with the minimum representable value in binary “000000”. The mantissa would be filled with all 0s, as the precision of the format does not allow for representing values less than 2^{-14}. The sign bit could either be 0 or 1, depending on whether the number is positive or negative. Since the number is effectively representing zero, the sign bit would typically be 0.

      Example:
      Sign bit: 0
      Exponent: 00000
      Mantissa: 0000000000
3. NaN values
   a. In this case NaN (Not a Number) represents an undefined or unrepresentable value resulting from an operation such as 0/0 or ∞ - ∞. The sign bit can either be 0 or 1, However it does not carry any significance in determining the type of NaN. The exponent would be filled with 1s “11111” indicating that the exponent field does not represent a valid exponent value. The mantissa should have at least one bit set to 1 to differentiate NaN from infinity. The remaining bits can be arbitrary.

      Example:
      Sign bit: 0 or 1
      Exponent: 11111
      Mantissa: 1xxxxxxxxx

Application

To use our program, open the JAR file of our half-precision binary converter, as this would run the application. The allowed inputs for the JAR file have to be in the format of valuexmultiplier^exponent. For example, 2.0x10^5 or 0.110x2^-5. Values specified must only be positive or negative and whole or decimal in binary or decimal format. The multiplier must be 10 if using decimal and 2 if using binary format. Exponents could also either be positive or negative.

Problems Encountered

A. Imprecise Values Causing Errors in Calculations

   In the initial versions of the code, most computations used String and Float data types. Usually, type-casting and parsing were used to switch between the two. It worked for smaller values, but eventually, errors arose when it came to more complex ones.

   An error occurred when it came to using Float/Double data types for the computations. These data types are less precise and cause mismatches in actual expected outputs. For example, when dealing with the decimal portion of a number, it sometimes adds a 1 at the end. This was fixed by refactoring the code to use BigDecimal for the computations. Additionally, the process for computations was also adjusted to accommodate BigDecimal.

   Then, another error occurred when converting the BigDecimal value into a string. When converting using .toString() method, the string value output is using scientific notation. It required some tracing to realize this error, and we found that the BigDecimal class offered another method called .toPlainString(). This function allowed us to get a string output without truncating the values into scientific notation.

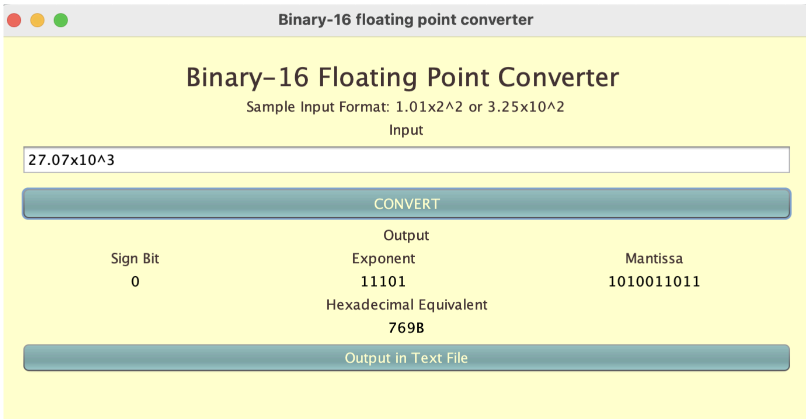
B. Incorrect Input Values Causing Errors for the Program

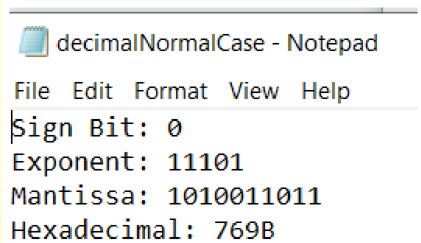
   Overall, the program works for all intended inputs. However, when it comes to inputs that the converter shouldn't accept, the program crashes. This was fixed by using a try-catch, which would result in an error message appearing in the GUI when an invalid input is entered. Moreover, the output text file would show blank values when an invalid input is entered. To fix this error, the Output in Text File Button is disabled when invalid inputs are entered. It is then enabled again once valid inputs are entered.

   The program was implemented to accept "sNaN" or "qNaN" strings as a replacement to accommodate NaN values. It would then output the corresponding values. However, since some values for this are represented by an "X" or random bit value, the hexadecimal representation outputs a range of values. Specifically, showing the least possible hexadecimal value and the greatest possible hexadecimal value.

