Bachelor´s Thesis

Generating word-level floating-point benchmarks

Robin Trüby

Examiner: Prof. Dr. Armin Biere

University of Freiburg

Department of Computer Science

Chair of Computer Architecture

January 04th, 2023

Author

Robin Trüby

Matriculation Number

4709886

Writing Period

29. 11. 2022 – 29. 02. 2023

Examiner

Prof. Dr. Armin Biere

Advisers

Prof. Dr. Armin Biere, Mathias Fleury

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date	Signature

Abstract

Zusammenfassung

German version is only needed for an undergraduate thesis.

Inhaltsverzeichnis

1 Introduction **1**

1.1 Motivation 1

2 Background **3**

2.1 Floating-point 3

2.1.1 Definition 3

2.1.2 Representation 4

2.1.3 Subnormal numbers 5

2.1.4 Special floating-point numbers 6

2.2 Rounding floating-point numbers 7

2.3 Addition of floating-point numbers 8

2.4 SMT-Solver 9

2.5 SMTLIB2 10

2.6 First-order logic 10

2.7 SAT-Solver 10

2.8 CNF 10

3 Approach **11**

3.1 Problem Definition 11

3.2 First Part of the Approach 11

3.3 N-th Part of the Approach 11

4 Results **13**

5 Conclusion **15**

Bibliography **17**

Abbildungsverzeichnis

1 Above normal floating-point only. b-a cannot be represented. Below: the subnormal numbers are included and they can represent b-a [1]. . 6

2 The four rounding modes for the two positiv numbers x and y [2]. . . 8

Tabellenverzeichnis

1 Sizes of various fields in the formats and values of the exponent bias from different floating-point types. 4
2 Binary encoding of various floating-point data in single precision. . . 6

List of Algorithms

1 Introduction

1.1 Motivation

In a world where almost every part of our daily life is affected by electronic devices or augmented intelligence people are either eager to see what the future holds or scared of lose control over technology. We are still a big step away from robots replacing people, but in the next few years we will see self-driving cars on our streets. To verify that these cars behave we need to make sure the software and hardware is bug free and reliable. Satisfiabality Modulo Theories Solver, short SMT-Solver, play a big role in automated verification of hardware and software designs. SMT-Solver generalizes the Boolean satisfiability problem (SAT) to more complex formulas. Instead of taking Boolean variables as input SMT-Solver takes a formula expressed in first-order logic and provides different first-order theories like, for example integer numbers, arrays, strings, bit vectors and floating-point numbers. In this thesis we are taking a closer look at floating-point arithmetic and their representation by only using the quantifier free bit vector (QF BV) logic. Especially we are looking at the addition of floating-point numbers and their commutativity. So do so we created benchmarks to compare different state of the art bit-vector SMT-Solver such as Z3, cvc5 and boolector. The thesis is organized as follows: we first give some information and definitions about the theoretical background of the topics involved, e.g floating-point, addition of floating point and SMT-Solver. After that, I am going to explain my algorithm for adding floating-point numbers. In the end my results and a conclusion are represented.

2 Background

2.1 Floating-point

2.1.1 Definition

In computer science, a floating-point number is a digital representation of a real number that can support a wide range of values. This is in contrast to a fixed-point number, which has a fixed number of digits to the right of the decimal point. The most common format for floating-point numbers are defined in the IEEE 754 standard. The two most used formats are binary32 (single-precision) and binary (double precision). These formats are characterized by the number of bits used to represent the number. A radix-β floating-point number x is a number of the form [1]

$$x = (-1)^s \cdot m \cdot \beta^e. \tag{1}$$

s is the sign bit of x. It is either 1 (negative) or 0 (positive). m is the *normal significand*. It has one number before the radix point. After the radix point are at most p-1 digits, whereby p is the *precision* of the number. Note, that only p-1 bits are stored (1), because in radix 2 the first digit, the one in front of the radix point, is a 1 for normal floating-point numbers and a 0 for subnormal numbers [1]. The not stored leftmost bit is called the hidden bit. As previously mentioned β is the radix. In this thesis the default value and only used radix is 2. The last part of the formula is e the exponent. In floating-point representations the exponent is stored with a bias, the so called

exponent bias. The biasing allows to have negative exponents without adding a sign bit to the number [3]

$$e_{real} = e_{stored} - bias. \tag{2}$$

In the IEEE 754-2008 standard the range of the exponent is defined as $e_{max} = bias$ and $e_{min} = 1 - bias$.

