DOI: 10.14421/ijid.2020.09103 IJID (International Journal on Informatics for Development), *e-ISSN: 2549-7448*
Vol. 9, No. 1, 2020, Pp. 15-20

Integer Representation of Floating-Point Manipulation with Float Twice

Wakhid Kurniawan[1], Hafizd Ardiansyah[2], Annisa Dwi Oktavianita[3], Mr. Fitree Tahe[4]
[1, 2, 3, 4]Student of Department of Informatics
Informatics Graduate Program, Faculty of Science and Technology
UIN Sunan Kalijaga Yogyakarta
[1]PT.Cobra Dental Indonesia
[3]PT. AVO Innovation Technology
[1]19206050003@student.uin-suka.ac.id, [2]19206050002@student.uin-suka.ac.id, [3]19206050019@student.uin-suka.ac.id

Article History
Received June 9th, 2020
Revised July 1st, 2020
Accepted July 28th, 2020
Published July, 2020

Abstract—**In the programming world, understanding floating point is not easy, especially if there are floating point and bit-level interactions. Although there are currently many libraries to simplify the computation process, still many programmers today who do not really understand how the floating point manipulation process. Therefore, this paper aims to provide insight into how to manipulate IEEE-754 32-bit floating point with different representation of results, which are integers and code rules of float twice. The method used is a literature review, adopting a float-twice prototype using C programming. The results of this study are applications that can be used to represent integers of floating-point manipulation by adopting a float-twice prototype. Using the application programmers make it easy for programmers to determine the type of program data to be developed, especially those running on 32 bits floating point (Single Precision).**

Keywords-float; bit_manipulation; integer; c_programming; float_twice

1 INTRODUCTION

Computers encode information in the form of bits, generally arranged in bytes. Different encodings are used for representation of integers, real numbers, and character strings. Different computer models use different conventions for encoding numbers and for ordering bytes in multi-byte data [1][2]. In a book states that until the 1980s, each computer manufacturer made a convention for how to represent floating-point numbers and detailed the operations carried out by them. In addition, they are less concerned about the accuracy of operations, seeing speed and ease of implementation as more important than numerical precision.

IEEE Standard 754 floating point is the most common representation standard for real numbers on computers today, which is used in Intel-based PC, Macintosh, and most Unix platforms are not just limited to floating point PCs already widely used on microcontrollers that are widely used for systems embedded [3][4][5]. This standard is also a product of the Floating-Point Working Group, and is sponsored by, the Microprocessor Standards Committee of the IEEE Computer Society. This standard gives order to carry out floating-point calculations that give results, regardless of whether the processing is carried out in hardware, software, or a combination of both. In a programming environment, this standard is also used to form the basis of a dialogue between numerical collections and programming language designers.

Therefore, it is hoped that language designers will see a series of operations, precision and exception controls, all explained here as a guide for programmers with easily acceptable control expressions and exceptions. In addition, it is hoped that designers will be guided by this standard in order to provide fully acceptable extensions. In a book, explaining his treatment of this material is based on a series of mathematical principles. They start with the basic definition of encoding and then acquire some properties as ranges of numbers that can be represented, bit-level representations and some arithmetic operation properties. They believe that it is important to examine material from the standpoint that this abstract, because the programmer must have a clear understanding of how the computer arithmetic generally associated with integer and real arithmetic[1].

Language C was designed to accommodate a variety of different implementations in terms of word size and numeric encoding. Understanding this encoding at the bit level, as well as understanding the mathematical characteristics of arithmetic operations, is very important for the programmer to operate properly in the full range of numerical values [6][7][8]. Most machines encode signed numbers using a two's-complement representation and encode floating-point numbers using IEEE Standard 754. In a book, understanding the understanding of bit-level coding, and understanding the mathematical characteristics of arithmetic operations is important to make the program operate in accordance with the range of numeric values [1].

The C ++ programming language created by Bjarne Stroustrup is a derivative of the C programming language developed at Bong Labs (Dennis Ritchie) in the early 1970s. The language was derived from the previous language, namely B, at first; the language was designed as a programming language that is run on UNIX systems. The language uses the exact same numerical representations and operations. Everything discussed about C in this section applies also to C ++. Standard C is indeed designed for implementation in a variety of possibilities.

There are two types of storage layouts in IEEE floating point 754, namely single (32 bit) and double (64 bit) [9][10]. Table 1 presents the IEEE floating point 754 32 bit and 64 bit standard components [11][12]. However, this paper focuses on manipulate Floating-Point 32-bit (single-precision).