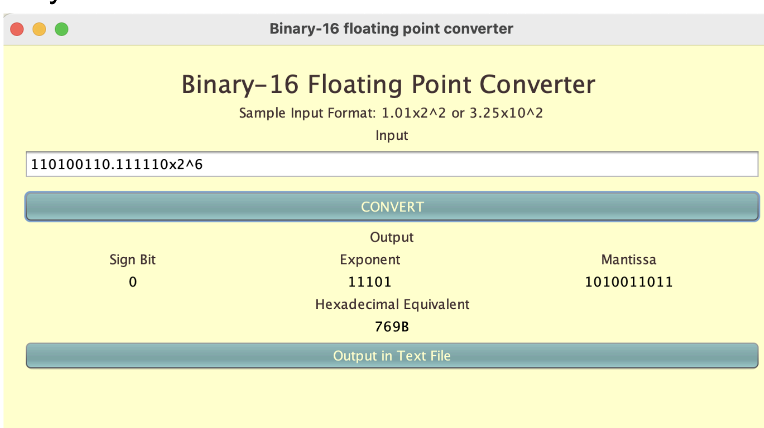
Sample Outputs

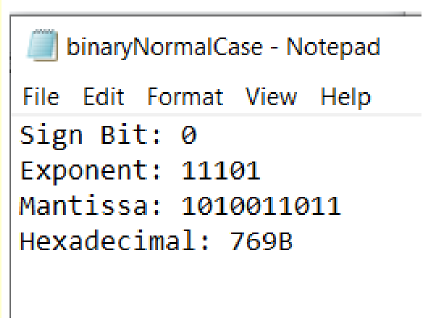
Decimal Normal Case: 27.07x10^3



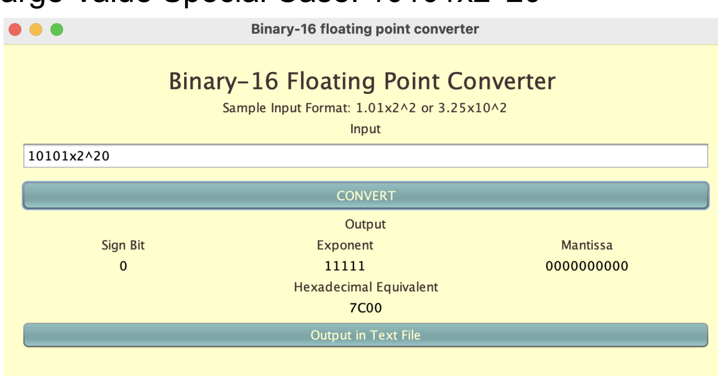


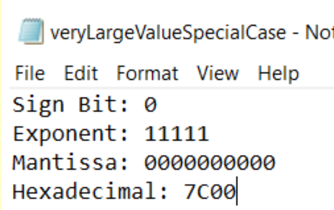
Binary Normal Case: 110100110.111110x2^6



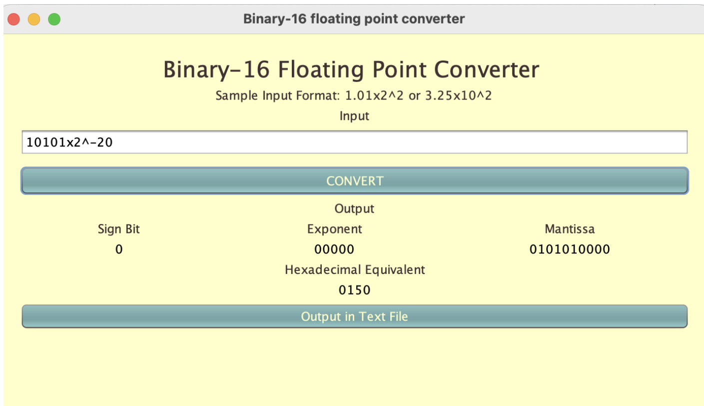


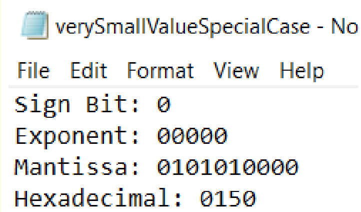
Very Large Value Special Case: 10101x2^20



