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

March 13th, 2023

Author
Robin Trüby

Matriculation Number
4709886

Writing Period
14. 12. 2022 – 13. 03. 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.

Freiburg, den 13.03.2023

Place, Date

Signature

Abstract

Satisfiability Modulo Theories (SMT) is the problem of deciding satisfiability of a logical formula, expressed in a combination of first-order theories. One of these theories is floating-point numbers. In this thesis we are interested in the floating-point arithmetic implemented in hardware. That is why we use bit-vectors to represent floating-point numbers. The multiplication of integers is hard. We want to show that already the addition of floating-point numbers can be hard. To demonstrate this, we have developed an algorithm that generates benchmarks for SMT solvers, showing that floating-point addition can be hard. These benchmarks are tested on six different SMT solvers. To do this we first demonstrate that performing two simple additions of floating-point numbers with the same numbers is not hard. Then, by expanding the algorithm, we demonstrate that the commutativity of floating-point numbers is hard.

Contents

List of Figures

List of Tables

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 losing 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 correctly and to avoid accidents, we need to make sure that 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 interested in the floating-point arithmetic implemented in hardware. Especially in the addition of floating-point numbers using GRS, a technique to round intermediate results. The addition is performed on a word-level, not on bit-level. To represent the floating-point numbers we use fixed-sized bit-vectors for the sign, the exponent and the significand. With different sets of benchmarks, mainly regarding the commutativity of floating-point numbers, we are challenging different state-of-the-art bit-vector SMT solver, such as Z3, Bitwuzla and Boolector. Because integer multiplication checking is hard, we conjecture that this problem might be hard as well.
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 numbers and SMT solver. After that, we are going to explain my algorithm for adding floating-point numbers. In the end the results and a conclusion are presented.

2 Preliminaries

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 formats for floating-point numbers, are defined in the IEEE 754 standard. The formats defined in the IEEE 754 standard are also what is implemented in most CPU's. The two most used formats are binary32 (single-precision) and binary64 (double precision). These formats are characterized by the number of bits used to represent the number. A radix-β floating-point number x is described thusly [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 digit before the radix point. After the radix point there are at most p-1 digits in the IEEE 754 standard, whereby p is the *precision* of the number. Note, that only p-1 bits are stored (Tab. 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 floating-point numbers to have negative exponents without adding a sign bit to the number [2]

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

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

Format	number of bits	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

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

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, that are commonly used.

A binary16 floating-point number is structured as follows:

$$x = 0|10011|1001100111$$

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.

$$10011_2 = 19_{10} \tag{3}$$

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

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

$$\begin{aligned} .1001100111 \\ 1.1001100111 \\ 1.1001100111_2 = 1,6005859375_{10}. \end{aligned}$$

Hence, x is equal to

$$\begin{aligned} x = (-1)^0 \cdot 1,6005859375_{10} \cdot 2^4 \\ x = 16005,859375_{10}. \end{aligned}$$

2.1.3 Subnormal numbers

Subnormal numbers are floating-point numbers, whose significand is close to zero. They have been included in the IEEE 754-1985 standard [2]. When adding subnormal numbers, a 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. Additionally the exponent is calculated differently.

$$e_{denormal} = e_{stored} - bias + 1 = e_{min}. \tag{5}$$

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

2.1.4 Special floating-point numbers

In the IEEE 754 standard, there are different representations for special numbers, including not-a-number (NaN), infinity, and zero. Zero and infinity are signed, means there is both a positive and a negative representation. The standard also includes two types of NaNs, signaling NaNs (sNaNs) and quiet NaNs (qNaNs) [3]. Signaling NaNs appear when there are invalid operations or uninitialized variables. They raise an 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.

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

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

2.2 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

$$\circ(a + b) = \circ(b + a) \tag{6}$$

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

$$x = 0|1011|100$$
$$y = 0|0111|011$$

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

$$1011_2 - 0111_2 = 0100_2 = 4_{10}$$

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.

$$1.100000_2 + 0.000101_2 = 1.100101_2$$

Note that, 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 properly.

$$1.100101_2 = 1.101_2$$

Putting everything together.

$$x + y = 0|1011|101$$

2.3 Rounding of 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 [3]:

- 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 algorithm of this thesis.

In order to round intermediate results, three additional bits are required for the significand. The three bits are called guard, round and sticky-bit (GRS). Further explanation are done on the following example of two binary 16 numbers.

$$x = 0|10011|1010101001$$
$$y = 0|10111|1100000000$$

The two numbers are now added. After comparing the exponents, the significand must be shifted to add both significands. The significand of x is shifted by 4 bits to the right.

$$0.00011010101001$$

The final significand oft he result can only be 10 bits long, but right now the significand is 14 bit long. To round the result, we keep three additional bits, and the last bit is truncated. The last bit oft he rounding bits, the sticky bit is an OR for all the bits, that are truncated. Because we cut off a one the sticky-bit is now a one too. After truncating the significand the two significands can be added.

$$\begin{aligned} &1.1100000000\ 000 \\ +&0.0001101010\ 101 \\ &---------- \\ &1.1101101010\ 101 \end{aligned}$$

To decide whether we round up or down a closer look at the GRS-bits is required. The guard-bit, the first one oft he GRS-bits, is a one. Additional, we look at the next two bits, the round an the sticky bit. If one of these bits is a one we round up, if not

significand with GRS	rounded	result
1.1101101010 101	up	1.1101101011
1.1101101010 010	down	1.1101101010
1.1101101010 100	down	1.1101101010
1.1101101011 100	up	1.1101101100

Table 3: Examples for rounding with GRS-bits.

the tiebreaker rule of round to even is applied. In this case the sticky-bit is a one so we round up. After rounding the significand of x will look like the following:

$$1.1101101011$$

In Table 3 are different examples of significands that need to be rounded, so that all possibilities are illustrated.

2.4 SMT solver

Satisfiability Modulo Theories (SMT) problem is a decision problem for logical formulas with respect to combinations of background theories [5]. 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, e. g. integer numbers, arrays, strings, bit vectors and floating-point numbers[6]. To go more into detail, we are working with the example of the Z3 SMT solver. Z3 is a cross-platform SMT solver by Microsoft. The main Theories supported are equality and uninterpreted functions (EUF), linear arithmetic, arrays, bit-vectors, algebraic data-types and sequences and strings [5]. Z3 supports a variety of languages as input.Next to its own native input language, Z3 accepts the SMT-LIB v2.6 script language. Furthermore, it is also accessible via an API, for example C++, C, Python, Java ,.Net and Ocaml. Z3 has a module called simplifier. The simplifier simplifies the input using different tactics, to optimize the performance, for example different preprocessing steps or *cube and conquer* [7]. The core technology of Z3 is a CDCL(T) algorithm. It combines different theory solver with a SAT solver, to solve the input formula.
To make it possible for the SAT-solver to solve the input formula, it must be translated into conjunctive-normal form (CNF). