College of Computer Studies

De La Salle University - Manila

Term 3, A.Y. 2023-2024

In partial fulfillment of the course

In CSARCH2 - S13

IEEE-754 binary-32 floating point operation

Submitted by:

Group 1

Capacio, John Dionel
Chung, Ford Ainstein T.
Cipriano, Johans Venedict
Recto, Rylan

Submitted to:

Pascual, Ronald

July 30, 2024

I. Introduction

The IEEE-754 binary-32 floating point operation project is about the creation of a Web-based application with a Graphical User Interface (GUI). This application aims to demonstrate the addition of two binary floating-point numbers which follow the IEEE-754 standard.

II. Getting Started

To run the application, you may check out our README.md file/

After following the README.md file and running the application, it asks for the following inputs:

- Two binary floating-point numbers in base-2 representation.
- How many number of digits supported
- Type of Rounding of your choice. You may choose either Guard/Round/Sticky (G/R/S) or traditional rounding.

Once inputted and pressed ‘Calculate’, it will show a step-by-step operation of how it got the final answer.

III. Analysis

This section provides a detailed explanation of how our application works. We will provide snippets of the code containing all the functions used. Once the user presses ‘Calculate’, the first thing that will happen is to check if all the inputs are valid.

```JavaScript
function validateInput(op1, op2, nBits, round) {
        let message = "";
        let message2 = "";
        let bValid = true;
        let error = false;

        $(".error-container").empty();
        let op1Temp = op1.replace(".", "");
        let op2Temp = op2.replace(".", "");

        if (op1Temp[0] == "-") {
            op1Temp = op1Temp.substring(1);
        }

        if (op2Temp[0] == "-") {
            op2Temp = op1Temp.substring(1);
        }

        var i;
        for (i = 0; i < op1Temp.length; i++) {
            if (op1Temp[i] != "0" && op1Temp[i] != "1") {
                bValid = false;
            }
        }
        for (i = 0; i < op2Temp.length; i++) {
            if (op2Temp[i] != "0" && op2Temp[i] != "1") {
                bValid = false;
            }
        }


        if (op1.length == 0 || op2.length == 0 || nBits.length == 0) {
            $(".error-container").append("<p class=\"error\">Please Provide all
inputs.</p>");
            error = true;
        }
        if (nBits <= 0) {
            $(".error-container").append("<p class=\"error\">Number of Digits
must be greater than 0</p>");
            error = true;
        }

        if (!bValid) {
            $(".error-container").append("<p class=\"error\">Input is not in
Binary.</p>");
            error = true;
        }

        return error;
    }
```

Index.js contains the validateInput function, which takes in the binary operands (‘op1’ and ‘op2’), the number of bits (‘nBits’), and the rounding mode (‘round’).

JavaScript
```
let op1Temp = op1.replace(".", "");
        let op2Temp = op2.replace(".", "");

        if (op1Temp[0] == "-") {
            op1Temp = op1Temp.substring(1);
        }

        if (op2Temp[0] == "-") {
            op2Temp = op1Temp.substring(1);
        }

        var i;
```

The function first removes the decimal points from the operands for validation purposes. It will then modify the operands by removing the negative signs if present.

```JavaScript
for (i = 0; i < op1Temp.length; i++) {
            if (op1Temp[i] != "0" && op1Temp[i] != "1") {
                bValid = false;
            }
        }
        for (i = 0; i < op2Temp.length; i++) {
            if (op2Temp[i] != "0" && op2Temp[i] != "1") {
                bValid = false;
            }
        }
```

They will then be looped to make sure that they only contain the binary digits (‘0’ or ‘1’). If any non-binary digit is found, ‘bValid’ will be set to false.

```JavaScript
if (op1.length == 0 || op2.length == 0 || nBits.length == 0) {
            $(".error-container").append("<p class=\"error\">Please Provide all
inputs.</p>");
            error = true;
        }
        if (nBits <= 0) {
            $(".error-container").append("<p class=\"error\">Number of Digits
must be greater than 0</p>");
            error = true;
        }
```

These two if statements check whether any input field is empty, and if ‘nBits’ is a non-positive value. If any are true, it will add an error message.

```
JavaScript
if (!bValid) {
            $(".error-container").append("<p class=\"error\">Input is not in
Binary.</p>");
            error = true;
        }
```

Lastly, it will check if ‘bValid’, which was used to check if the operands are binary digits, is true or false. If nBits is false, the error will be true. This will return the value of the error. If all inputs are valid, it will continue the function: step1.

```
JavaScript
function step1(op1, op2, nBits, round, exp1, exp2, s1, s2){
        download+= "Step 1: Normalization\n";
        let op1Dec = op1.indexOf(".");
        let op2Dec = op2.indexOf(".");
        let res1 = normalize(op1, nBits);
        let res2 = normalize(op2, nBits);

        // index 0 contains exp while index 1 contains the number itself
        // calculate new exponent
        exp1 += parseInt(res1[0]);
        exp2 += parseInt(res2[0]);
        op1 = res1[1];
        op2 = res2[1];

        return alignDecimal(op1, op2, nBits, round, exp1, exp2, s1, s2);
    }
```

‘op1Dec’ and ‘op2Dec’ locates the position of the decimal point in each operand. If the decimal point is not present, ‘indexOf’ will return -1. ‘res1’ and ‘res2’ use the normalize function.

```
JavaScript
function normalize(op, nBits) {
        let one = op.indexOf("1");
        let n, result;
        if (op.indexOf(".") == 1 && op.indexOf("1") == 0) {
            n = 0;
            result = op;
        } else if (op.indexOf(".") == -1) {
            result = op.substring(0, one+1) + "." + op.substring(one + 1);
            n = op.length - one - 1;
        } else {
            if (op.indexOf(".") < one) {
                n = op.indexOf(".") - one;
            } else {
                n = op.indexOf(".") - one - 1;
            }

            op = op.replace(".", "");
            one = op.indexOf("1");

            result = op.substring(0, one + 1) + "." + op.substring(one + 1);
        }

        result = result.substring(op.indexOf("1"));

        // extend to required digits
        if (result.length - 1 < nBits) {
            let tempLen = result.length - 1;
            for (let i = 0; i < nBits - tempLen; i++) {
                result += "0";
            }
        }

        return [n, result];
    }
```

The normalize function normalizes a binary string to a floating-point representation. It adjusts the position of the decimal point and ensures the binary number fits within the specified number of bits that the user inputted.

```JavaScript
exp1 += parseInt(res1[0]);
        exp2 += parseInt(res2[0]);
        op1 = res1[1];
        op2 = res2[1];

return alignDecimal(op1, op2, nBits, round, exp1, exp2, s1, s2);
```

Going back to the step1 function, the exponents are updated by adding the exponent values obtained from the normalization function. These are then assigned to ‘op1’ and ‘op2’. It will then return a value that is being received from another function called alignDecimal.

```JavaScript
function alignDecimal(op1, op2, nBits, round, exp1, exp2, s1, s2) {
        if (exp1 > exp2) {
            let n = exp1 - exp2;
            op2 = shiftRight(nBits, op2, n);
            exp2 += n;
        }

        if (op1 < op2) {
            let n = exp2 - exp1;
            op1 = shiftRight(nBits, op1, n);
            exp1 += n;
        }

        //insert Result
        $(".align").append("<p class=\"results\"> Operator1: "+ s1 + op1
+"</p>");
        $(".align").append("<p class=\"results\"> Exponent1: "+ exp1 +"</p>");
        $(".align").append("<p class=\"results\"> Operator2: "+ s2 + op2
+"</p>");
        $(".align").append("<p class=\"results\"> Exponent2: "+ exp2 +"</p>");

        download += "\n\nAlign Decimal Points\n\n";
        download += "Op1: " + s1 + op1 + " x2^"+exp1 + "\n";
        download += "Op2: " + s2 + op2 + " x2^"+exp2 + "\n";

        // Make into appropriate length
        if (round == "1") {
            op1 = GRS(op1, nBits);
            op2 = GRS(op2, nBits);
        } else {
            op1 = rounding(op1, nBits);
            op2 = rounding(op2, nBits);
        }

        $(".round1").append("<p class=\"results\"> Operator1: "+ s1 +op1
+"</p>");
        $(".round1").append("<p class=\"results\"> Operator2: "+ s2 +op2
+"</p>");
        download += "\n\nRound to Required Length:\n\n";
        download += "Op1: " + s1 + op1 + " x2^"+exp1 + "\n";
        download += "Op2: " + s2 + op2 + " x2^"+exp2 + "\n";

        return [op1, op2, exp1, exp2];

    }
```