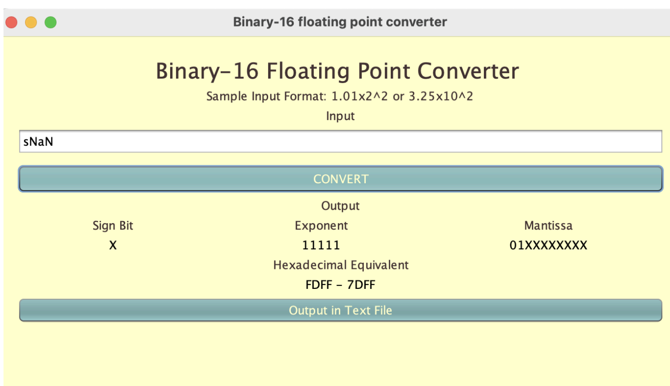


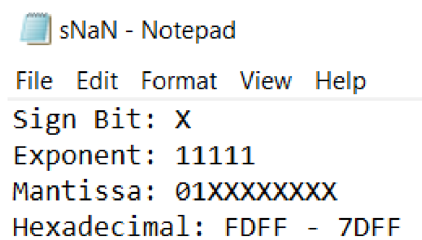
Very Small Value Special Case: 10101x2^-20



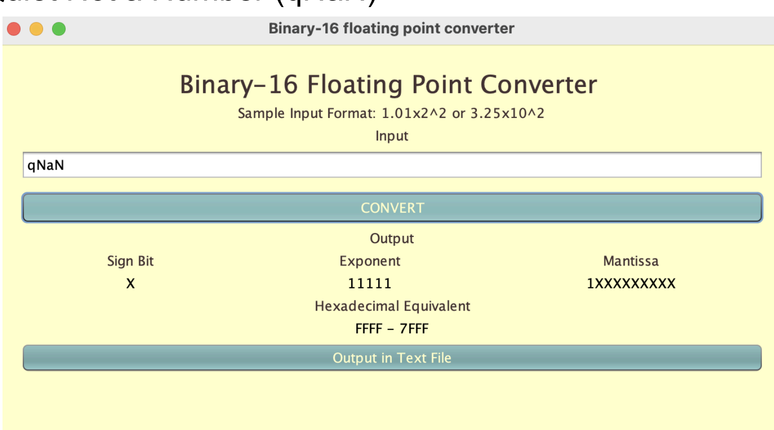


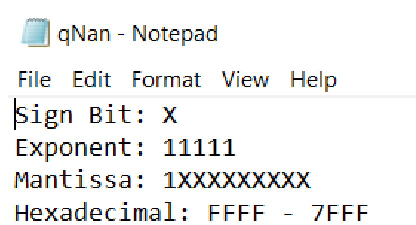
Silent Not a Number (sNaN) Special Case:





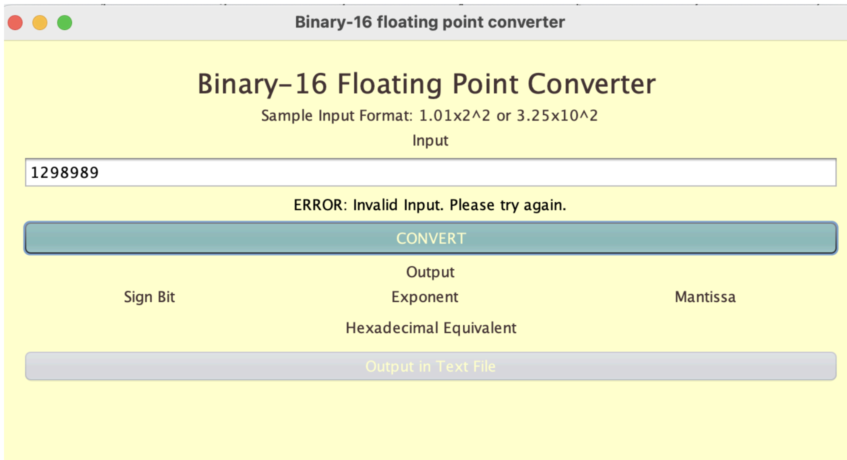
Quiet Not a Number (qNaN)



