College of
Computer Studies

Software Technology Department

Exhibit Proposal

Exhibit Project Information

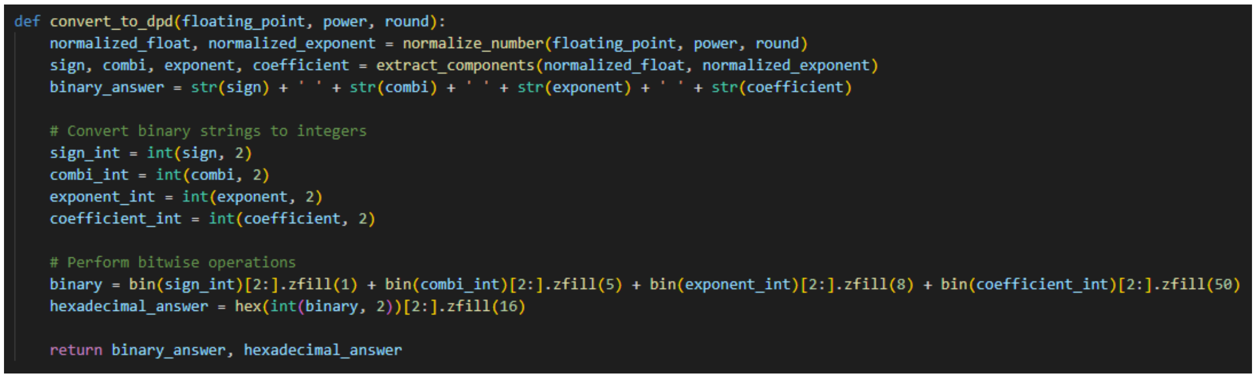
Team Number	3
Section	S12
Team Members	Chong, Jordan Chester Sze
	Gutierrez, Mark Daniel Cacho
	Kua, Miguel Carlo Francisco
	Santos, Kaci Reena Go
	Tuco, Kevin Bryan Layugan
Date Updated	March 23, 2024

I. Code Documentation

For our implementation with regards to the IEEE-754 decimal 64 floating point converter to BCD application, the group decided that in order to proceed with the conversion, it is necessary to normalize the given input by the user.

A. convert_to_dpd() Function

The **convert_to_dpd()** function accepts a number (either float, integer, or string), power (as an integer), and rounding method (as a string). This function changes the numbers from the input and outputs both binary and hexadecimal numbers respectively.

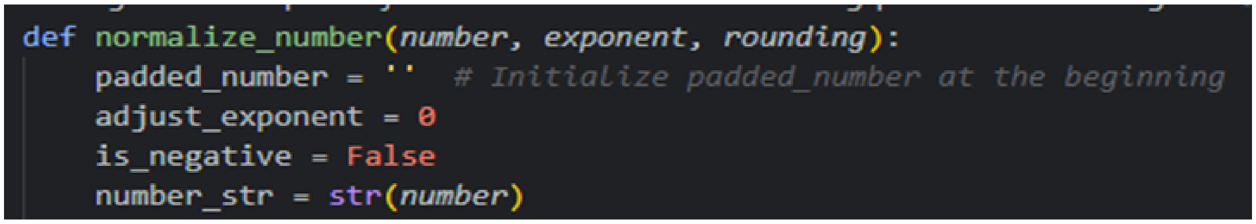


```python
def convert_to_dpd(floating_point, power, round):
    normalized_float, normalized_exponent = normalize_number(floating_point, power, round)
    sign, combi, exponent, coefficient = extract_components(normalized_float, normalized_exponent)
    binary_answer = str(sign) + ' ' + str(combi) + ' ' + str(exponent) + ' ' + str(coefficient)

    # Convert binary strings to integers
    sign_int = int(sign, 2)
    combi_int = int(combi, 2)
    exponent_int = int(exponent, 2)
    coefficient_int = int(coefficient, 2)

    # Perform bitwise operations
    binary = bin(sign_int)[2:].zfill(1) + bin(combi_int)[2:].zfill(5) + bin(exponent_int)[2:].zfill(8) + bin(coefficient_int)[2:].zfill(50)
    hexadecimal_answer = hex(int(binary, 2))[2:].zfill(16)

    return binary_answer, hexadecimal_answer
```

Figure 1: convert_to_dpd function

B. normalize_number() Function

The **normalize_number()** function accepts inputs for the initial input, exponent and the rounding method. Based on Figure 1, the input will be converted into a string value, then it will be first assessed if it is a negative value or not. The main objective of this function is to normalize the input before proceeding to converting it into densely-packed BCD. Aside from this, the function also checks if the input exceeds 16 bits or not. If it exceeds, the program will have to round the value based on the desired input by the user. Finally, the function will end by returning the number alongside the exponent after normalization.

