Presented to the College of Computer Studies

De La Salle University - Manila

Term 2, A.Y. 2023-2024

In partial fulfillment of the course

In CSARCH2 S15

Simulation Project:

Binary-128 Floating Point Converter

Submitted by:

Caasi, Samantha Nicole

Lacuesta, Angelo

Latosa, Jose Romulo

Ramilo, Paulo

Marcellana, John Patrick

Submitted to:

Sir Roger Luis Uy

March 23, 2024

I. Introduction

In this analysis, we delve into the development of a Binary-128 floating-point converter application featuring a graphical user interface (GUI), diverging from conventional text-based solutions. This paper aims to describe the methods, design choices, and technical approaches used to create an application that showcases improvements in computational tools and user experience design.

II. Requirements

The task is to create a Binary-128 floating-point converter application that includes all special cases (NaN, Infinity, 0, denormalized). The inputs required are (1) *Binary Mantissa and base-2 exponent* (i.e., 101.01x2^5) or (2) *Decimal and base-10 exponent* (i.e.65.0x10^3). The output is the binary representation with space between sections (sign bit, exponent field) and its hexadecimal equivalent. Additionally, receiving the output in a text file is an option.

III. Methodology

For the programming language, the group decided to use Javascript. HTML and CSS accompany this as they plan to create a web page rather than a stand-alone application.

A. Constants

```
const smallestExponentNormalized = -16382;
const largestExponentNormalized = 16383;
```

Figure 1. Constants of the Javascript code

These constants define the range for the exponents of a normalized input in the Binary-128 format. The -16382 is the smallest exponent possible before reaching denormalization, and 16383 is the largest exponent possible before reaching infinity.

B. Functions

```
const getExcess = (base) => {
  return parseInt(base) + 16383;
};

const getBinary = (decimal) => {
  let binary = decimal.toString(2);
  return binary;
};

const getHex = (binaryDigits) => {
  console.log(binaryDigits);
  let decimal = parseInt(binaryDigits, 2);
  let hex = decimal.toString(16);
  return hex.toUpperCase();
};

const getRemainingDigits = (binaryInput) => {
  let result = "";
  result = binaryInput.toString().substring(2);
  return result;
};

const completeSignificand = (remainingDigits) => {
  while (remainingDigits.length != 112) {
    remainingDigits += "0";
  }
  return remainingDigits;
};
```

Figure 2. Functions for the conversion

- *getExcess(base)*: Calculates the excess for the exponent by adding the provided base to the bias (16383), essential for normalizing the exponent.
- *getBinary(decimal)*: Converts a decimal number to its binary representation.
- *getHex(binaryDigits)*: Converts a binary string to its hexadecimal representation in uppercase.
- *getRemainingDigits(binaryInput)*: Extracts the binary fraction part by removing the '0.' prefix from a binary string.
- *completeSignificand(remainingDigits)*: Ensures the significand (fraction part) has 112 bits, padding with zeroes if necessary, as required by the Binary-128 format.

C. Initialize Binary Digits and Sign Bit

```
// declare the binary digits
let binaryDigits = "";
let signBit = "";
// append the sign bit
if (binaryInput < 0) {
  signBit = "1";
  binaryInput *= -1; // make input positive
} else {
  signBit = "0";
}
binaryDigits += signBit;
```

Figure 3. Sign Bits

- *Initialization*: The variables binaryDigits and signBit are initialized to hold the string representations of the binary digits and the sign bit, respectively.
- *Sign Bit Determination*: The code checks if the input (binaryInput) is negative. If so, the sign bit is set to 1, and the input is turned positive to simplify further processing. If the input is non-negative, the sign bit is set to 0. This bit is the first part of the resulting Binary-128 representation.

D. Normalize Binary Input

```
// normalize the binary input
if (binaryInput > 1) {
  while (Math.floor(binaryInput) != 1) {
    binaryInput /= 10;
    baseInput += 1;
  }
} else if (binaryInput < 1 && binaryInput > 0) {
  while (Math.floor(binaryInput) != 1) {
    binaryInput *= 10;
    baseInput -= 1;
  }
} else {
  // baseInput and binaryinput remains the same
}
```

Figure 4. Normalize Binary

- If the input is greater than 1, it's repeatedly divided by 10, and the baseInput (which tracks the exponent adjustment) is incremented until the input is normalized.
- If the input is between 0 and 1, it's repeatedly multiplied by 10, and the baseInput is decremented to achieve normalization.
- If the input is exactly 1, it's already normalized, and no action is taken.