Now many programmers do not understand how to make a program that uses light memory. Increasing number of plugins, bundles of function, Framework simplify writing encoding, without the programmer knowing the origin of the process or data representation of the various types of data.

The purpose of this paper is to become more familiar with integer representations of floating-point numbers, especially on float-twice. The programmer will easily solve a series of programming. Although many of these puzzles are quite artificial, readers will find more knowledge about bits in doing them. The results of this study are applications that can represent integers of floating-point manipulation based on the float-twice prototype using the C programming language [1].

1.1 *Integer Representation*

This section explains two ways of bit, which can be used for encoding integers, the first integer representing non-negative numbers and the second integer representing negative, zero and positive numbers [1].

1.2 *IEEE Standard 754 Floating Point*

All changed when the emergence of the IEEE Standard 754 in the 1985s, standards and operations that were carefully crafted to represent floating-point numbers. This effort began in 1976 under Intel sponsors with the design of 8087, a chip that provides floating-point support for 8086 processors [1][7].

The format for floating point data operations depends on the number of bits in the data. They are named half precision for 16 bits, single precision for 32 bits, double precision for 64 bits and Quadruple precision for 128 bits as shown in Table 1. They are divided into 3 parts as Sign bits, Exponents and Mantissa [12][13][14][15] [16][17].

Table 1 Table Type Styles

	Sign	**Exponent**	**Fraction**
Single Precision	1 [31]	8 [30–23]	23 [22–00]
Double Precision	1 [63]	11 [62–52]	52 [51–00]

1.2.1 *Storage Layout*

1.2.1.1 *Sign:* At the marker the value of 0 indicates a positive number, while the value of 1 indicates a negative number. So what if reversing the bit value, the value will be adjusted again.

1.2.1.2 *Exponent:* The exponent field is used to represent positive or negative exponents. To do this, the bias is added to the actual exponent to get the exponent to be saved. In IEEE single-precision, the bias value is 127. This means that the zero exponent 127 is stored in the exponent field. For example, if the stored value is 200, then it shows that the exponent (200-127) or equal to 73. While the value of 8 on the exponent indicates that the exponent field in single precision is 8 bits. For double precision, the exponent field is 11 bits, and has a bias of 1023.

1.2.1.3 *Mantissa/Fraction:* Mantissa, also known as significand is used to represent the precision bits of a number. Mantissa consists of an implicit main bit and a fraction bit.

1.2.2 *Special Value*

IEEE reserves the exponent field values of all 0 and all 1 to show special values in the floating-point scheme [1]. How to determine whether the number is normalized, denormalized, or special value, can be seen in Figure 1.

1.2.2.1 *Denormalize*

If all exponents are 0, but the fraction is not 0 (if it will not be interpreted as zero), then the value is a denormalizer number, which does not have a main assumption of 1 before the binary point. So, this represents numbers $(-1)^s \times 0.f \times 2^{-126}$, where s is the sign bit and f is a fraction. For double precision, normalized numbers are shaped $(-1)^s \times 0.f \times 2^{-1022}$. From here, you can interpret zero as a special type of number that is normalized.

1.2.2.2 *Zero*

As noted above, zeros cannot be directly represented in a straight format, because assumption 1 is leading, (we need to determine the true zero mantissa to produce a zero value). Zero is a special value denoted by the zero exponent field and the zero fraction field. Note that -0 and +0 are different values, even though they are the same [18].

1.2.2.3 *Infinity*

The value (+) of negative infinity and (-) positive infinity are denoted by all exponents having a value of 1 and all fractions of value 0. A bit sign distinguishes between negative infinity and positive infinity. Being able to show an infinite value as a specific value, because it allows operations to proceed through an overflow situation. Operations with unlimited values are well defined in IEEE floating point [18].

1.2.2.4 *Not A Number*

Not a Number value used to represent values that do not represent real numbers. NaN is represented by the exponent of all non-faction [18].

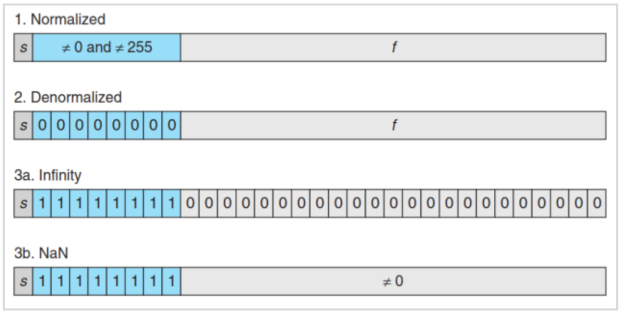


Figure 1. Category of single-precision floating-point values.