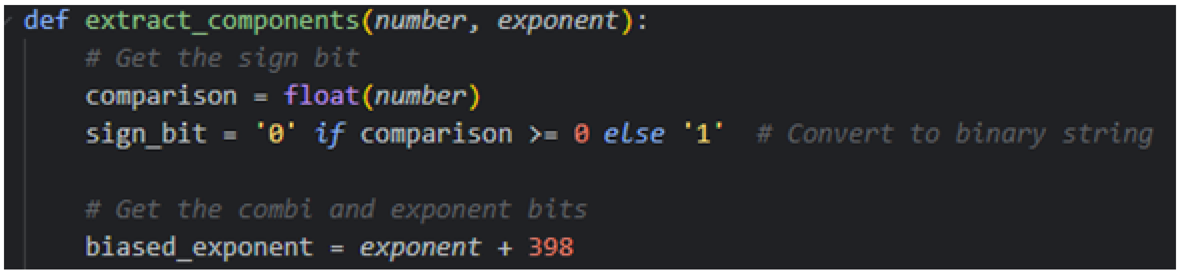


```python
def normalize_number(number, exponent, rounding):
    padded_number = ''  # Initialize padded_number at the beginning
    adjust_exponent = 0
    is_negative = False
    number_str = str(number)
```

Figure 2: normalize_number function.

C. extract_components() Function

Afterwards, the **extract_components()** function outputs the sign, combination, exponent, and coefficient bits. Upon checking the normalized value from the previous function, the program will initially check if the number is negative. If it is, the signed bit will be changed accordingly. Since the program only accepts up to 64-bit floating point input, the highest exponent possible for the input can reach up to 369, which can be seen in Figure 2. There are also special cases within the program if the exponent is more than 369 or less than -398. If the exponent follows these special cases, then the output would be infinity or NaN respectively. Otherwise, the number will be redirected to the next function, **extract_combi()**, which can be seen on Figure 3. The coefficient values will be passed through the function **decode_coefficient().**

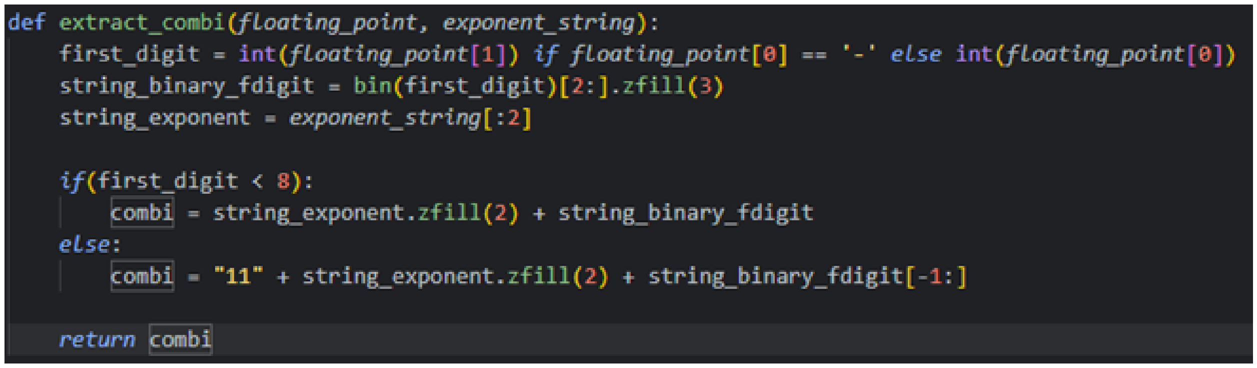


```python
def extract_components(number, exponent):
    # Get the sign bit
    comparison = float(number)
    sign_bit = '0' if comparison >= 0 else '1'  # Convert to binary string

    # Get the combi and exponent bits
    biased_exponent = exponent + 398
```

Figure 3:extract_components function

D. extract_combi() Function

The **extract_combi()** function's key objective is to get the combination bits based on the input and the exponent. The algorithm will first extract the Most Significant digit (MSd) of the input and convert the MSd into a binary value. The first two bits of the combination field will be determined whether the value is a minor or major digit. If the digit is either of the two, it will go through the specific cases resulting in outputting their appropriate combination bits.