The purpose of the alignDecimal function is to adjust the decimal points of the two binary numbers, ‘op1’ and ‘op2’ by changing it so that both numbers have the same exponent value. In order to make it into appropriate length, it may use the GRS function or the rounding function, depending on the user’s initial choice of the type of rounding.

```JavaScript
function GRS(op, nBits) {

        if (op.length - 1 == nBits)
            return op + "000";

        let g, r, s = 0;
        let excess = op.substring(parseInt(nBits) + 1)
        op = op.substring(0, parseInt(nBits) + 1);

        for (let i = excess.length; i < 3; i++) {
            excess += "0";
        }

        g = excess[0];
        r = excess[1];

        let sTemp = excess.substring(2);

        for (let i = 0; i < sTemp.length; i++) {
            if (sTemp[i] == "1") {
                s = "1";
            }
        }

        return op + g + r + s;

    }
```

Let’s start off with the GRS function. This function follows the Guard, Round, and Sticky rounding method to the binary. If for example, the binary number is already at the length that is supposed to be, it appends “000” as guard bits. However, for longer numbers, it will examine the excess bits. It sets the guard bit as the first excess bit, the round bit as the second excess bit, and determines the sticky bit as ‘1’ if any remaining excess bits are ‘1’. This will then return the rounded binary.

```javascript
function rounding(op, nBits) {
        let carry = "";
        let result = "";
        // nearest even
        if (op.length - 1 != nBits) {
            nDigits = op.substring(0, parseInt(nBits) + 1);
            excess = op.substring(parseInt(nBits) + 1)
            let half = excess;
            half = half.replaceAll("1", "0").replace("0", "1");

            let check = compareBinary(half, excess);

            if (check == 1) {
                return op.substring(0, parseInt(nBits) + 1);
            } else if (check == 2 || check == 3) {
                if(op[nBits] == "."){
                    //if the last character is the decimal point
                    tempRes = parseInt(op[nBits-1]) + 1;
                    nBits--;
                    if (check == 3 && op[nBits-1] == 0) {
                        return op.substring(0, parseInt(nBits) + 1);
                    }
                }else{
                    tempRes = parseInt(op[nBits]) + 1;
                    if (check == 3 && op[nBits] == 0) {
                        return op.substring(0, parseInt(nBits) + 1);
                    }
                }

                tempCarry = Math.floor(tempRes / 2);
                tempRes = tempRes % 2;
                result += "" + tempRes;
                carry += "" + tempCarry;

                for (let i = nBits - 1; i >= 0; i--) {
                    if (op[i] == ".") {
                        result += ".";
                        continue;
                    }

                    tempRes = parseInt(op[i]) + parseInt(carry[carry.length -
1]);

                    tempCarry = Math.floor(tempRes / 2);
                    tempRes = tempRes % 2;

                    result += "" + tempRes;
                    carry += "" + tempCarry;
                }

                if (tempCarry == 1) {
                    result += "1";
                    carry += "0";
                }

                result = Array.from(result).reverse().join("");

                return result;
            }
        } else {
            return op;
        }


    }
```

Next is the rounding function. As many already know, normal rounding is when a binary number is rounded to a specific number of bits using the “nearest even” method. Let's say for example that the current binary number is not equal to ‘nBits’. It will shorten the binary number to ‘nits’ and analyze the excess bits. It will calculate the rounding value by flipping the excess buts and use the compare function to the original number to determine how to round.

Going back to the alignDecimal function, it will return the following, ‘op1’, ‘op2’, ‘exp1’, and ‘exp2’, which will be used in the later steps.

```
Java
function step2(op1, op2, sign1, sign2){
        download += "\n\nStep 2: Addition Operation\n";

        // for formating
        let aligned = alignFloatingPoints(op1, op2);
        op1 = aligned.alignedOp1;
        op2 = aligned.alignedOp2;
        let placeholder;
        let resSign = " ";


        if ((sign1 == "-" || sign2 == "-") && !(sign1 == "-" && sign2 == "-"))
{
            let greater = compareBinary(op1, op2);
            if (greater == 1 || greater == 3) {
                placeholder = subtraction(op1, op2);
                resSign = sign1;
            } else if (greater == 2) {
                placeholder = subtraction(op2, op1);
                resSign = sign2;
            }
            placeholder[1] = " " + placeholder[1]
        } else {
            placeholder = addition(op1, op2);
        }


        let result = placeholder[0];
        let carry = placeholder[1];
        $(".op-table").append("<p class=\"results\">&nbsp &nbsp &nbsp &nbsp" +
carry + "</p>");
        $(".op-table").append("<p class=\"results\">&nbsp &nbsp &nbsp &nbsp" +
sign1 + op1 + "</p>");
        $(".op-table").append("<p class=\"results\">&nbsp &nbsp &nbsp &nbsp" +
sign2 + op2 + "</p>");
        $(".op-table").append("<p class=\"results\">&nbsp&nbsp + </p>");
        $(".op-res").append("<p class=\"results\">  &nbsp &nbsp &nbsp &nbsp"+
resSign +result +"</p>");
        download += "\n\nAddition Operation\n\n";
        download += "Carry   |     " + carry + "\n";
        download += "Op1     |    " + sign1 + op1 + "\n";
        download += "Op2     |    " + sign2 + op2 + "\n";
        download += "           +\n";
        download += "\n--------------------------------------------\n"
        download += "             " + resSign + result + "\n";

        return [resSign, result];
    }
```

Now we go to the step2 function. This function will now perform binary floating-point addition. The code will first ensure that ‘op1’ and ‘op2’ have the same exponent and alignment for the decimal point before adding.

```JavaScript
if ((sign1 == "-" || sign2 == "-") && !(sign1 == "-" && sign2 == "-")) {
    let greater = compareBinary(op1, op2);
    if (greater == 1 || greater == 3) {
        placeholder = subtraction(op1, op2);
        resSign = sign1;
    } else if (greater == 2) {
        placeholder = subtraction(op2, op1);
        resSign = sign2;
    }
    placeholder[1] = " " + placeholder[1];
} else {
    placeholder = addition(op1, op2);
}
```

This snippet above checks the signs of ‘op1’ and ‘op2’. If one number is negative, it will use the compareBinary function in order to check which one is bigger. This will then use the subtraction function and set the result sign.

```JavaScript
function subtraction(op1, op2) {
        let carry = "", tempCarry = 0;
        let result = "", tempRes;

        for (let i = op1.length - 1; i >= 0; i--) {
            if (op1[i] == ".") {
                result += ".";
                carry += " ";
                continue;
            }
            tempRes = parseInt(op1[i]) - parseInt(op2[i]) - tempCarry;
            if (tempRes < 0) {
                tempCarry = 1;
                tempRes += 2;
            } else {
                tempCarry = 0;
            }

            carry += tempCarry;
            result += tempRes;

        }


        return [Array.from(result).reverse().join(""),
Array.from(carry).reverse().join("")];
    }
```

This is the subtraction function. To put it in simple terms, this performs digit-by-digit subtraction. For usability, the two numbers are put into a temporary string. It will then loop both the strings and move towards the beginning. The variable, 'tempRes' will calculate the result of the digits and will be modified by 'tempCarry'. And when I say beginning, It goes to the leftmost digit. For better visualization, here is an example:

If op1 = "123" and op2 = "099", the function will:

- Process 3 - 9 (borrow needed).
- Process 2 - 9 (with carry).
- Process 1 - 0 (no carry).