1.3 *Float Twice*

Float twice is a concept for IEEE 754 floating point bit-manipulation, following the rules of the bit-level floating-point code, carrying out the function following the prototype is shown in Figure 2. To sum up, the following in Table 2 are the corresponding values for a given representation.

Table 2 Float Values

Sign	Exponent (e)	Fraction (f)	Value
0	00···00	00···00	+0
0	00···00	00···01 ⋮ 11···11	Positive Denormalized Real $0.f \times 2^{(-b+1)}$
0	00···01 ⋮ 11···10	XX···XX	Positive Normalized Real $1.f \times 2^{(e-b)}$
0	11···11	00···00	+∞
0	11···11	00···01 ⋮ 01···11	SNaN
0	11···11	1X···XX	QNaN
1	00···00	00···00	−0
1	00···00	00···01 ⋮ 11···11	Negative Denormalized Real $-0.f \times 2^{(-b+1)}$
1	00···01 ⋮ 11···10	XX···XX	Negative Normalized Real $-1.f \times 2^{(e-b)}$
1	11···11	00···00	−∞
1	11···11	00···01 ⋮ 01···11	SNaN
1	11···11	1X···XX	QNaN

2 METHOD

This paper uses the literature review method from several literatures, journals, e-books, webpages, and proceedings. In addition, this paper also follows the rules for coding float twice of bit-level floating-point, in its implementation following the prototype shown in Figure 2.

2.1 *Literature Review*

Literature review or literature review is written material in the form of books, journals that discuss topics to be studied. From the opinions of several experts, in general it can be concluded that the literature review is a description or description of the literature that is relevant to the particular field or topic to be examined [19][20].

2.2 *Prototype of Float Twice*

In this study following is the prototype of float twice as shown in Figure 2. The float twice formula is as in Formula (1).

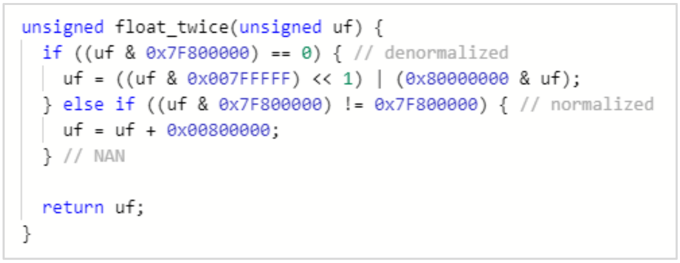
```
/* Calculation formula 2 * f. If f is NaN (Not a Number),
then return f.*/
float_bits float_twice(float_bits f);
```

Figure 2. Prototype of float twice

$$2^*f. \tag{1}$$

3 RESULT AND DISCUSSION

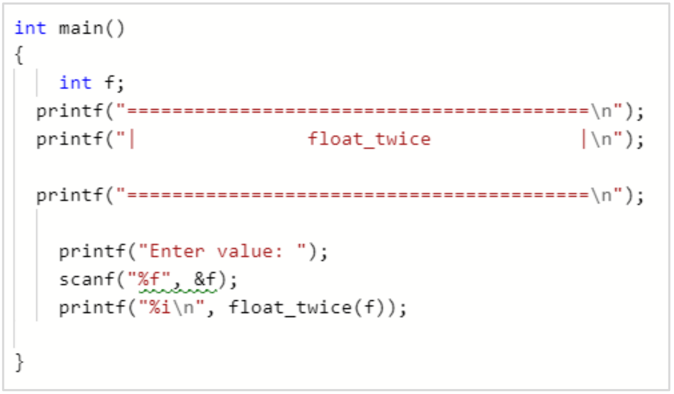
This session presents the results of the float twice implementation and the discussion, which in subbab 1.1.2. presents the process of denormalization, normalization, and NaN (Not a Number) or numeric data type values that represent undetermined or under-represented values, especially in floating-point arithmetic. The first thing to do was to create a function, as shown in Figure 3 below.



```
unsigned float_twice(unsigned uf) {
  if ((uf & 0x7F800000) == 0) { // denormalized
    uf = ((uf & 0x007FFFFF) << 1) | (0x80000000 & uf);
  } else if ((uf & 0x7F800000) != 0x7F800000) { // normalized
    uf = uf + 0x00800000;
  } // NAN

  return uf;
}
```

