K. J. Somaiya College of Engineering, Mumbai-77

Batch: B1 **Roll No.: 1711072**
Experiment / assignment / tutorial No. 5
Grade: AA / AB / BB / BC / CC / CD /DD
Signature of the Staff In-charge with date

TITLE: Implementation of IEEE-754 floating point representation

AIM: To demonstrate the single and double precision formats to represent floating point numbers.

Expected OUTCOME of Experiment:

CO 2-Detail working of the arithmetic logic unit and its sub modules

Books/ Journals/ Websites referred:

1. Carl Hamacher, Zvonko Vranesic and Safwat Zaky, "Computer Organization", Fifth Edition, TataMcGraw-Hill.
2. William Stallings, "Computer Organization and Architecture: Designing for Performance", Eighth Edition, Pearson.

Pre Lab/ Prior Concepts:

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE). The standard addressed many problems found in the diverse floating point implementations that made them difficult to use reliably and portably. Many hardware floating point units now use the IEEE 754 standard.

The standard defines:

- *arithmetic formats:* sets of binary and decimal floating-point data, which consist of finite numbers (including signed zeros and subnormal numbers), infinities, and special "not a number" values (NaNs)
- *interchange formats:* encodings (bit strings) that may be used to exchange floating-point data in an efficient and compact form

- *rounding rules:* properties to be satisfied when rounding numbers during arithmetic and conversions
- *operations:* arithmetic and other operations (such as trigonometric functions) on arithmetic formats
- *exception handling:* indications of exceptional conditions (such as division by zero, overflow, *etc*

Example (Single Precision- 32 bit representation)

$(10.25)_{10}= (1010.01)_2$

Normalization: 1.01001×2^3
Formula: E-127=3
E=130
$(130)_{10}= (10000010)_2$
Single Precision 32 bit representation:

Sign	Exponent	Mantissa
0	10000010	01001000000000000000000
1 bit	8 bits	23 bits

Example (Double Precision- 64 bit representation)

$(10.25)_{10}= (1010.01)_2$

Normalization: 1.01001×2^3
Formula: E-1023=3
E=1026
$(1026)_{10}= (10000000010)_2$
Double Precision 64 bit representation:

Sign	Exponent	Mantissa
0	10000000010	0100100000000000000000000000000000000000000000000000
1 bit	11 bits	52 bits