The output will be the result of the subtraction. Going back to the step2 function, if both numbers have the same, it performs binary addition using the addition function. The addition function will also perform digit-by digit binary addition which is similar to the subtraction function.

```javascript
let result = placeholder[0];
let carry = placeholder[1];
```

```
$(".op-table").append("<p class=\"results\">&nbsp &nbsp &nbsp &nbsp" + carry +
"</p>");
$(".op-table").append("<p class=\"results\">&nbsp &nbsp &nbsp &nbsp" + sign1 +
op1 + "</p>");
$(".op-table").append("<p class=\"results\">&nbsp &nbsp &nbsp &nbsp" + sign2 +
op2 + "</p>");
$(".op-table").append("<p class=\"results\">&nbsp&nbsp + </p>");
$(".op-res").append("<p class=\"results\">  &nbsp &nbsp &nbsp &nbsp"+ resSign +
result +"</p>");

download += "\n\nAddition Operation\n\n";
download += "Carry   |     " + carry + "\n";
download += "Op1     |    " + sign1 + op1 + "\n";
download += "Op2     |    " + sign2 + op2 + "\n";
download += "           +\n";
download += "\n----------------------------------------------\n"
download += "             " + resSign + result + "\n";
return [resSign, result];
```

For displaying purposes, it will append the results to the HTML, so that the user will be able to see the the step-by-step procedures during step 2. After it has been appended, the function will return the sign of the result and the result itself.

```
JavaScript
function step3(sign, added, nBits, exp1, round){
        download += "\n\nStep 3: post-operation normalization\n";
        //normalize check for overflow or just adjust the decimal point
        let placeholder;
        let exp = exp1;
        let result;
        // grs == 1, else rounding
        placeholder = normalize(added, nBits);
        exp += parseInt(placeholder[0]);
        result = placeholder[1];

        var latexExpression = `^ {${exp}}`;
        $(".overflow").append("<p class=\"results\"> Normalized: "+ sign +
result +" x 2"+ `\\( ${latexExpression} \\)` + "</p>");
        MathJax.typeset();

        download += "Normalized: " + sign + result + " x2^" + exp + "\n";
        result = rounding(result, nBits);


        placeholder = normalize(result, nBits);
        exp += parseInt(placeholder[0]);
        result = placeholder[1];

        $(".post-rounding").append("<p class=\"results\"> Rounding: "+ sign
+result +" x 2"+ `\\( ${latexExpression} \\)` +"</p>");
        MathJax.typeset();
        download += "Rounding: " + sign + result + " x2^" + exp + "\n";
        $(".final-answer").append("<p class=\"results\"> FINAL ANSWER: "+ sign
+result +" x 2"+ `\\( ${latexExpression} \\)`+"</p>");
        MathJax.typeset();
        download += "\n\nFinal: "+ sign + result + " x2^" + exp + "\n";
    }
```

Last but not least, we have the final step, also the step3 function. Initially, the function normalizes the result obtained from the step2 function. The ‘placeholder’ will call the normalize function.

```Java
function normalize(op, nBits) {
        let one = op.indexOf("1");
        let n, result;
        if (op.indexOf(".") == 1 && op.indexOf("1") == 0) {
            n = 0;
            result = op;
        } else if (op.indexOf(".") == -1) {
            result = op.substring(0, one+1) + "." + op.substring(one + 1);
            n = op.length - one - 1;
        } else {
            if (op.indexOf(".") < one) {
                n = op.indexOf(".") - one;
            } else {
                n = op.indexOf(".") - one - 1;
            }

            op = op.replace(".", "");
            one = op.indexOf("1");

            result = op.substring(0, one + 1) + "." + op.substring(one + 1);
        }

        result = result.substring(op.indexOf("1"));

        // extend to required digits
        if (result.length - 1 < nBits) {
            let tempLen = result.length - 1;
            for (let i = 0; i < nBits - tempLen; i++) {
                result += "0";
            }
        }
```

The normalize function modifies the binary number ‘op’ to fit the specified number of bits, which is ‘nBits’. It first identifies the position of the first 1 and the decimal point. If the decimal point is present before the first 1, or if there is no decimal point, it inserts a decimal point appropriately. It then removes any leading zeros and extends the result with trailing zeros to ensure it meets the required bit length.

And once that is done, it will go back to the continuation of the step3 function, and update the exponent based on the normalization result.

```JavaScript
result = rounding(result, nBits);

placeholder = normalize(result, nBits);
exp += parseInt(placeholder[0]);
result = placeholder[1];

$(".post-rounding").append("<p class=\"results\"> Rounding: "+ sign +result +" x 2"+ `\\( ${latexExpression} \\)` +"</p>");
MathJax.typeset();

download += "Rounding: " + sign + result + " x2^" + exp + "\n";
 $(".final-answer").append("<p class=\"results\"> FINAL ANSWER: "+ sign +result +" x 2"+ `\\( ${latexExpression} \\)`+"</p>");
        MathJax.typeset();
        download += "\n\nFinal: "+ sign + result + " x2^" + exp + "\n";
```

Next is, finally rounding the final result. Similar to how it was used in the alignDecimal function, the result will be using the rounding function, so that the ‘result’ will be rounded in a specified number of bits, ‘nBits’. Once that is done, it will once again normalize it with the result and nBits as the new parameters. And now the final answer will be appended to the Display Screen. The user may also be able to print the steps and answer via a .txt file, which was processed during the functions.

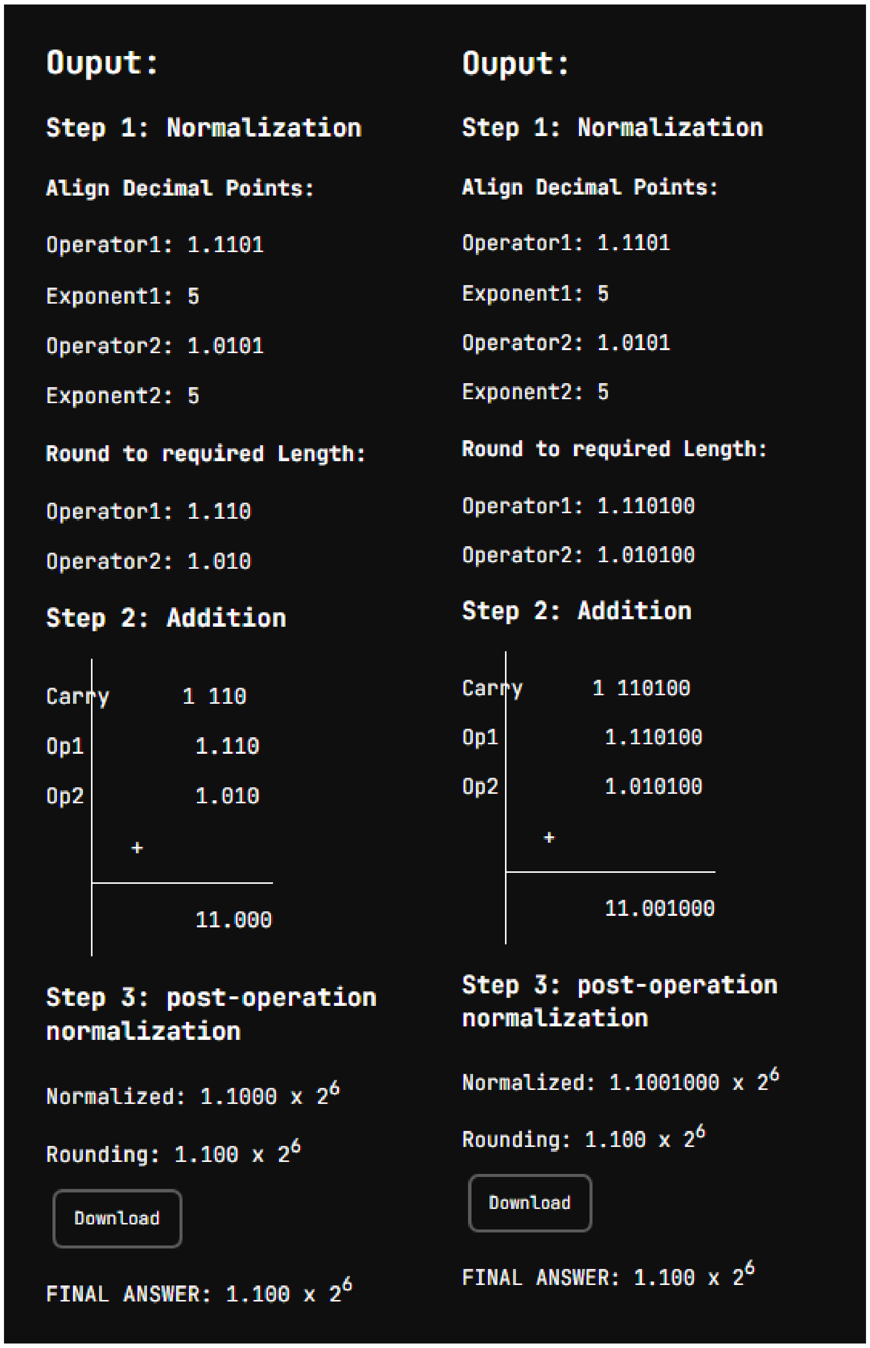
IV. Testcases

Below are possible test cases that will cover the specifications.

