De La Salle University - Manila

In Partial Fulfillment of the Course
Introduction to Computer Organization and
Architecture 2 (CSARCH2)

IEEE-754 Decimal-32 Floating-Point Converter

Technical Report

Submitted by:
Abenoja, Amelia Joyce L.
Gon Gon, Zhoe Aeris F.
Mojica, Harold C.
Sulit, Anne Gabrielle M.
Torio, Ysobella D.

Submitted to:
Mr. Roger Luis Uy

Term 2, A.Y. 2023-2024
March 23, 2024

I. Introduction

Decimal-32 Floating Point is one of the three Decimal Floating Point formats of the IEEE 754-2008 standard (Chen et al., 2009). The parameters for the Decimal-32 Floating Point format are as follows, totaling 32 bits of storage:

Sign Bit	Combination Field	Exponent Continuation	Coefficient Continuaation
1 bit	5 bits	6 bits	20 bits

Table 1. Parameters of Decimal-32 Floating Point

Where:

- The sign bit can either be 0 (positive) or 1 (negative)
- The combination field is 5-bit and composed of the two most significant bits of the exponent representation (valid bits: 00, 01, and 10 only) and 1 or 3 bits of the most significant digit of the significand for finite cases. The infinity type is 11110, while 11111 is for Not a Number (NaN) types.
- The exponent continuation is the rest of the 6 bits in the exponent continuation field, where the maximum normalized exponent value can be represented as 90. On the other hand, -101 is the minimum normalized exponent value.
- The coefficient continuation is the remaining six decimal digits transformed and represented as Densely Packed BCD.

With the understanding of Decimal-32 Floating-Point, the developers created an application that converts the decimal digits to their corresponding Decimal-32 Floating-Point values represented in binary and hexadecimal form with the consideration of special cases such as Not a Number (NaN) and Infinity.

II. Implementation

Tech Stack

The Decimal-32 Floating Point Converter application was developed using Python, a programming language chosen due to its familiarity among all team members. Python's versatility and extensive libraries made it a suitable choice for implementing the converter's functionality. To create the graphical user interface (GUI) for the app, the team utilized the HoloViz panels library. This library was selected for its compatibility with Jupyter Notebook, where the initial development and testing of the app took place.

In addition to the core programming language and GUI library, the team incorporated several Python modules to enhance the app's functionality:

- param: This module was used in conjunction with panels to manipulate the widgets and optimize the user interface.
- os: The os module was employed to manipulate directory and file names, aiding in the organization and management of the app's files and resources.
- tempfile: The tempfile module was utilized to temporarily store the exported output, ensuring efficient and secure data handling.
- fractions and math: These modules were employed to handle special input cases, providing accurate and reliable conversion results for a wide range of input values.

Together, these modules were integrated to create a functional and user-friendly converter web application, catering to both general and special input cases with precision and efficiency.

Densely Packed BCD Implementation

The `bin_to_dpbcd()` function is a pivotal part of the IEEE-754 Decimal-32 Floating-Point Conversion process. This function operates on a binary string that represents three decimal digits in standard packed BCD format. It transforms this 12-bit packed BCD into a 10-bit densely packed BCD following specific logic rules:

- The string input of 12 bits is deconstructed into individual binary digits `a` through `m` (excluding variable `l` for better readability).
- The function computes intermediary variables p through y based on logical operations that involve the unpacked binary digits.
    - The values `r`, `u`, and `y` are determined by the values of `d`, `h`, and `m`, respectively.
    - The value of `v` is determined if the values `a`, `e`, or `i` contain the major bit, 1.
    - The remaining variables follow the following specific logical conditions that are based on the Densely Packed BCD encoding table:

```
p = int(b or (a and j) or (a and f and i))
q = int(c or (a and k) or (a and g and i))
s = int((f and (not a or not i)) or (not a and e and j) or (e and i))
t = int(g or (not a and e and k) or (a and i))
w = int(a or (e and i) or (not e and j))
x = int(e or (a and i) or (not a and k))
```

- The densely packed BCD result is obtained by concatenating the binary representation of p through y.

The overall conversion to densely packed BCD for a six-digit decimal number (with the most significant digit omitted) is achieved by applying the `bin_to_dpbcd()` function to each half (the most significant and the least significant three digits) of the normalized decimal number. The results of both halves are then concatenated to form the final densely packed BCD encoding of the IEEE-754 Decimal-32 Floating-Point representation.

Other Implementations

Processing the input and deriving the remaining output to obtain the final hexadecimal answer proved to be relatively straightforward compared to acquiring the densely packed BCD. The team executed this task by utilizing the insights gained from the CSARCH2 discussions and translating them into actionable code. Initially, during input processing, the team's primary objective was to determine whether there were more than seven digits present in the inputted decimal. Subsequently, a series of different functions were used to (1) normalize the decimal, (2) obtain the final exponent, and (3) calculate the final e-prime. However, the team implemented various modifications to the code to handle special cases such as infinity and NaN, as well as others like zero or those values close to zero itself which is further discussed in *Section IV. Challenges Encountered*.

Transitioning to finalizing the output values, the procedures included identifying the sign bit, combination bits, and exponent bits with the previously mentioned densely packed BCD, thus forming the final binary output. Following this, the conversion from binary to hexadecimal was the final step which was accomplished simply by calling a function that uses the logic learned from the various lessons about base conversion.