Figure 3. Function of float twice

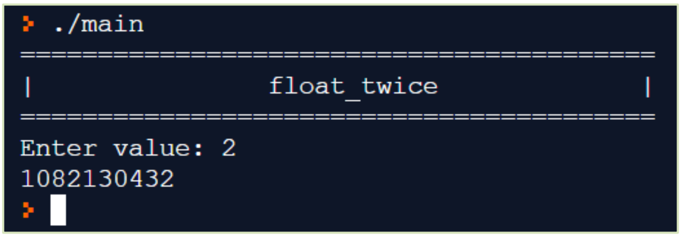
Then we made a program like in Figure 4 below, but before calling the function, do not forget to #include <stdio.h> so the program can run correctly.



```
int main()
{
    int f;
  printf("=========================================\n");
  printf("|               float_twice             |\n");

  printf("=========================================\n");

    printf("Enter value: ");
    scanf("%f", &f);
    printf("%i\n", float_twice(f));

}
```

Figure 4. Call float twice function

The input is of integer type, but assumed to be float. Then do the float twice operation of the input. Therefore, the result is an integer from the bit representation of the float number from the float twice. Look at Figure 5. Result C programming below.



```
> ./main
=========================================
|               float_twice             |
=========================================
Enter value: 2
1082130432
>
```

Figure 5. Result of C program

Furthermore, how the calculation process in this program is presented as follows. The calculation result of float twice from input 2 is 4. Normalization of binary representation of number 4, shifts the position of decimal point 2 to the left so that only one non-zero digit is left on the left:

$4_{(10)} =$

$100_{(2)} =$

$100_{(2)} \times 2^0 =$

$1.00_{(2)} \times 2^2$

In this position, there are the following elements that will be included in 32-bit single-precision IEEE 754 binary floating-point representation:

Sign: 0 (a positive number)

Exponent (unadjusted): 2

Mantissa (not normalized): 1.00

Adjust the exponent in the 8 bits excess / bias notation and then change from decimal (base 10) to 8 binary bits, using the same technique to divide it repeatedly with 2:

Exponent (adjusted) = Exponent (unadjusted) + $2^{(8-1)}$ - 1 = 2 + $2^{(8-1)}$ - 1 = $(2 + 127)_{(10)} = 129_{(10)}$

Division = Quotient + Remainder;

129 ÷ 2 = 64 + 1;

64 ÷ 2 = 32 + 0;

32 ÷ 2 = 16 + 0;

16 ÷ 2 = 8 + 0;

8 ÷ 2 = 4 + 0;

4 ÷ 2 = 2 + 0;

2 ÷ 2 = 1 + 0;

1 ÷ 2 = 0 + 1;

Exponent (adjusted) = $129_{(10)} = 10000001_{(2)}$

Conclusion $4_{(10)}$ = 0-10000001-00000000000000000000000

So the integer representation of binary

0-10000001-00000000000000000000000 is **1082130432.**

Table 3 Floating-point IEEE-754 Format

Sign	0
(1 bit):	31
Exponent	1 0 0 0 0 0 0 1
(8 bits):	30 29 28 27 26 25 24 23
Mantissa:	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
(23 bits)	22 21 20 19 18 17 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

4 CONCLUSION

Based on the results of this research, a C programmer, in order to manipulate bits of floating-point numbers, should first understand the binaries and standard components of the IEEE Floating Point-754 Single Precision 32 bits, understand the conversion of data types at C. Our approach to do research described in this paper is started from the input in the form of an integer which is assumed to be a float. Then we produce an integer representation, which can be used by the programmer to see the integer representation of a value. Therefore, in the future the programmer can determine the use of data types that run on 32 bits (single precision) correctly. Either suggestions for future researchers is to explore IEEE Floating Point-754 Double Precision 32-bit or it can be other bit-level floating-point such as *i2f*, *f2i*. All of these could be seen as the production of knowledge about the basis of programming.

ACKNOWLEDGMENT

This paper is one of the assignment from Computer Systems and Organization course, the Department of Informatics Graduate Program, Faculty of Science and Technology UIN Sunan Kalijaga Yogyakarta.

REFERENCES

[1] R. E. Bryant and D. R. O. Hallaron, *Computer Systems. A Programmer's Perspective [3rd ed.]*. Boston: Pearson, 2016.

[2] M. Drumond, T. Lin, B. Falsafi, and M. Jaggi, "Training Dnns with hybrid block floating point," *Adv. Neural Inf. Process. Syst.*, vol. 2018-Decem, no. NeurIPS, pp. 453–463, 2018.