A. Test case 1

Operation 1 = 1.1101
Exponent 1 = 5
Operation 2 = 101.01
Exponent 2 = 3
Digits = 4
Rounding = GRS
Expected = 1.100 x 2^6

Operation 1 = 1.1101
Exponent 1 = 5
Operation 2 = 101.01
Exponent 2 = 3
Digits = 4
Rounding = Rounding
Expected = 1.100 x 2^6



```
Ouput:

Step 1: Normalization

Align Decimal Points:

Operator1: 1.1101

Exponent1: 5

Operator2: 1.0101

Exponent2: 5

Round to required Length:

Operator1: 1.110

Operator2: 1.010

Step 2: Addition

Carry      1 110
Op1         1.110
Op2         1.010
     +
         ---------
           11.000

Step 3: post-operation
normalization

Normalized: 1.1000 x 2^6

Rounding: 1.100 x 2^6
```

Download

```
FINAL ANSWER: 1.100 x 2^6
```

```
Ouput:

Step 1: Normalization

Align Decimal Points:

Operator1: 1.1101

Exponent1: 5

Operator2: 1.0101

Exponent2: 5

Round to required Length:

Operator1: 1.110100

Operator2: 1.010100

Step 2: Addition

Carry      1 110100
Op1         1.110100
Op2         1.010100
     +
         ------------
           11.001000

Step 3: post-operation
normalization

Normalized: 1.1001000 x 2^6

Rounding: 1.100 x 2^6
```

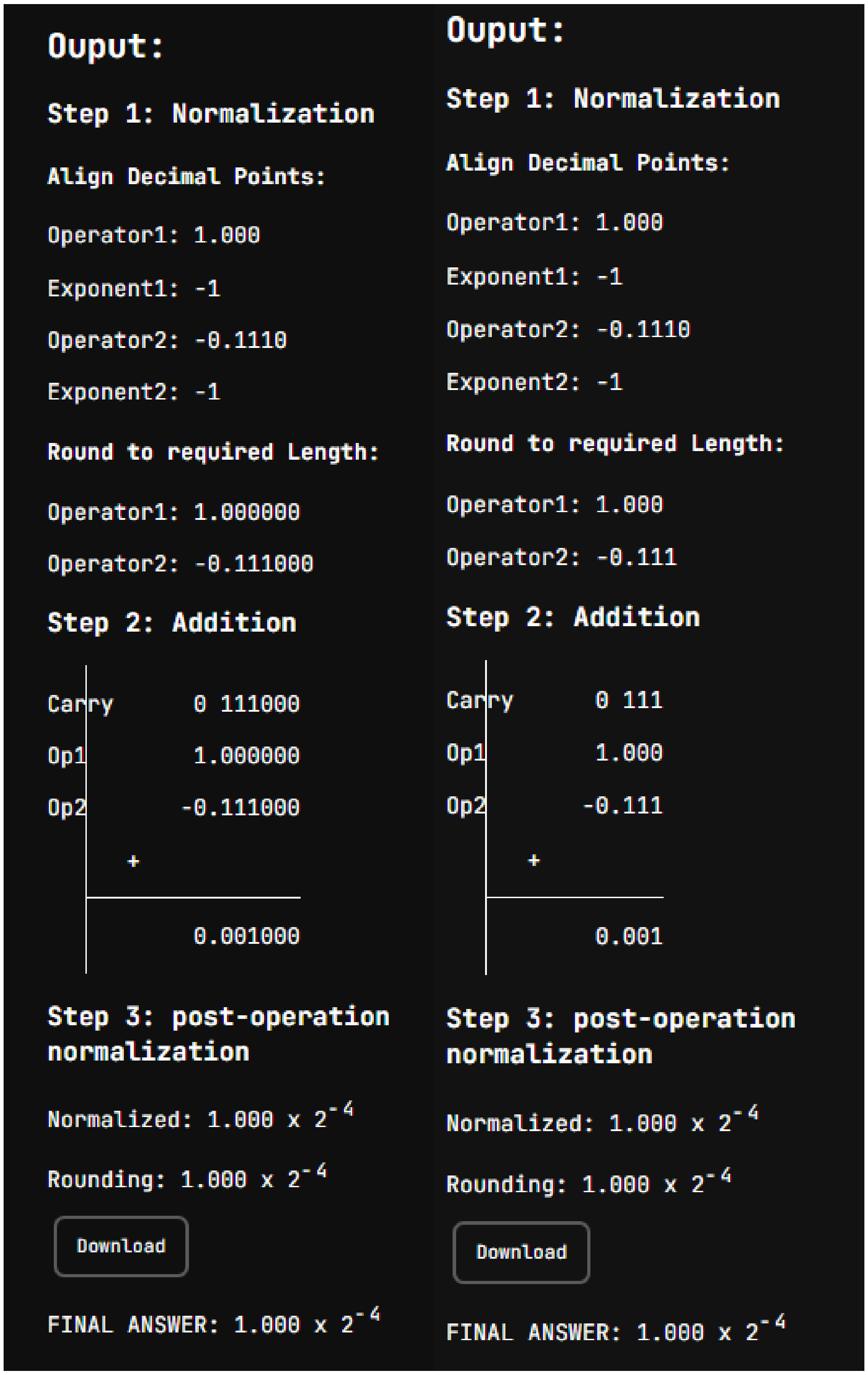
Download

```
FINAL ANSWER: 1.100 x 2^6
```

B. Test case 2

Operation 1 = 1.000
Exponent 1 = -1
Operation 2 = -1.110
Exponent 2 = -2
Digits = 4
Rounding = GRS
Expected = 1.000 x 2^-4

Operation 1 = 1.000
Exponent 1 = -1
Operation 2 = -1.110
Exponent 2 = -2
Digits = 4
Rounding = Rounding
Expected = 1.000 x 2^-4