III. Test Cases

The developers designed the program to accept positive and negative decimal values, the 'NaN' string for decimal input, and base-10 integers for the exponent field. Once the user clicks the compute button, the converting process displays the normalized decimal, final exponent, and e-prime values. Finally, the output fields display the sign bit, the combination bits, the exponent bits, and the Densely Packed BCD bits before the final answer in Binary and Hexadecimal Form. In Binary Form, each highlighted color field above matches the highlighted bits to distinguish them from one another.

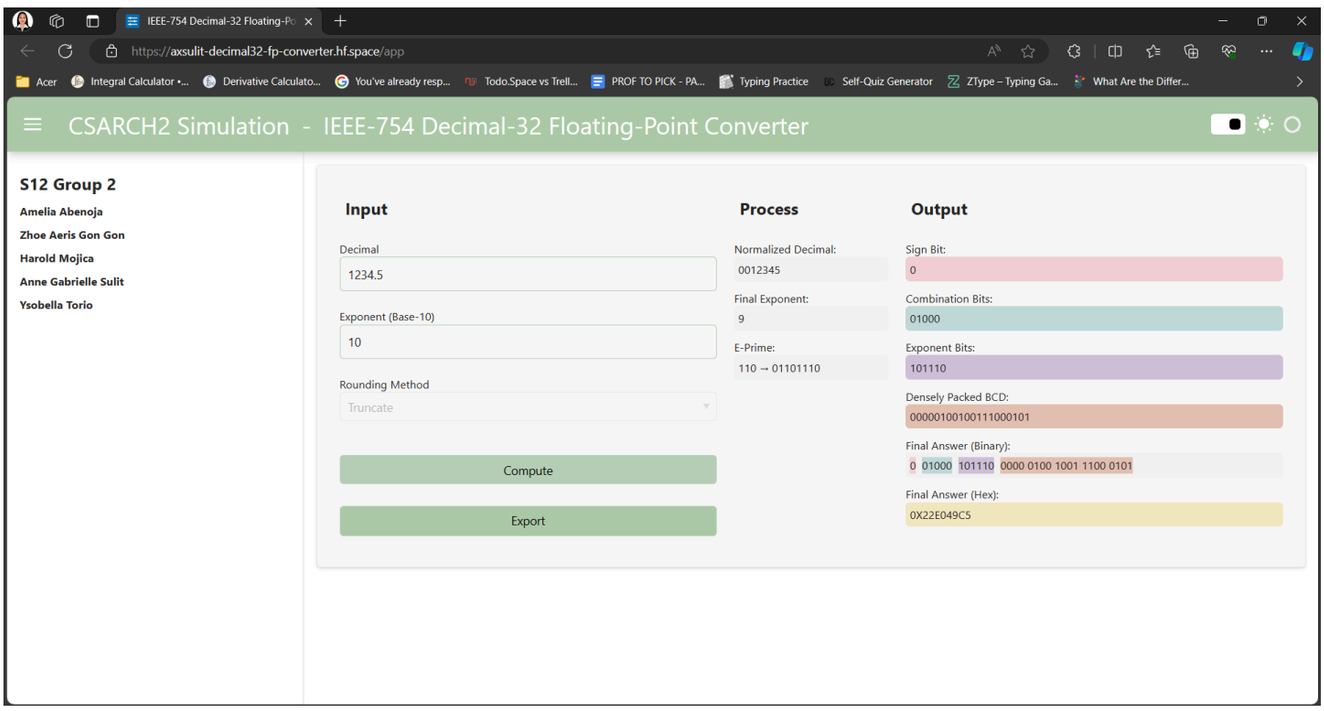

Figure 1. Test Case on Finite Positive