<u>Implementation Details (in Java):</u>

```
import java.util.*;
class Main {
    static double num, frac;
    static int sign=0, integer;
    static Vector int_bin=new Vector();
    static Vector frac_bin=new Vector();
    static int sing_prec[]=new int[32];
    static int doub_prec[]=new int[64];

    public static void main(String[] args) {
```

```
        System.out.println("IEEE-754 Notation ");
        System.out.print("Enter your number: ");
        Scanner sc=new Scanner(System.in);


        num=sc.nextDouble();
        int exp=0;
        if(num<0){
            sign=1;
            sing_prec[0]=doub_prec[0]=sign;
            num=Math.abs(num);
        }
        String[] input=String.valueOf(num).split("\\.");
        integer=Integer.parseInt(input[0]);
        frac=Double.parseDouble("0."+input[1]);
        System.out.println("Integral: "+integer+" Fraction:
"+frac);
        int integral=integer;
        double fraction=frac;
        int_binary(integral);
        frac_binary(fraction);
        int i=0;
        Iterator int_itr=int_bin.iterator();
        Iterator frac_itr=frac_bin.iterator();

        while(int_itr.hasNext()){
            if(((int)int_itr.next())==1){
                exp=i;

            }
            i++;
        }

        if(exp==0){
            while(frac_itr.hasNext()){
                exp--;
                if((int)(frac_itr.next())==1){

                    break;
                }
            }
        }
        int sing_exp=127+exp;
        int doub_exp=1023+exp;
        exp_binary(sing_exp,8);
```

```
        exp_binary(doub_exp,11);

        Vector bigvec = new Vector();
        Vector intbinrev = (Vector)int_bin.clone();
        Collections.reverse(intbinrev);
        bigvec.addAll(intbinrev);

        bigvec.addAll(frac_bin);
        int in = bigvec.indexOf(1)+1;

        int j = 0;
        for(; j<23 && j<bigvec.size()-in; j++)
        {
            sing_prec[j+9]=(int)bigvec.get(in+j);
        }
        for(;j<23;j++){
            sing_prec[j+9]=0;
        }
        j = 0;
        for(; j<52 && j<bigvec.size()-in; j++)
        {
            doub_prec[j+12]=(int)bigvec.get(in+j);
        }
        for(;j<52;j++){
            doub_prec[j+12]=0;
        }
        System.out.println("Single precision:");
        System.out.print(sign+" ");
        for(j=1;j<=31;j++){
            if(j==9){
                System.out.print(" ");
            }
            System.out.print(sing_prec[j]);
        }
        System.out.println("\nDouble precision: ");
        System.out.print(sign+" ");
        for(j=1;j<64;j++){
            if(j==12){
                System.out.print(" ");
            }
            System.out.print(doub_prec[j]);
        }
    }
    public static void int_binary(int num){
```

```
        int i=0,rem;
        while(num>0){
            rem=num%2;
            int_bin.add(i,rem);
            num/=2;
            i++;
        }
    }

    public static void frac_binary(double fraction){
        int i=0;

        while(fraction!=(double)(1) && frac_bin.size()<52){
            fraction=fraction*2;
            if(fraction>=1){
                frac_bin.add(1);
                if(fraction == 1) return;
                fraction=fraction-1;
            }
            else
                frac_bin.add(0);

        }

    }

    public static void exp_binary(int num,int length){
        int i;
        if(length==8){
            for(i=1;i<=length;i++){
                sing_prec[9-i]=num%2;
                num/=2;
            }
        }
        else if(length==11){
            for(i=1;i<=length;i++){
                doub_prec[12-i]=num%2;
                num/=2;
            }
        }
    }
}
```

For verification, code can be found at:
https://repl.it/@ARGHYADEEPDAS/COAExpt5

Output Screen:

```
IEEE-754 Notation
Enter your number:  -639.6875
Integral: 639 Fraction: 0.6875
Single precision:
1 10001000 00111111110110000000000
Double precision:
1 10000001000 0011111111011000000000000000000000000000000000000000
```

Post Lab Descriptive Questions (Add questions from examination point view)

1. **Give the importance of IEEE-754 representation for floating point numbers?**

Ans. There are several ways to represent real numbers on computers. Fixed point places a radix point somewhere in the middle of the digits, and is equivalent to using integers that represent portions of some unit. For example, one might represent 1/100ths of a unit; if you have four decimal digits, you could represent 10.82, or 00.01. Another approach is to use rationales, and represent every number as the ratio of two integers. Floating-point representation – the most common solution – uses scientific notation to encode numbers, with a base number and an exponent. For example, 123.456 could be represented as 1.23456×10^2. In hexadecimal, the number 123.abc might be represented as $1.23abc \times 16^2$. In binary, the number 10100.110 could be represented as 1.0100110×2^4.

Floating-point solves a number of representation problems. Fixed-point has a fixed window of representation, which limits it from representing both very large and very small numbers. Also, fixed-point is prone to a loss of precision when two large numbers are divided. Floating-point, on the other hand, employs a sort of "sliding window" of precision, appropriate to the scale of the number. This allows it to represent numbers from 1,000,000,000,000 to 0.0000000000000001 with ease, and while maximizing precision (the number of digits) at both ends of the scale.

While representing floating point numbers, we can face either precision or accuracy loss. To overcome these issues, IEEE came up with a way to represent floating type numbers for complex computations; known as the IEEE-754 format.

Conclusion: The IEEE-754 representation of floating point numbers has been done successfully and has been verified for various test cases.

Date: 22/08/2018 **Signature of faculty in-charge**