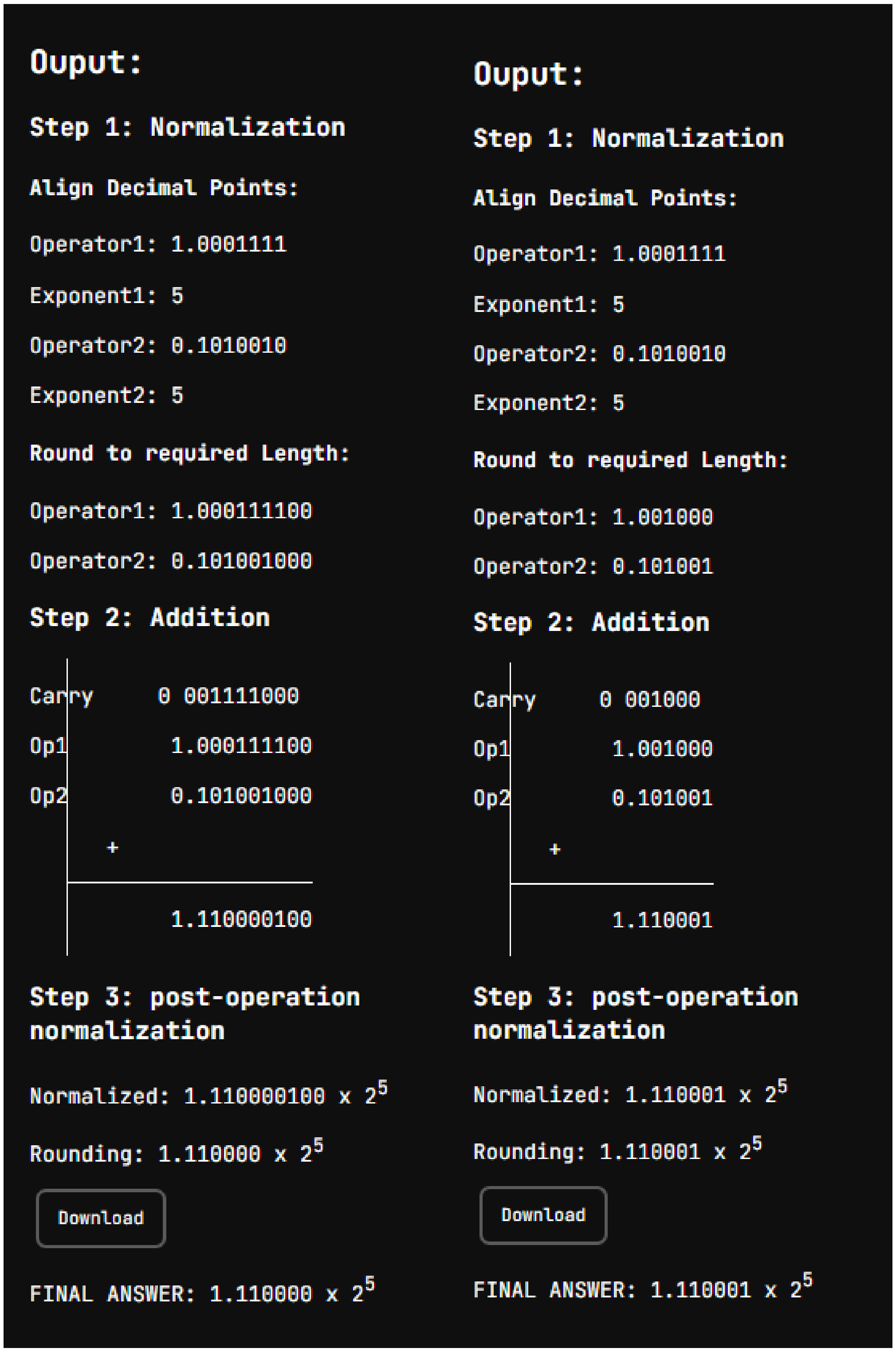
```
Ouput:

Step 1: Normalization

Align Decimal Points:

Operator1: 1.000

Exponent1: -1

Operator2: -0.1110

Exponent2: -1

Round to required Length:

Operator1: 1.000000

Operator2: -0.111000

Step 2: Addition

Carry      0 111000
Op1         1.000000
Op2        -0.111000
     +
         ----------
            0.001000

Step 3: post-operation
normalization

Normalized: 1.000 x 2^-4

Rounding: 1.000 x 2^-4

Download

FINAL ANSWER: 1.000 x 2^-4
```

```
Ouput:

Step 1: Normalization

Align Decimal Points:

Operator1: 1.000

Exponent1: -1

Operator2: -0.1110

Exponent2: -1

Round to required Length:

Operator1: 1.000

Operator2: -0.111

Step 2: Addition

Carry      0 111
Op1         1.000
Op2        -0.111
     +
         -------
            0.001

Step 3: post-operation
normalization

Normalized: 1.000 x 2^-4

Rounding: 1.000 x 2^-4

Download

FINAL ANSWER: 1.000 x 2^-4
```

C. Test Case 3

Operation 1 = 1.0001111
Exponent 1 = 5
Operation 2 = 1.01001
Exponent 2 = 4
Digits = 7
Rounding = GRS
Expected = 1.110000x2^5

Operation 1 = 1.0001111
Exponent 1 = 5
Operation 2 = 1.01001
Exponent 2 = 4
Digits = 7
Rounding = Rounding
Expected = 1.110001x2^5



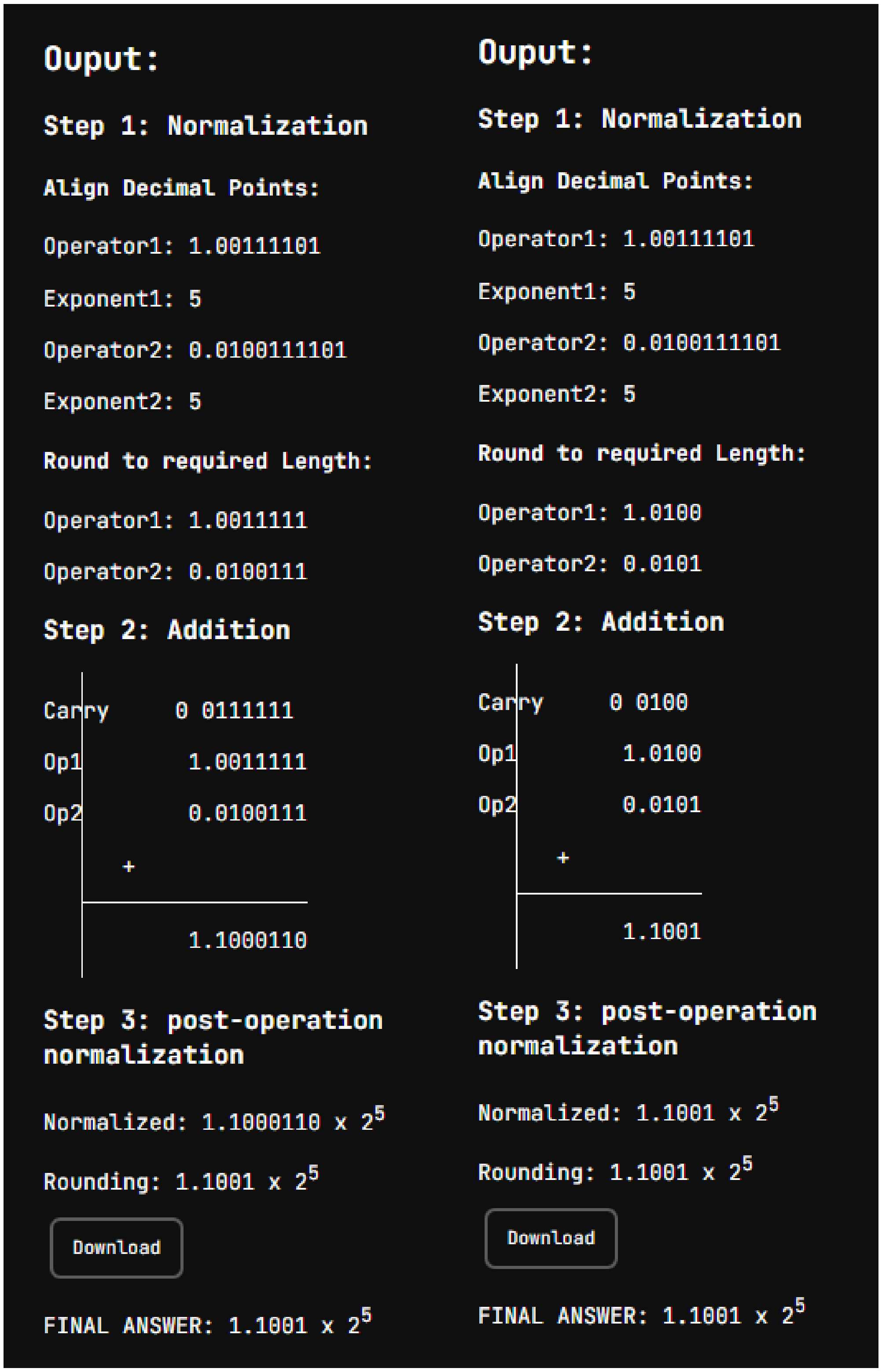
```
Ouput:

Step 1: Normalization

Align Decimal Points:

Operator1: 1.0001111

Exponent1: 5

Operator2: 0.1010010

Exponent2: 5

Round to required Length:

Operator1: 1.000111100

Operator2: 0.101001000

Step 2: Addition

Carry     0 001111000
Op1         1.000111100
Op2         0.101001000
      +
            -----------
            1.110000100

Step 3: post-operation
normalization

Normalized: 1.110000100 x 2^5

Rounding: 1.110000 x 2^5

Download

FINAL ANSWER: 1.110000 x 2^5
```

```
Ouput:

Step 1: Normalization

Align Decimal Points:

Operator1: 1.0001111

Exponent1: 5

Operator2: 0.1010010

Exponent2: 5

Round to required Length:

Operator1: 1.001000

Operator2: 0.101001

Step 2: Addition

Carry     0 001000
Op1         1.001000
Op2         0.101001
      +
            --------
            1.110001

Step 3: post-operation
normalization

Normalized: 1.110001 x 2^5

Rounding: 1.110001 x 2^5

Download

FINAL ANSWER: 1.110001 x 2^5
```