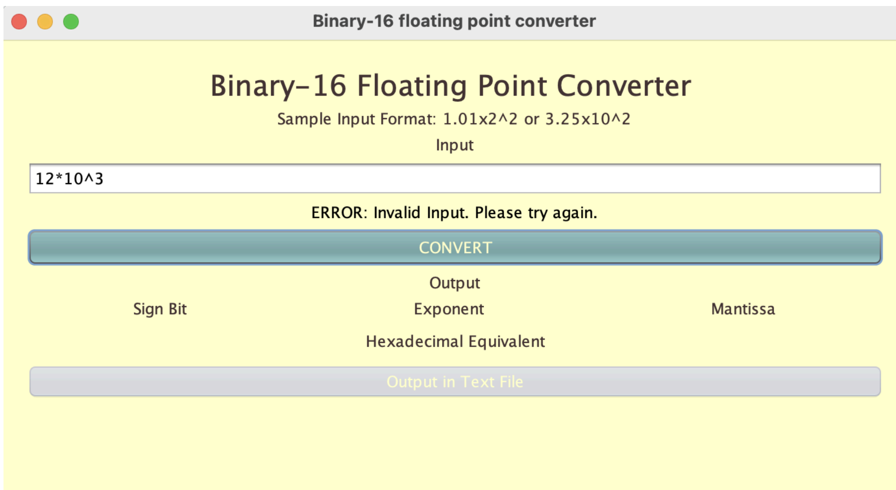


Invalid Inputs

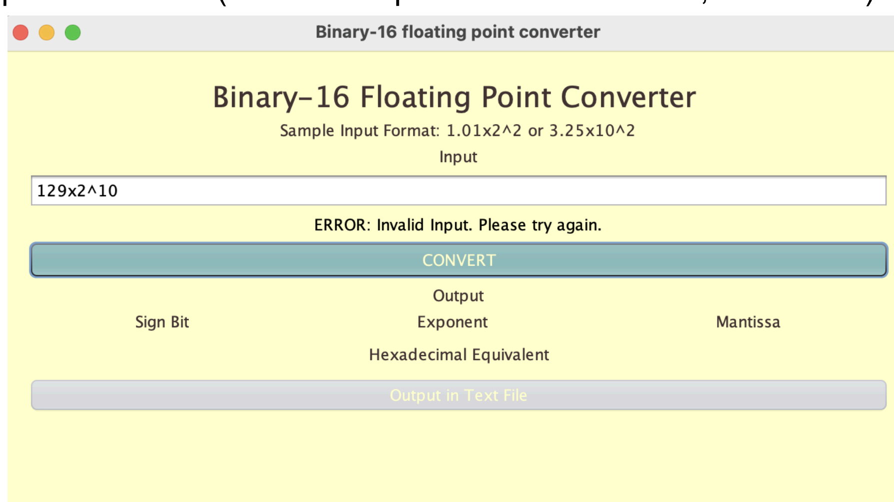
a. No Multiplier

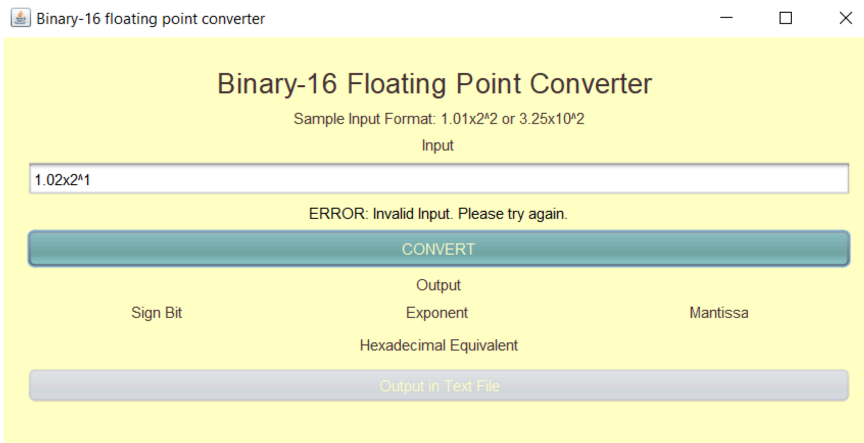


Invalid Symbol (ex. * for multiplication)



Invalid multiplier mismatch (ex. X2 multiplier for base10 value, 1.02x2^10)





Invalid String Input (ex. CSARCH2)