2.1.2 Representation

Depending on the floating-point format different amount of storage is needed. In table 1 you can see different floating-point formats.

A binary16 floating-point number is structured as follows:

$$x = 0|10011|1001100111 \tag{3}$$

The first bit, the sign bit is 0, which indicates that $x \geq 0$. The next part of the number is the *biased exponent*. It is neither 00000_2 nor 11111_2, that means it is a normal number. To get the real exponent we need to subtract the bias

$$10011_2 = 19_{10} \tag{4}$$

$$e_{stored} - bias = 19_{10} - 15_{10} = 4_{10} \tag{5}$$

Format	word size	sign	exponent W_E	significand *p-1*	exponent bias b
bfloat8	8	1	4	3	7
bfloat16	16	1	8	7	127
binary16	16	1	5	10	15
binary32	32	1	8	23	127
binary64	64	1	11	52	1023
binary128	128	1	15	112	16384

Tabelle 1: Sizes of various fields in the formats and values of the exponent bias from different floating-point types.

After calculating the real exponent we need to calculate the significand. Because only the trailing significand is stored and it is a normal number, we need to add 1 before the radix point.

$$.1001100111 \tag{6}$$

$$1.1001100111 \tag{7}$$

$$1.1001100111_2 = 1,6005859375_{10}. \tag{8}$$

Hence, x is equal to

$$x = (-1)^0 \cdot 1,6005859375_{10} \cdot 2^4 \tag{9}$$

$$x = 16005,859375_{10}. \tag{10}$$

2.1.3 Subnormal numbers

Subnormal numbers are floating-point numbers that are close to zero. They have been included in the IEEE 754-1985 standard [3]. The benefit of subnormal is illustrated in 1. When adding subnormal numbers, an subtraction of two different normal or subnormal floating-point numbers can never be 0. The difference between normal and subnormal numbers is that instead of adding a 1 in front of the radix point of the *normal significand* you add a 0. Additional to that the exponent is calculated different

$$e_{denormal} = e_{stored} - bias + 1 = e_{min}. \tag{11}$$

Subnormal numbers are represented with only zeros in the exponent. An example is shown in table 2.

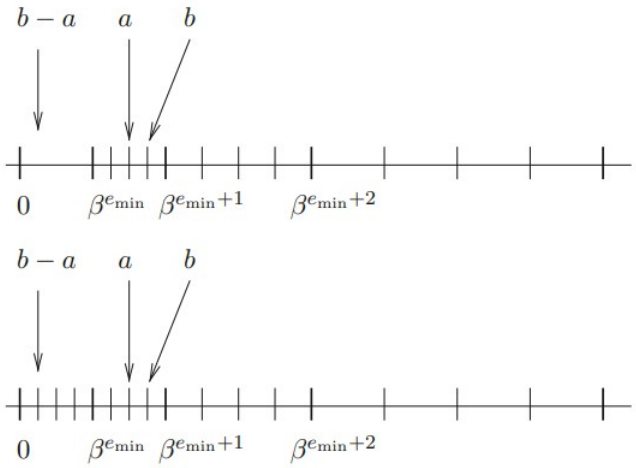

Abbildung 1: Above normal floating-point only. b-a cannot be represented. Below: the subnormal numbers are included and they can represent b-a [1].

Datum	Sign	Biased exponent	Trailing significand
-0	1	00000000	00000000000000000000000
+0	0	00000000	00000000000000000000000
$-\infty$	1	11111111	00000000000000000000000
$+\infty$	0	11111111	00000000000000000000000
qNaN	0	11111111	1xxxxxxxxxxxxxxxxxxxxxx
sNaN	0	11111111	0 nonzero string
Subnormal	0/1	00000000	nonzero string
5	0	10000001	01000000000000000000000

Tabelle 2: Binary encoding of various floating-point data in single precision.

2.1.4 Special floating-point numbers

In the IEEE 754 standard, there are different representations for special numbers, including NaN, infinity, and zero. Zero is signed, meaning there is both a positive and a negative zero. The standard also includes two types of NaNs, signaling NaNs (sNaNs) and quiet NaNs (qNaNs) [2]. Signaling NaNs appear when there are invalid operations or uninitialized variables. They raise a exception.Quiet NaNs are returned when there are invalid arithmetic operations. For Example $\frac{0}{0}$ = qNaN and $\sqrt{-2}$ = qNaN return a quiet NaN. Table 2 shows the different implementations of these special values.