D. Test Case 4

Operation 1 = 1.00111101
Exponent 1 = 5
Operation 2 = 1.00111101
Exponent 2 = 3
Digits = 5
Rounding = GRS
Expected = 1.1001 x 2^5

Operation 1 = 1.00111101
Exponent 1 = 5
Operation 2 = 1.00111101
Exponent 2 = 3
Digits = 5
Rounding = Rounding
Expected = 1.1001 x 2 ^5



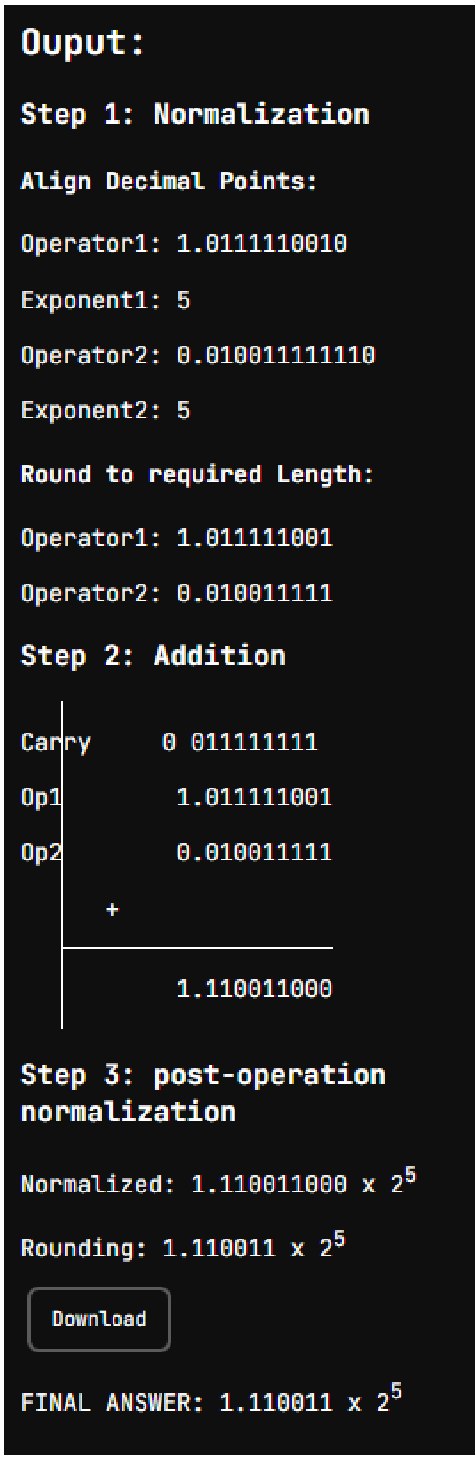
```
Ouput:

Step 1: Normalization

Align Decimal Points:

Operator1: 1.00111101

Exponent1: 5

Operator2: 0.0100111101

Exponent2: 5

Round to required Length:

Operator1: 1.0011111

Operator2: 0.0100111

Step 2: Addition

Carry     0 0111111
Op1         1.0011111
Op2         0.0100111
   +
            1.1000110

Step 3: post-operation normalization

Normalized: 1.1000110 x 2^5

Rounding: 1.1001 x 2^5

Download

FINAL ANSWER: 1.1001 x 2^5
```

```
Ouput:

Step 1: Normalization

Align Decimal Points:

Operator1: 1.00111101

Exponent1: 5

Operator2: 0.0100111101

Exponent2: 5

Round to required Length:

Operator1: 1.0100

Operator2: 0.0101

Step 2: Addition

Carry     0 0100
Op1         1.0100
Op2         0.0101
   +
            1.1001

Step 3: post-operation normalization

Normalized: 1.1001 x 2^5

Rounding: 1.1001 x 2^5

Download

FINAL ANSWER: 1.1001 x 2^5
```

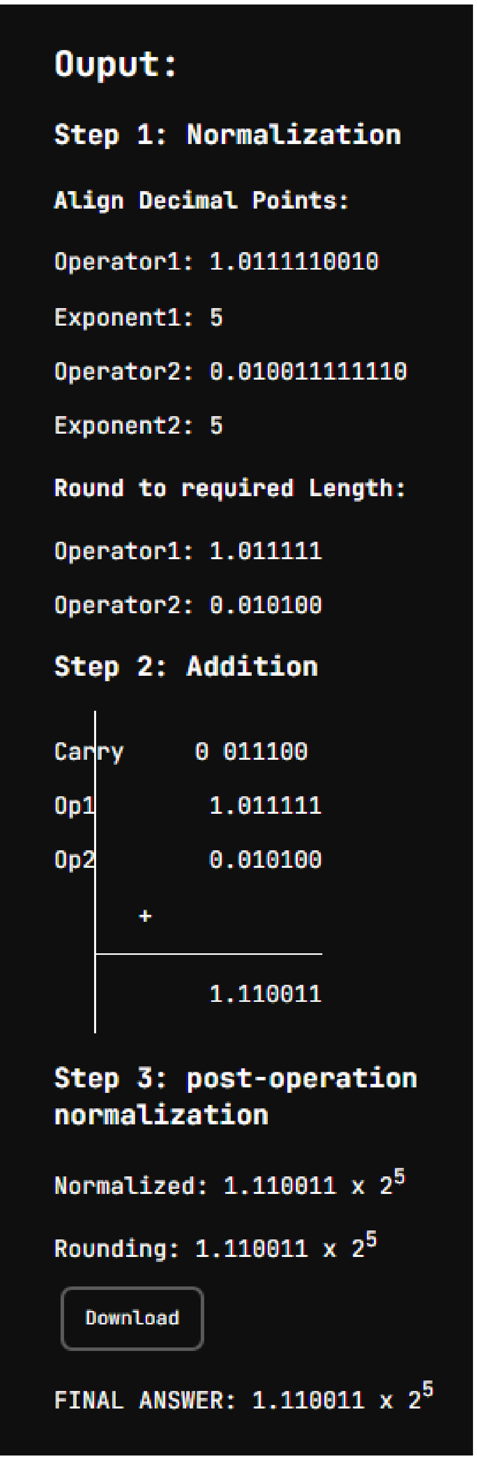
E. Test Case 5

Operation 1 = 1.0111110010
Exponent 1 = 5
Operation 2 = 1.0011111110
Exponent 2 = 3
Digits = 7
Rounding = GRS
Expected = 1.110011x2^5

Operation 1 = 1.0111110010
Exponent 1 = 5
Operation 2 = 1.0011111110
Exponent 2 = 3
Digits = 7
Rounding = Rounding
Expected = 1.110011x2^5



```
Ouput:

Step 1: Normalization

Align Decimal Points:

Operator1: 1.0111110010

Exponent1: 5

Operator2: 0.010011111110

Exponent2: 5

Round to required Length:

Operator1: 1.011111001

Operator2: 0.010011111

Step 2: Addition

Carry     0 011111111

Op1        1.011111001

Op2        0.010011111

     +
      ___________________
           1.110011000

Step 3: post-operation
normalization

Normalized: 1.110011000 x 2^5

Rounding: 1.110011 x 2^5

Download

FINAL ANSWER: 1.110011 x 2^5
```