[3] S. Janakiraman, K. Thenmozhi, J. B. B. Rayappan, and R. Amirtharajan, "Lightweight chaotic image encryption algorithm for real-time embedded system: Implementation and analysis on 32-bit microcontroller," *Microprocess. Microsyst.*, vol. 56, pp. 1–12, 2018.

[4] Y. P. You, T. C. Lin, and W. Yang, "Translating AArch64 floating-point instruction set to the x86-64 platform," *ACM Int. Conf. Proceeding Ser.*, 2019.

[5] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou, and X. Ji, "High-performance fpga-based cnn accelerator with block-floating-point arithmetic," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 27, no. 8, pp. 1874–1885, 2019.

[6] U. Sidi, M. Ben Abdellah, M. Fez, D. Chenouni, M. Berrada, and A. Tahiri, "Paper—A Serious Game for Learning C Programming Language Concepts Using Solo Taxonomy A Serious Game for Learning C Programming Language Concepts Using Solo Taxonomy Alaeeddine Yassine," *iJET*, pp. 110–127, 2017.

[7] A. Sanchez-Stern, P. Panchekha, S. Lerner, and Z. Tatlock, "Finding root causes of floating point error," *ACM SIGPLAN Not.*, vol. 53, no. 4, pp. 256–269, 2018.

[8] C. Series, "Analysis and Research of Sorting Algorithm in Data Structure Based on Analysis and Research of Sorting Algorithm in Data Structure Based on C Language," 2020.

[9] S. Smith, *Programming with 64-Bit ARM Assembly Language*. Canada: Apress, 2020.

[10] K. D. Rao, P. V. Muralikrishna, and C. Gangadhar, "FPGA Implementation of 32 Bit Complex Floating Point Multiplier Using Vedic Real Multipliers with Minimum Path Delay," *2018 5th IEEE Uttar Pradesh Sect. Int. Conf. Electr. Electron. Comput. Eng. UPCON 2018*, pp. 1–6, 2018.

[11] J. J. J. Nesam and S. Sivanantham, "An area-efficient 32-bit floating point multiplier using hybrid GPPs addition," *2017 Int. Conf. Microelectron. Devices, Circuits Syst. ICMDCS 2017*, vol. 2017-Janua, pp. 1–4, 2017.

[12] A. Burud and P. Bhaskar, "Design and Implementation of FPGA Based 32 Bit Floating Point Processor for DSP Application," *Proc. - 2018 4th Int. Conf. Comput. Commun. Control Autom. ICCUBEA 2018*, no. ref 15, pp. 1–5, 2018.

[13] P. Lindstrom, S. Lloyd, and J. Hittinger, "Universal coding of the reals: Alternatives to IEEE floating point," *ACM Int. Conf. Proceeding Ser.*, no. March, 2018.

[14] I. Corporation, "[3]Intel ® 64 and IA-32 Architectures Software Developer ' s Manual Documentation Changes," *System*, vol. 3, no. 253665, 2011.

[15] C. R. S. Hanuman and J. Kamala, “Hardware Implementation of 24-bit Vedic Multiplier in 32-bit Floating-Point Divider,” *2018 4th Int. Conf. Electr. Electron. Syst. Eng. ICEESE 2018*, pp. 60–64, 2018.

[16] A. M. San and A. N. Yakunin, “Hardware implementation of floating-point operating devices by using IEEE-754 binary arithmetic standard,” *Proc. 2019 IEEE Conf. Russ. Young Res. Electr. Electron. Eng. ElConRus 2019*, pp. 1624–1630, 2019.

[17] R. Watpade and P. Palsodkar, “BSD adder for floating point arithmetic: A review,” *Proc. 2017 IEEE Int. Conf. Commun. Signal Process. ICCSP 2017*, vol. 2018-Janua, pp. 553–556, 2018.

[18] L. Kamble, P. Palsodkar, and P. Palsodkar, “Research trends in development of floating point computer arithmetic,” *Proc. 2017 IEEE Int. Conf. Commun. Signal Process. ICCSP 2017*, vol. 2018-Janua, no. April, pp. 329–333, 2018.

[19] C. R. Semiawan, *Metode Penelitian Kualitatif: Jenis, Karakteristik dan Keunggulannya*. Jakarta: Grasindo, 2010.

[20] S. A. Bawankar and P. G. D. Korde, “Review on 32 bit single precision Floating point unit (FPU) Based on IEEE 754 Standard using VHDL,” *Int. Res. J. Eng. Technol.*, vol. 4, no. 2, pp. 1077–1082, 2017.