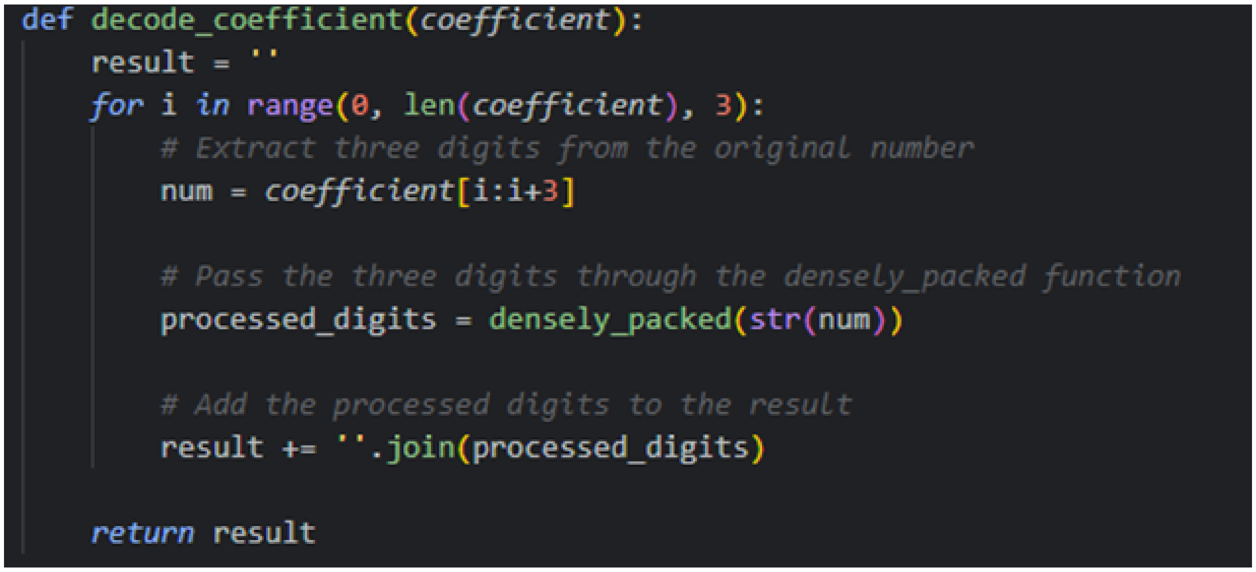


```python
def extract_combi(floating_point, exponent_string):
    first_digit = int(floating_point[1]) if floating_point[0] == '-' else int(floating_point[0])
    string_binary_fdigit = bin(first_digit)[2:].zfill(3)
    string_exponent = exponent_string[:2]

    if(first_digit < 8):
        combi = string_exponent.zfill(2) + string_binary_fdigit
    else:
        combi = "11" + string_exponent.zfill(2) + string_binary_fdigit[-1:]

    return combi
```

Figure 4:extract_combi function

E. decode_coefficients() Function

The **decode_coefficient()** gathers the remaining digits from the normalized input and converts it accordingly to BCD in 5 groups of 3. Throughout this code snippet in Figure 4, the program will continuously iterate in getting the 3 digits and redirecting them into the function named the **densely_packed()**. After getting the BCD from the function, then it will be concatenated until there are no more remaining digits left to be converted.

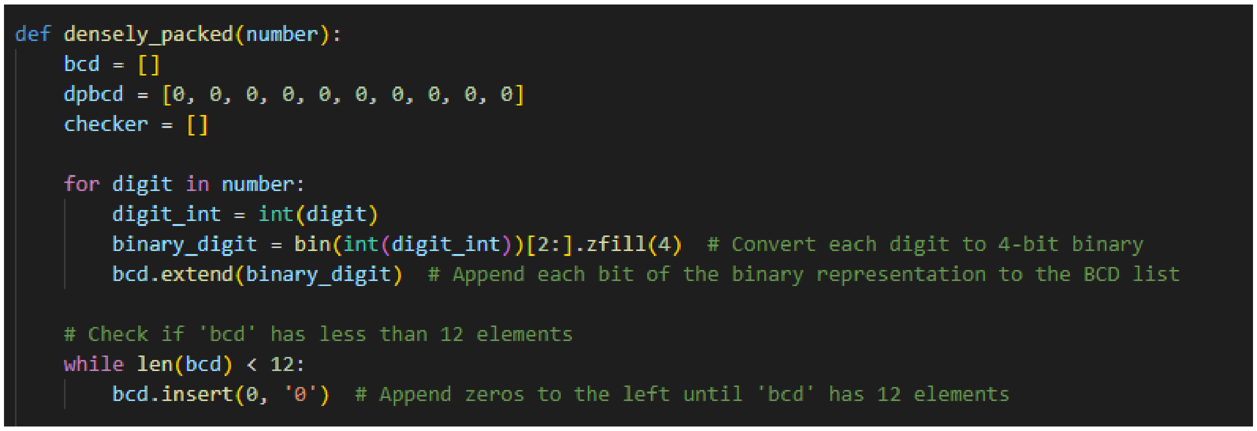


```
def decode_coefficient(coefficient):
    result = ''
    for i in range(0, len(coefficient), 3):
        # Extract three digits from the original number
        num = coefficient[i:i+3]

        # Pass the three digits through the densely_packed function
        processed_digits = densely_packed(str(num))

        # Add the processed digits to the result
        result += ''.join(processed_digits)

    return result
```