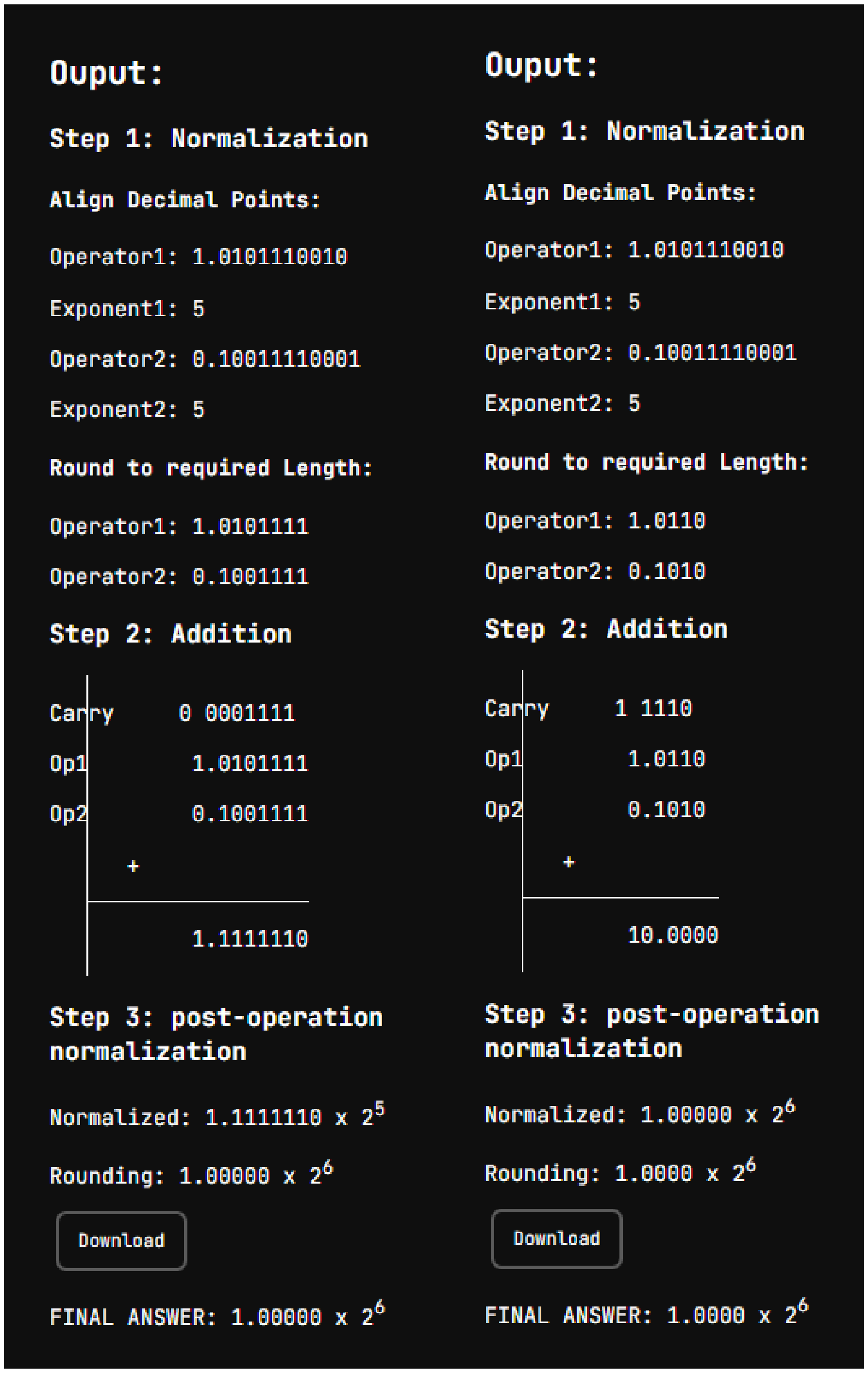


```
Ouput:

Step 1: Normalization

Align Decimal Points:

Operator1: 1.0111110010

Exponent1: 5

Operator2: 0.010011111110

Exponent2: 5

Round to required Length:

Operator1: 1.011111

Operator2: 0.010100

Step 2: Addition

Carry     0 011100

Op1        1.011111

Op2        0.010100

     +
      ________________
           1.110011

Step 3: post-operation
normalization

Normalized: 1.110011 x 2^5

Rounding: 1.110011 x 2^5

Download

FINAL ANSWER: 1.110011 x 2^5
```

F. Test case 6

Operation 1 = 1.0101110010
Exponent 1 = 5
Operation 2 = 1.0011110001
Exponent 2 = 4
Digits = 5
Rounding = GRS
Expected = 1.0000x2^6

Operation 1 = 1.0101110010
Exponent 1 = 5
Operation 2 = 1.0011110001
Exponent 2 = 4
Digits = 5
Rounding = Rounding
Expected = 1.0000x2^6



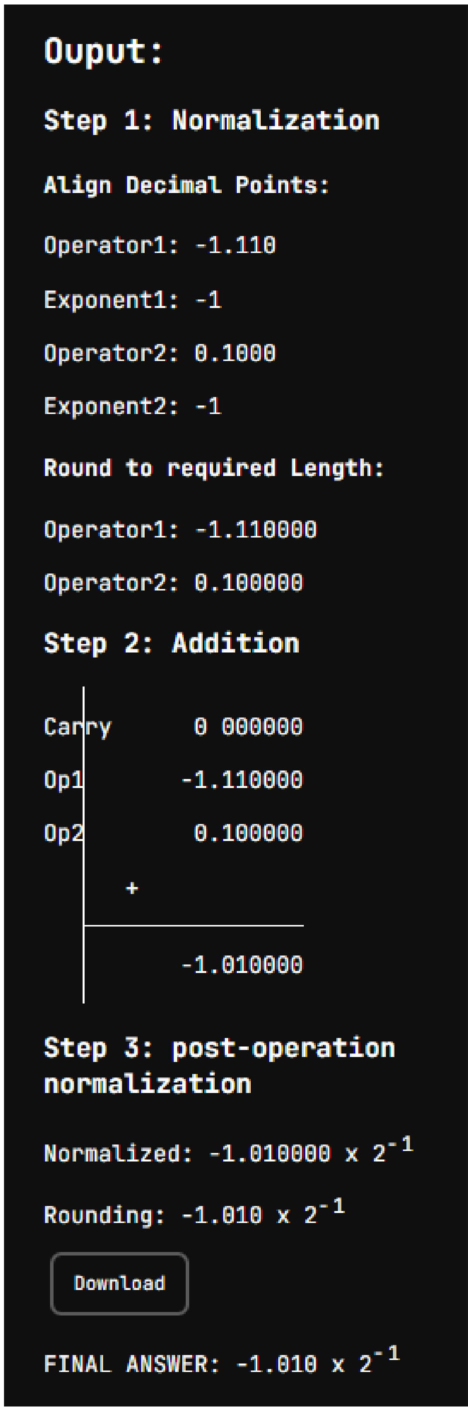
```
Ouput:

Step 1: Normalization

Align Decimal Points:

Operator1: 1.0101110010

Exponent1: 5

Operator2: 0.10011110001

Exponent2: 5

Round to required Length:

Operator1: 1.0101111

Operator2: 0.1001111

Step 2: Addition

Carry      0 0001111

Op1          1.0101111

Op2          0.1001111

      +
         ------------
             1.1111110

Step 3: post-operation
normalization

Normalized: 1.1111110 x 2^5

Rounding: 1.00000 x 2^6

Download

FINAL ANSWER: 1.00000 x 2^6
```

```
Ouput:

Step 1: Normalization

Align Decimal Points:

Operator1: 1.0101110010

Exponent1: 5

Operator2: 0.10011110001

Exponent2: 5

Round to required Length:

Operator1: 1.0110

Operator2: 0.1010

Step 2: Addition

Carry     1 1110

Op1         1.0110

Op2         0.1010

      +
         ----------
           10.0000

Step 3: post-operation
normalization

Normalized: 1.00000 x 2^6

Rounding: 1.0000 x 2^6

Download

FINAL ANSWER: 1.0000 x 2^6
```