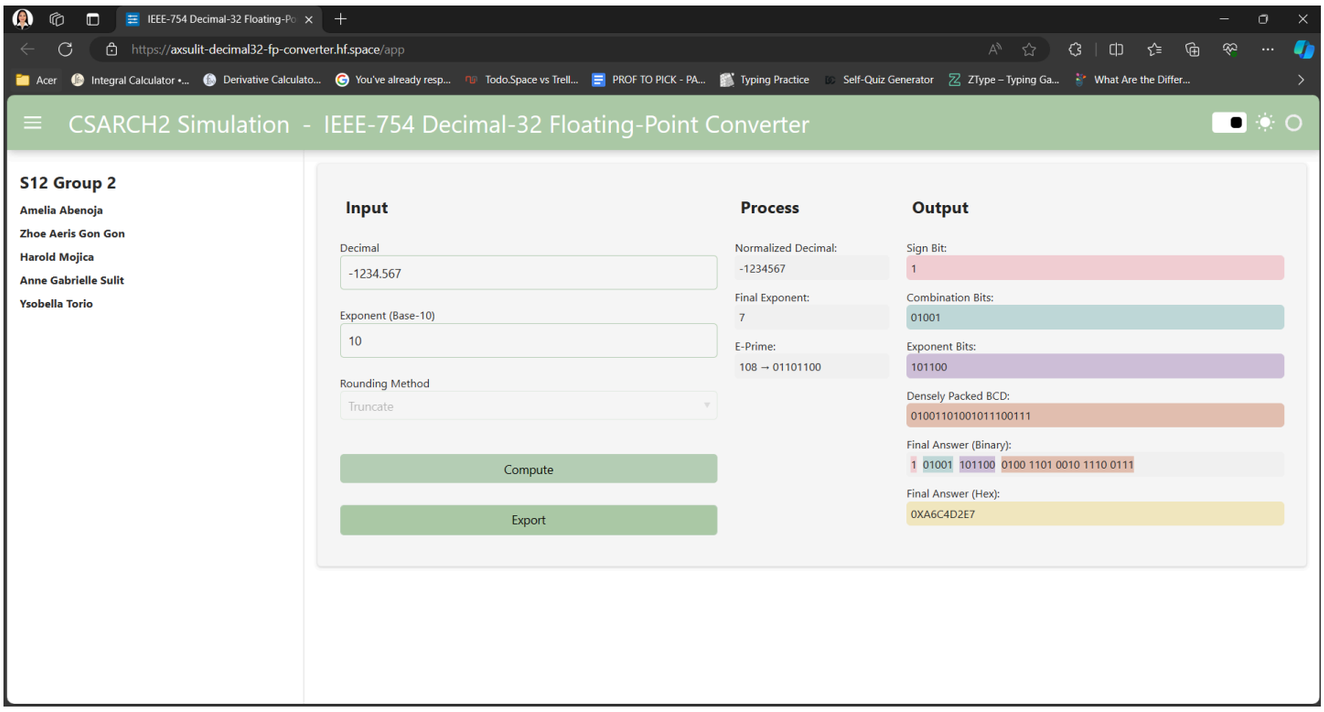

Figure 2. Test Case on Finite Negative

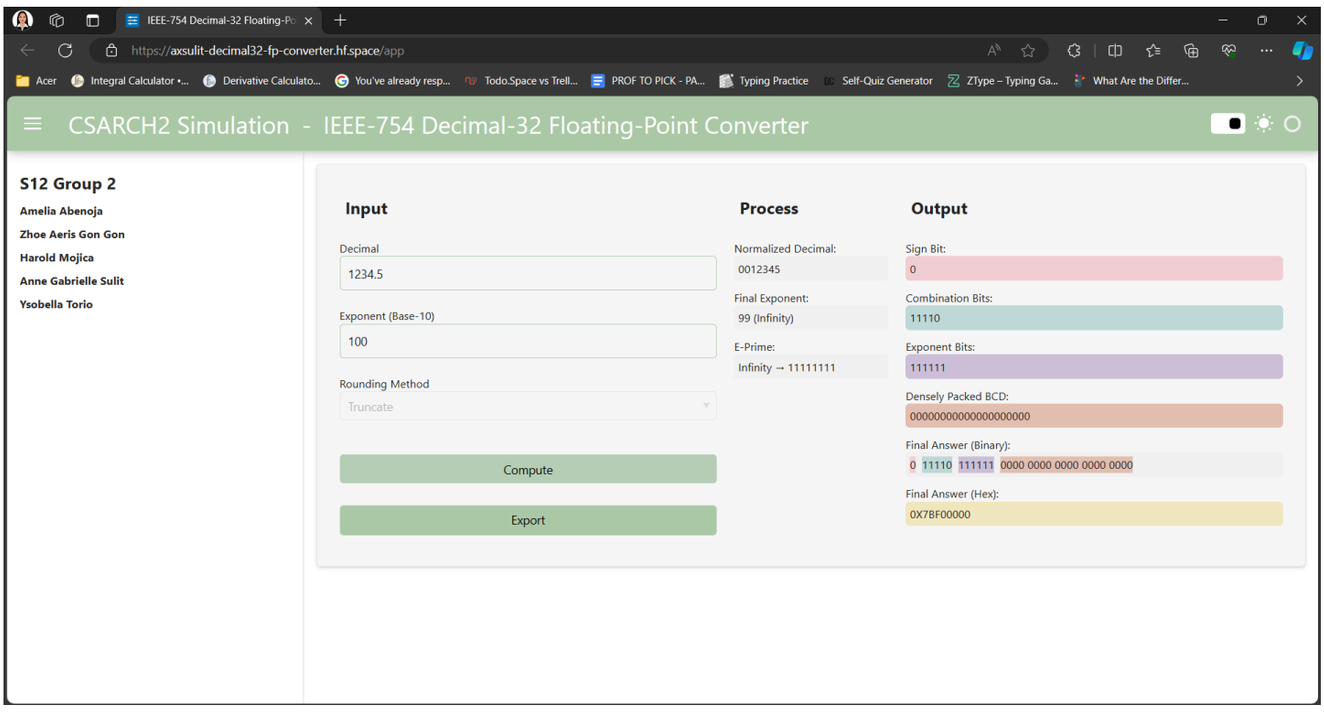

Figure 3. Test Case on Infinite Positive

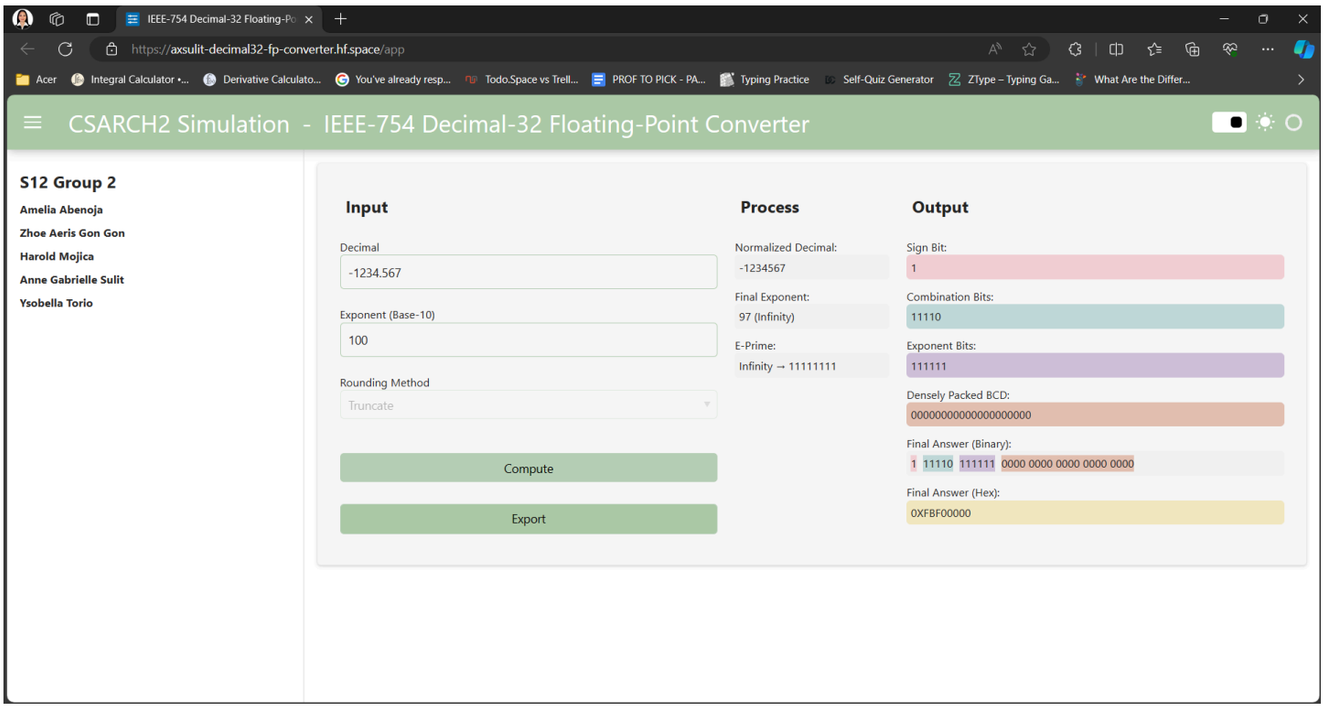

Figure 4. Test Case on Infinite Negative

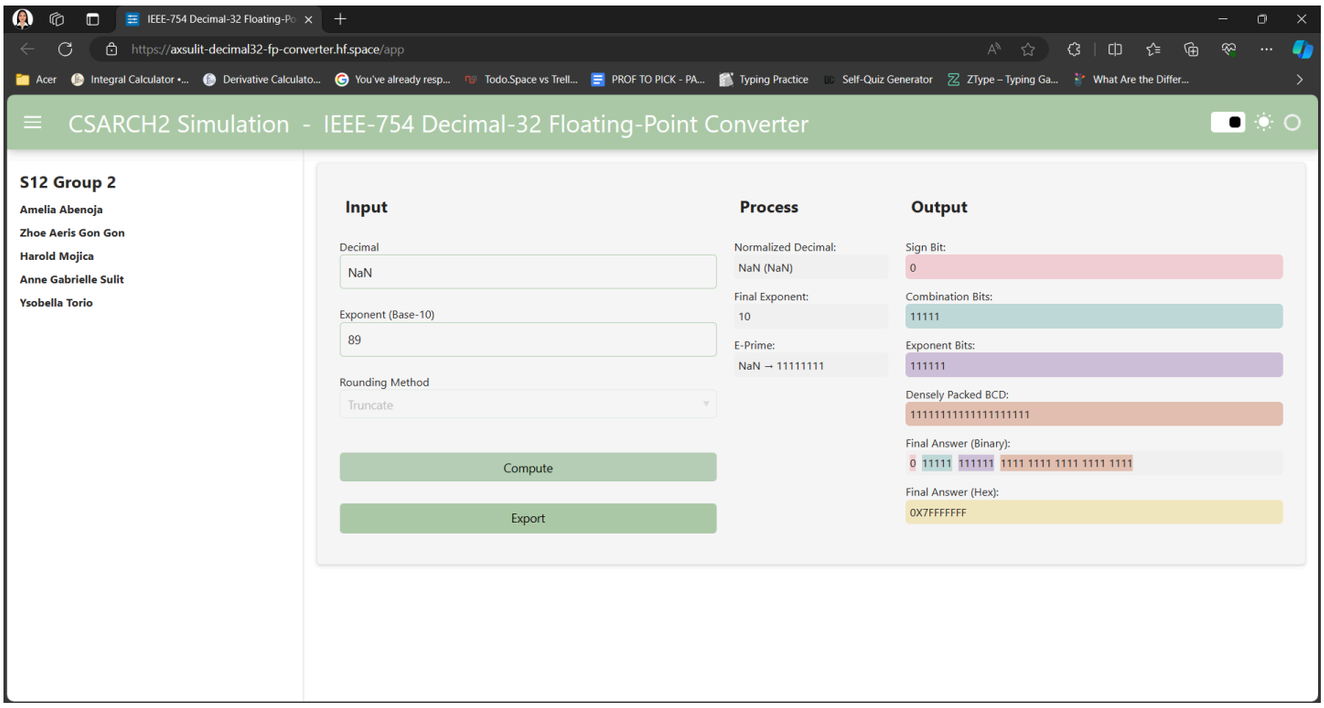

Figure 5. Test Case on Not a Number (NaN)