2.2 Rounding floating-point numbers

Floating-point arithmetic can be subject to rounding errors and other inaccuracies, due to the limited precision of the mantissa. In these cases, it is important that we have a proper rounding function. The four main rounding modes mentioned in the IEEE 754-2008 are [2]:

- round toward $-\infty$: RD(x) is the largest floating-point number less than or equal to x;

- round toward $+\infty$: RU(x) is the smallest floating-point number greater than or equal to x;

- round toward zero: RZ(x) is the closest floating-point number to x that is no greater in magnitude than x;

- round to nearest: RN(x) is the closest floating-point number possible to x. If x is exactly halfway between to consecutive floating-point the rule *round to the nearest even* is applied. That means x is rounded to the one whose integral significand is even. Thus, for example, +7,5 becomes +8, as does +8,5 [4];

In the IEEE 754-2008 Standard *round to nearest even* is the default mode. *Round to nearest even* is also the rounding mode used in the software for this thesis. In figure2, the four rounding modes are illustrated. In order to round, three further bits are required in addition to the normal bits of the Mantissa. The Guard, the Round and the Sticky Bit. The Guard and the Round bit follow directly after the last stored bit. The Sticky Bit is an OR for the remaining bits that are removed.

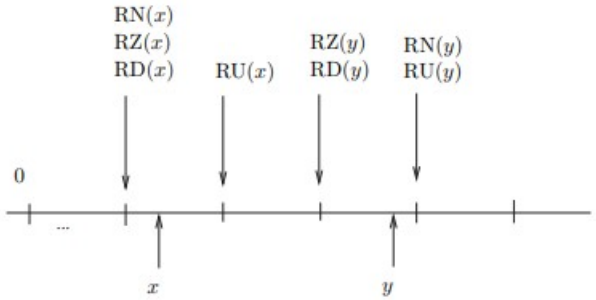

Abbildung 2: The four rounding modes for the two positiv numbers x and y [2].

2.3 Addition of floating-point numbers

The addition of real numbers has several well-known properties like commutativity, associativity and distributivity. When using floating-point addition associativity and distributivity are lost. However, when the arithmetic operations are correctly rounded floating-point addition remains commutative [1]. If $\circ$ is the rounding function

$$\circ(a + b) = \circ(b + a) \tag{12}$$

is true for all floating-point numbers a and b.
To explain how floating-point numbers are added lets regard the example of the two float16 numbers

$$x = 0|1011|100 \tag{13}$$

$$y = 0|0111|011. \tag{14}$$

First, the two numbers must be brought to the same exponent. To do so the smaller exponent need to be subtracted from the bigger one

$$1011_2 - 0111_2 = 0100_2 = 4_{10}. \tag{15}$$

After calculating the difference the mantissa of the smaller number can be shifted. When shifting it is important to add the hidden bit. After shifting the two mantissa can be added

$$1.100000_2 + 0.000101_2 = 1.100101_2. \tag{16}$$

Note even though the stored mantissa is only 3 bits long, 7 bits are used for the calculation. One extra bit for the hidden bit and 3 for rounding (GRS-Bits). After adding the number must be rounded proper

$$1.100101_2 = 1.101_2. \tag{17}$$

Putting everything together

$$x + y = 0|1011|101. \tag{18}$$

2.4 SMT-Solver

maybe short description and declaration?
describe eager lazy approach?
explain how for example z3 works?

Do you think the next sections are necessary or are the above enough?

2.5 SMTLIB2

2.6 First-order logic

Bitvector

QF BVFP

2.7 SAT-Solver

2.8 CNF

3 Approach

The approach starts with the problem definition and continues with what you have done. Try to give an intuition first and describe everything with words and then be more formal like 'Let g be ...'.

3.1 Problem Definition

Start with a very short motivation why this is important. Then, as stated above, describe the problem with words before getting formal.

3.2 First Part of the Approach

3.3 N-th Part of the Approach

4 Results

5 Conclusion

Literaturverzeichnis

[1] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, V. Lefevre, G. Melquiond, N. Revol, and S. Torres in *Handbook of Floating-Point Arithmetic*, vol. 2, pp. 20–120, Birkhäuser Bosten, 2018.

[2] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefevre, G. Melquiond, N. Revol, D. Stehle, and S. Torres in *Handbook of Floating-Point Arithmetic*, vol. 1, pp. 20–120, Birkhäuser Bosten, 2010.

[3] D. Goldberg in *What every computer scientist should know about floating-point arithmetic.*, Computing Surveys, 1991.

[4] NASA, "Engineering drawing standards manual.," no. X-673-64-1F, p. 90.