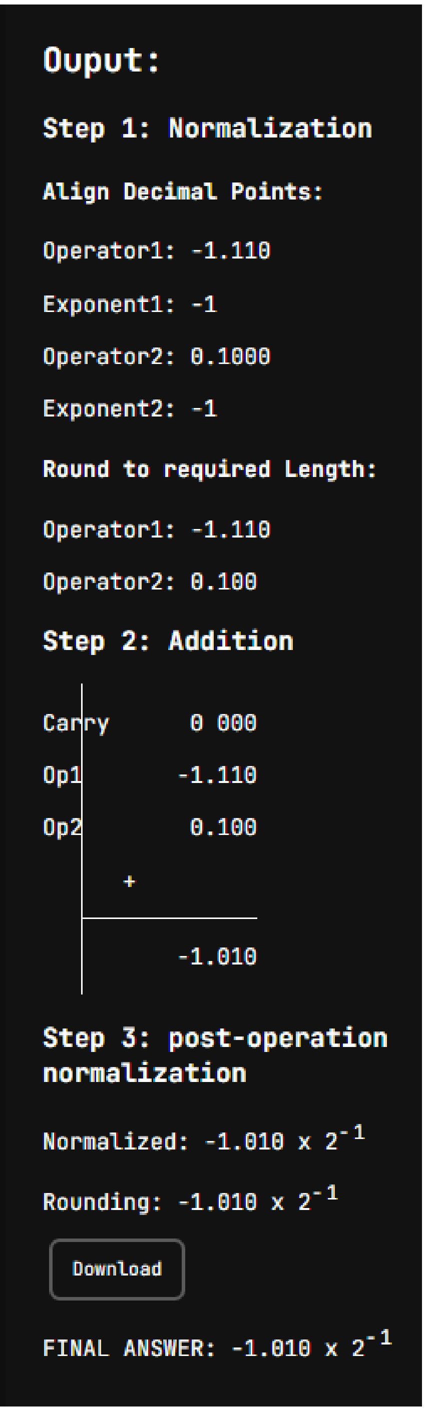
G. Test case 6

Operation 1 = -1.110
Exponent 1 = -1
Operation 2 = 1.000
Exponent 2 = -2
Digits = 4
Rounding = GRS
Expected = -1.010x2^-1

Operation 1 = -1.110
Exponent 1 = -1
Operation 2 = 1.000
Exponent 2 = -2
Digits = 4
Rounding = Rounding
Expected = -1.010x2^-1



```
Ouput:

Step 1: Normalization

Align Decimal Points:

Operator1: -1.110

Exponent1: -1

Operator2: 0.1000

Exponent2: -1

Round to required Length:

Operator1: -1.110000

Operator2: 0.100000

Step 2: Addition

Carry      0 000000

Op1       -1.110000

Op2        0.100000

     +

          -1.010000

Step 3: post-operation
normalization

Normalized: -1.010000 x 2^-1

Rounding: -1.010 x 2^-1

FINAL ANSWER: -1.010 x 2^-1
```

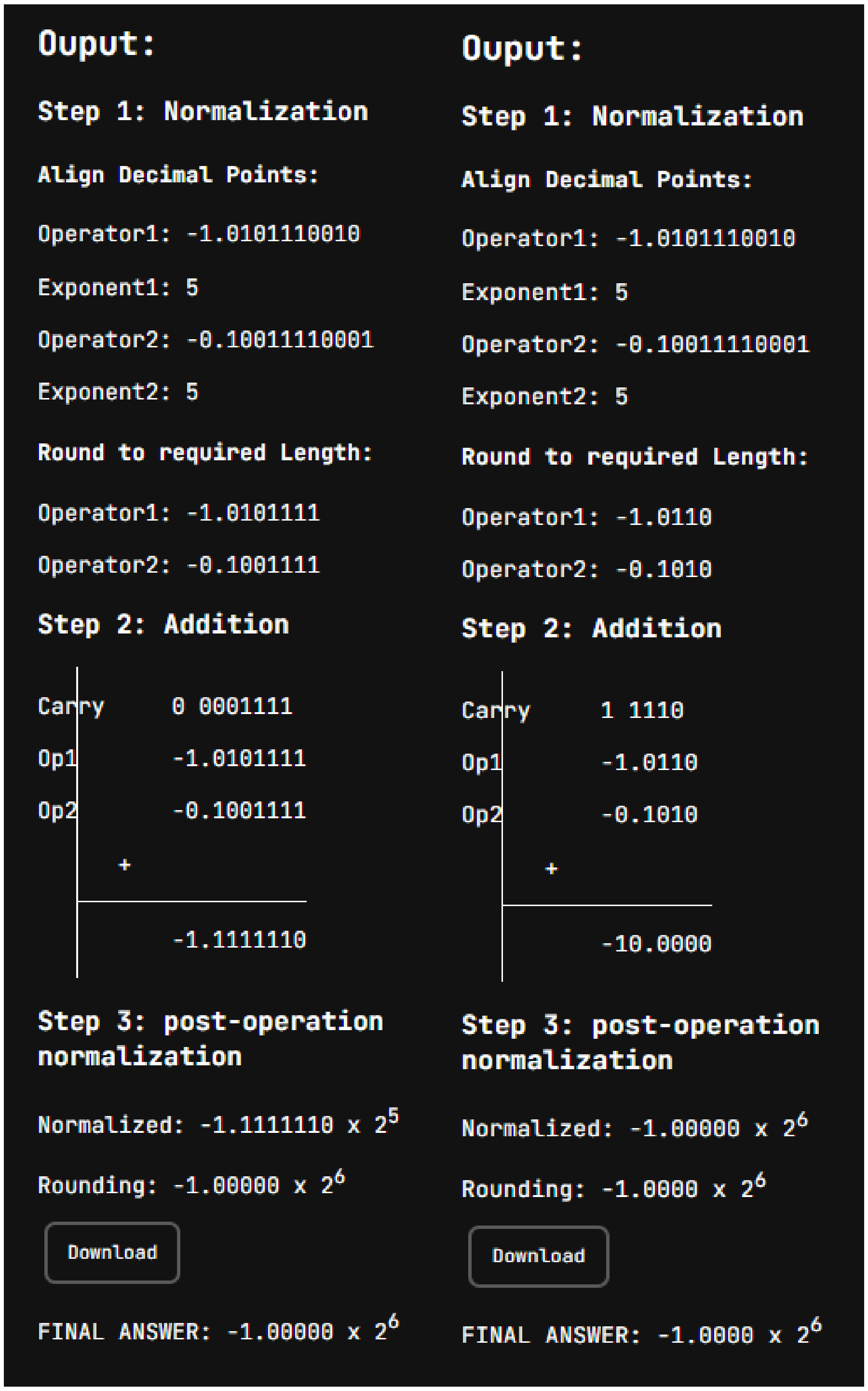


```
Ouput:

Step 1: Normalization

Align Decimal Points:

Operator1: -1.110

Exponent1: -1

Operator2: 0.1000

Exponent2: -1

Round to required Length:

Operator1: -1.110

Operator2: 0.100

Step 2: Addition

Carry      0 000

Op1       -1.110

Op2        0.100

     +

          -1.010

Step 3: post-operation
normalization

Normalized: -1.010 x 2^-1

Rounding: -1.010 x 2^-1

FINAL ANSWER: -1.010 x 2^-1
```

H. Test Case 7

Operation 1 = -1.0101110010
Exponent 1 = 5
Operation 2 = -1.0011110001
Exponent 2 = 4
Digits = 5
Rounding = GRS
Expected = -1.0000

Operation 1 = -1.0101110010
Exponent 1 = 5
Operation 2 = -1.0011110001
Exponent 2 = 4
Digits = 5
Rounding = Rounding
Expected = -1.0000



```
Ouput:

Step 1: Normalization

Align Decimal Points:

Operator1: -1.0101110010

Exponent1: 5

Operator2: -0.10011110001

Exponent2: 5

Round to required Length:

Operator1: -1.0101111

Operator2: -0.1001111

Step 2: Addition

Carry     0 0001111
Op1       -1.0101111
Op2       -0.1001111
     +
          -1.1111110

Step 3: post-operation normalization

Normalized: -1.1111110 x 2^5

Rounding: -1.00000 x 2^6

Download

FINAL ANSWER: -1.00000 x 2^6
```

```
Ouput:

Step 1: Normalization

Align Decimal Points:

Operator1: -1.0101110010

Exponent1: 5

Operator2: -0.10011110001

Exponent2: 5

Round to required Length:

Operator1: -1.0110

Operator2: -0.1010

Step 2: Addition

Carry     1 1110
Op1       -1.0110
Op2       -0.1010
     +
          -10.0000

Step 3: post-operation normalization

Normalized: -1.00000 x 2^6

Rounding: -1.0000 x 2^6

Download

FINAL ANSWER: -1.0000 x 2^6
```