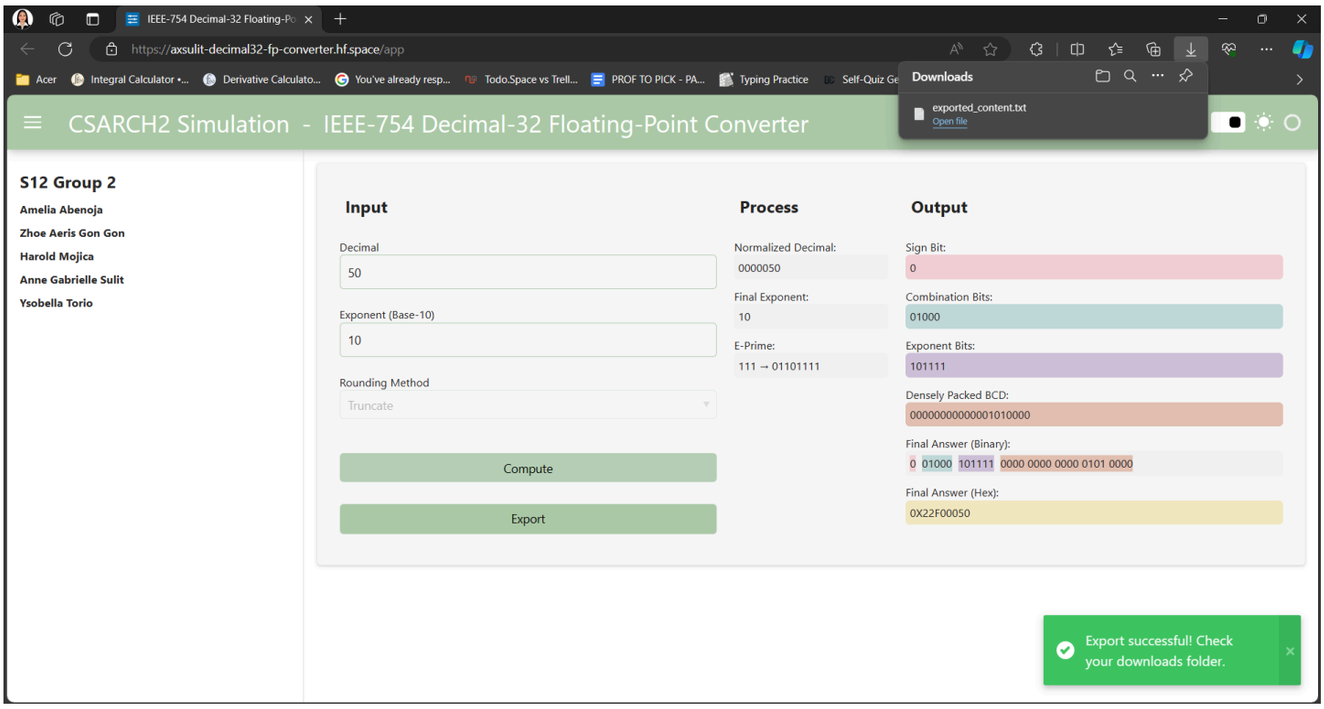

Figure 6. Feedback on Successful Exporting of Outputs in a Text File

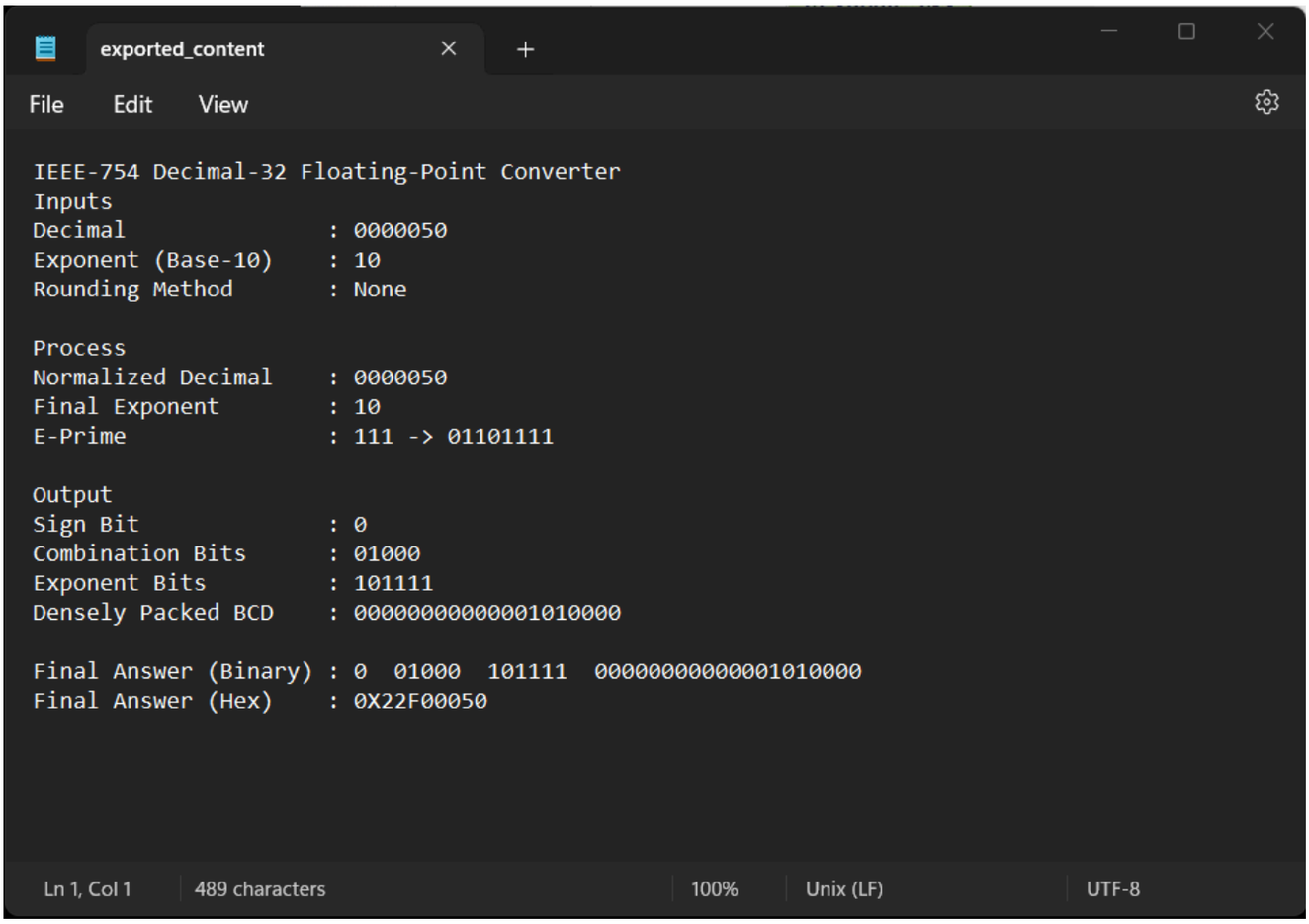

Figure 7. Exported Output of a Not a Number (NaN) Case in a Text File

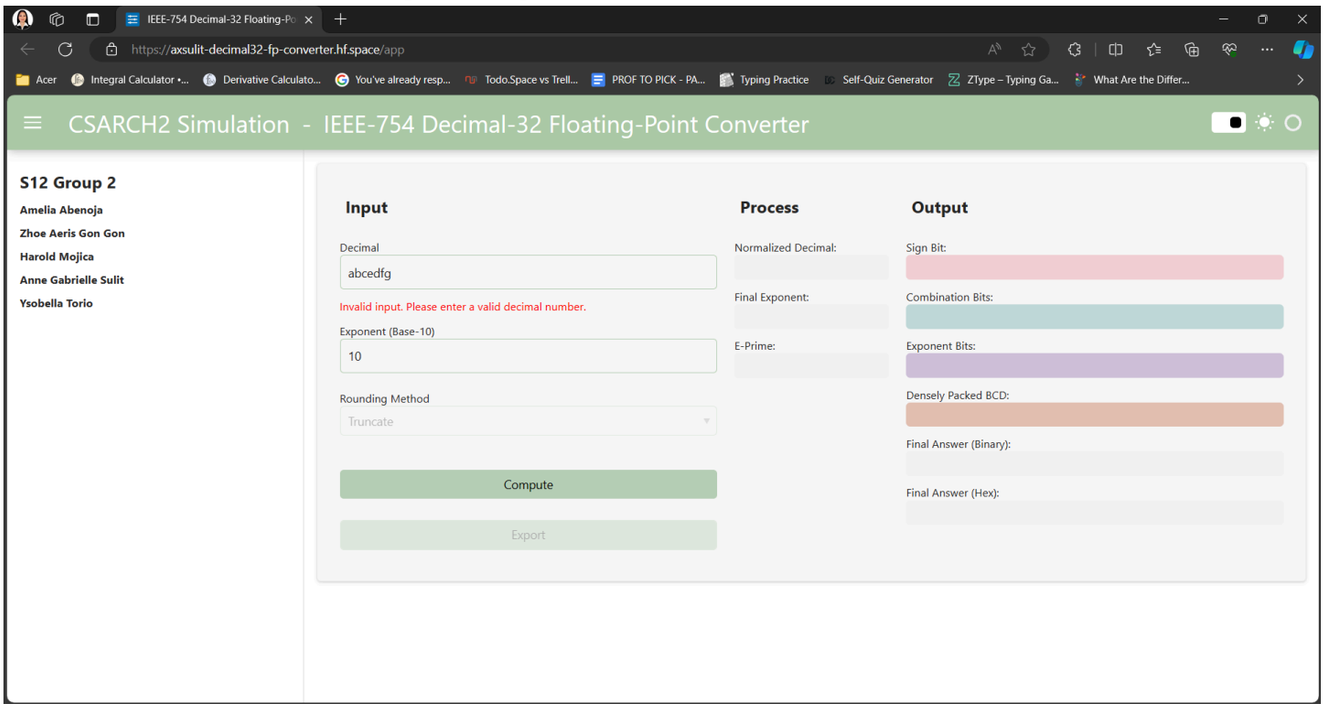

Figure 8. Invalid Decimal Input and Valid Exponent Input

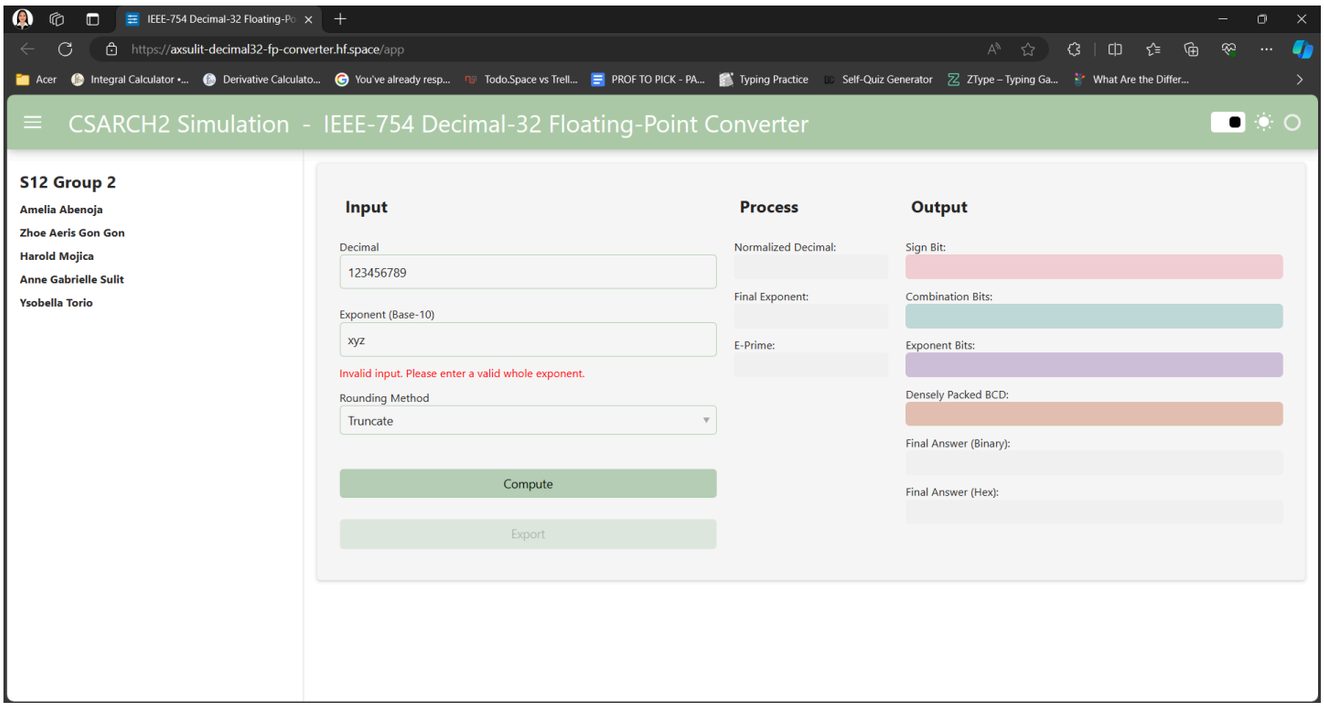

Figure 9. Valid Decimal Input and Invalid Exponent Input

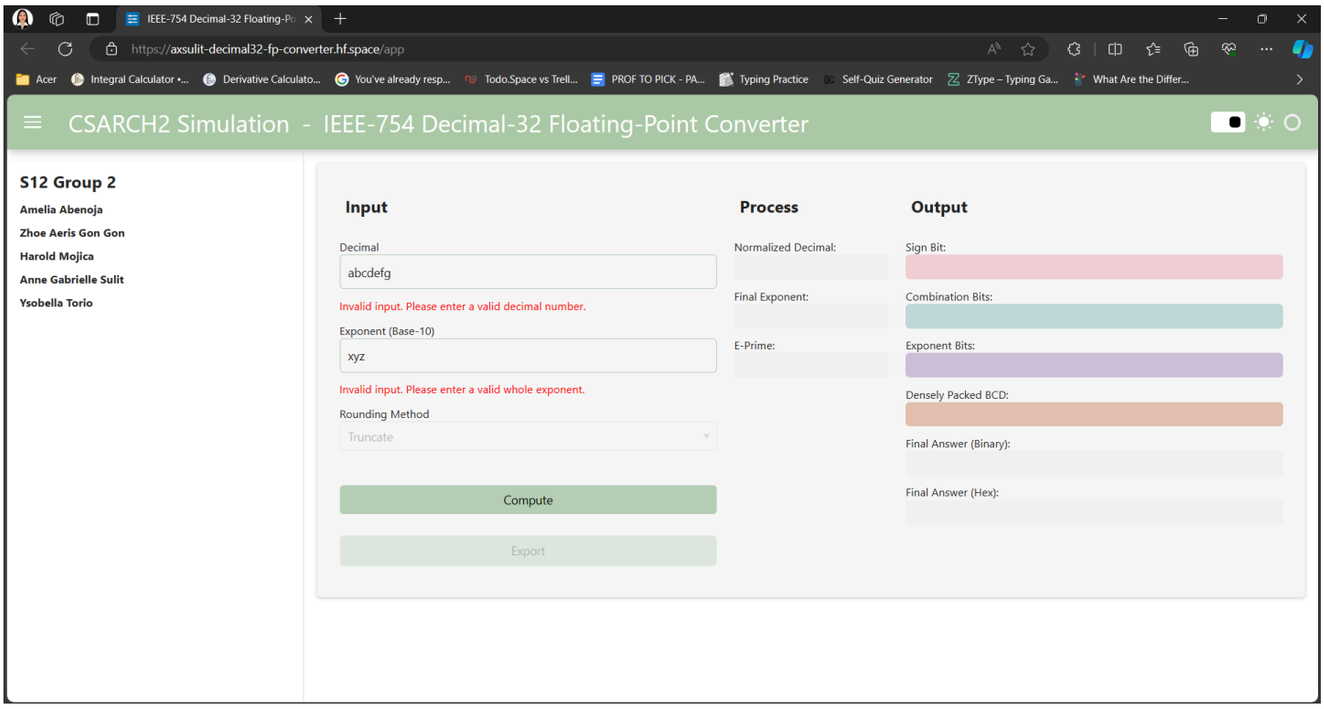

Figure 10. Invalid Decimal Input and Invalid Exponent Input

IV. Challenges Encountered

The team faced several challenges that required careful consideration and problem-solving. These include:

1. Special Cases

A. Infinity

As the first conversion feature that the team addressed, this was difficult to implement due to the learning curve associated with how it ties along with the fractional part of the decimal and the rounding process, and which variables must be utilized and modified accordingly. For instance, it was already known to us that 90 is the limit for the exponent. However, we had to create functions for moving the decimal point and modifying the initial exponent since having an input of *91* or above doesn't immediately equate to an infinity case. Furthermore, we had to take into account the rounding process, as it also influences the number of decimal places the decimal point must move.

B. Denormalized and Zero