Figure 5:decode_coefficient function

F. densely_packed() Function

The **densely_packed()** function will convert the 3 digits sent from the **decode_coefficient()** function and convert them into packed BCD. Once packed, it will then be returned back to **decode_coefficient()** where the 10 bits will be joined with the existing BCD. The densely packed function was derived from the code from

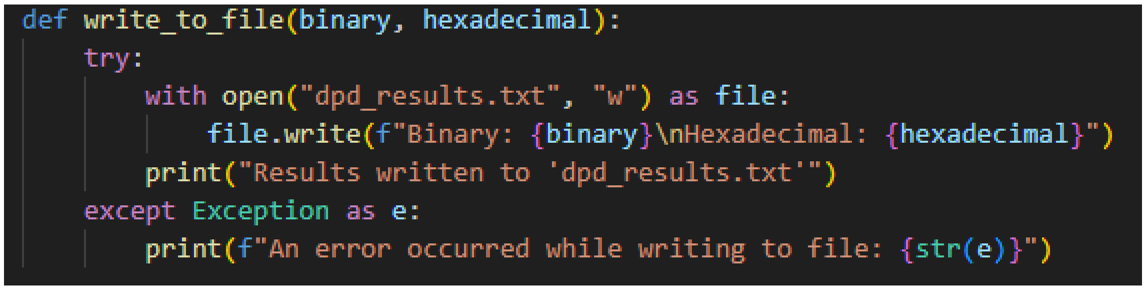


```python
def densely_packed(number):
    bcd = []
    dpbcd = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    checker = []

    for digit in number:
        digit_int = int(digit)
        binary_digit = bin(int(digit_int))[2:].zfill(4)  # Convert each digit to 4-bit binary
        bcd.extend(binary_digit)  # Append each bit of the binary representation to the BCD list

    # Check if 'bcd' has less than 12 elements
    while len(bcd) < 12:
        bcd.insert(0, '0')  # Append zeros to the left until 'bcd' has 12 elements
```

Figure 6: densely_packed function

G. write_to_file() Function

The **write_to_file()** function accepts binary and hexadecimal strings. This function parses the binary and hexadecimal strings and turns these strings into a .txt file.



```python
def write_to_file(binary, hexadecimal):
    try:
        with open("dpd_results.txt", "w") as file:
            file.write(f"Binary: {binary}\nHexadecimal: {hexadecimal}")
        print("Results written to 'dpd_results.txt'")
    except Exception as e:
        print(f"An error occurred while writing to file: {str(e)}")
```

Figure 7: write_to_file function

II. Reflections

For our team, there were a lot of problems that we were facing. Firstly, we were not sure about if what we were doing was correct or not. There are a lot of lessons that convert decimals to binary to the point where we were not sure what to code in the first place. Because of that, we wasted some time trying to look at calculators online that can convert decimals to binary.

After looking at how the calculator works, we thought that the best course of action was to use BCD to convert decimals to binary (in which, it was reinforced that we were on the right track after getting similar responses from our friends stating that they were also doing BCD from Binary). So, the team started working on the simulation project. The developers (Kevin and Jordan) were able to do the code, but the code was filled with bugs to the point where teammates stepped in and debugged all of the functions.

Once the functions were finally debugged, the last step was to officially plug the logic file into the GUI. Thankfully, the developers were able to make a simple GUI that encompasses the specs that the project needed.

In the end, the project was finished on the due date, and the team was able to finish the code properly.