E. Function Calls: Converting Input to Binary-128 Representation

```
// get the excess
let excess = getExcess(baseInput);

// get the binary value of excess and append
let exponent = getBinary(excess);
binaryDigits = binaryDigits + exponent;

// get the remaining digits
let remainingDigits = getRemainingDigits(binaryInput);

// add zeroes to binary if not complete
let significand = completeSignificand(remainingDigits);
binaryDigits = binaryDigits + significand;

// get hex value
let hexOutput = getHex(binaryDigits);
```

Figure 5. Usage of Functions

- The getExcess function calculates the excess by adding the baseInput to the bias (16383), aligning with the Binary-128 format requirements.
- The excess (now representing the adjusted exponent) is converted to binary and appended to binaryDigits, forming the exponent part of the Binary-128 format.
- After normalization, the fractional part of the binary input is obtained (excluding the '0.' prefix) using getRemainingDigits.
- The completeSignificand function pads this fractional part with zeros until it reaches the length required for the Binary-128 significand (112 bits), ensuring the format's precision requirement is met.
- The complete Binary-128 representation (binaryDigits, which now includes the sign bit, the exponent, and the significand) is converted into hexadecimal using getHex.

F. Special Cases

```
//=========== DENORMALIZED ============
if (baseInput < smallestExponentNormalized) {
    // Denormalized case handling
    exponent = "0".repeat(15);
    binaryDigits = binaryDigits + exponent;

    // Get the denormalized significand
    remainingDigits = getRemainingDigits(binaryInput);
    significand = completeSignificand(remainingDigits)

    binaryDigits = binaryDigits + significand;

    // Get hex value
    hexOutput = getHex(binaryDigits);
```

Figure 6. Denormalized condition

- Checks if baseInput is less than the smallest normalized exponent. In the Binary-128 format, this indicates a denormalized number (very small numbers closer to 0 than what normal numbers can represent).
- Sets the exponent to 15 zeros ("0".repeat(15)) because denormalized numbers use an exponent of all zeros.
- Computes the significand from the binary input, padding it with zeros to meet the 112-bit requirement for the Binary-128 format significand.
- Converts the full binary string (binaryDigits) into hexadecimal (hexOutput).

```
//=========== INFINITY ============
} else if (baseInput > largestExponentNormalized) {
    // Denormalized case handling
    console.log("hello")
    exponent = "1".repeat(15);
    binaryDigits = binaryDigits + exponent;

    // Get the denormalized significand
    remainingDigits = getRemainingDigits("0");
    significand = completeSignificand(remainingDigits)

    // append the significand
    binaryDigits = binaryDigits + significand;

    console.log(binaryDigits)

    // Get hex value
    hexOutput = getHex(binaryDigits);
```

Figure 7. Infinity Condition

- Checks if baseInput exceeds the largest normalized exponent. This condition represents infinity or negative infinity based on the sign bit.
- Sets the exponent to 15 ones ("1".repeat(15)) because infinity is represented with an exponent of all ones and a significand of all zeros.
- Ensures the significand is all zeros by fetching zeros and then padding as necessary.
- Converts the binary representation to hexadecimal.

```
//=========== ZERO ============
} else if (binaryInput == 0) {
    // get the binary value of excess and append
    exponent = "0".repeat(15);
    binaryDigits = binaryDigits + exponent;

    // get the remaining digits
    remainingDigits = getRemainingDigits("0");

    // add zeroes to binary if not complete
    significand = completeSignificand(remainingDigits);
    binaryDigits = binaryDigits + significand;

    // get hex value
    hexOutput = getHex(binaryDigits);
```

Figure 8. Zero Condition

- Checks if the input is exactly zero. This special case is straightforward since both the exponent and significand are zeros.
- Sets the exponent to 15 zeros to represent a zero exponent.
- Ensures the significand is also represented by zeros, padding it to the correct length.
- Converts this representation into hexadecimal.

```
if (!validateBinary(binaryInput)) {
    if(binaryInput != "" && binaryInput != 0 && baseInput != "") {
        alert('NaN input')
        skipForNan = true;
    } else {
        alert('Not a Binary')
        return
    }
}
```

```
if (!validateDecimal(decimalInput)) {
    if(decimalInput === "" && !baseInput &&
    baseInput != 0 || decimalInput === NaN &&
    !baseInput && baseInput != 0 ||
    !decimalInput && decimalInput != 0 &&
    !baseInput && baseInput != 0 ) {
        alert('Not a Number')
        return
    } else {
        alert('NaN input')
        skipForNan = true;
    }
}
```

Figure 9. NaN

- Checks if the input is not a number (NaN) and gives the appropriate prompt

IV. Conclusion

This JavaScript code provides a tool for converting numbers from binary or decimal formats into their IEEE 754 Binary-128 floating-point representation, further encoding this representation into a hexadecimal format. Ultimately, it showcases the complexity of binary floating-point representation and facilitates a deeper understanding of numerical data encoding in computing.