Despite not being considered a special case for the IEEE-754 Decimal-32 Floating-Point representation, the team still had to finalize how to handle cases such as denormalized inputs. The challenge itself lies not in the process of coding, but rather in finalizing which outputs must be displayed based on the given inputs. After careful brainstorming, research, and reflection on past notes, the team decided to treat denormalized inputs as zero. In other words, those with a final

exponent less than *-101* would be treated as 0×10^0. The important consideration was to apply similar logic to the special case of infinity, where the process of normalizing and determining the final exponent is first implemented.

Similarly, after completing the feature of handling denormalized values, the team realized the need to handle zero since there may be cases where the inputted exponent is not equal to zero. In this case, the exponent is immediately treated as zero regardless of the input, for uniformity.

C. NaN

The special case NaN was the last conversion feature that was implemented in the application. Throughout the whole process, creating the required input to get a result of NaN was the main problem that was encountered. Since the application is only a converter, we thought that it was not intuitive enough to input equations that can potentially result to NaN, such as computing for division (fraction) and square root of a number. Thus, the implementation was done by entering only the word "NaN". However, this was raised by one member, and it was suggested that the input should allow fraction inputs and the square root of a decimal, in order to demonstrate the special case. Moreover, since Python was utilized to create the application, there are specific libraries that allow the detection of NaN. With that, for every fractional value that a user inputs in the application, it will automatically convert to decimal, while the square root of a number automatically results in its final answer. If a fraction has a denominator with zero while a square root of a negative number is entered, it will automatically return NaN. Alongside that, we also retained the string input of "NaN", to demonstrate the straightforward approach. Lastly, all three types of input allow empty exponent inputs, which will automatically result in 0.

2. App Deployment

Another significant challenge was the deployment of the app. The initial version of the converter was developed in a Jupyter Notebook using the HoloViz panels library. However, the team was uncertain about how to deploy the app outside of the Jupyter environment. After conducting some research, the team discovered that Hugging Face supports the deployment of .py files that use the panels library. Hugging Face is a company known for providing a platform that allows developers to share, train, and deploy machine learning models.

After discovering Hugging Face's deployment capabilities, the team modified the source code to be compatible with Hugging Face's requirements. This enabled the successful deployment of the decimal32 converter app on the Hugging Face platform. The modified code ensured that the app could be executed and accessed outside the Jupyter environment, resolving the deployment challenge effectively.

V. Conclusion

The application, Decimal-32 Floating-Point Converter, is developed using Python with various libraries such as HoloViz Panel to provide a precise overview of the step-by-step process results of the Normalized Decimal, Final Exponent, and E-prime, and outputs of the conversion processes that separately displays the sign bit, combination bits, exponent bits, Densely PAcked BCD, followed by the final results in binary and hexadecimal format. This gives a more intuitive, user-friendly approach to converting decimal digits to floating point representations considering exceptional cases such as Not a Number (NaN) and Infinity.

The team faced significant challenges in implementing the converter, including the complex handling of special cases such as infinity, denormalized numbers, and NaN (Not a Number), which required a deeper understanding to ensure the accuracy of the representation. Moreover, deploying the application outside of the Jupyter environment was challenging due to the consideration of adapting the code for compatibility with the Hugging Face platform. These challenges highlighted the difficulties involved in converting and representing floating-point numbers accurately, as well as the complexities of deploying such applications in different environments.

Despite these challenges, the group successfully deployed the application, which made the application available online for other users to access easily.

VI. References

Chen, D., Zhang, Y., Choi, Y., Lee, M. H., & Ko, S. (2009). A 32-bit Decimal Floating-Point Logarithmic Converter. *2009 19th IEEE Symposium on Computer Arithmetic*, 195–203. https://doi.org/10.1109/arith.2009.22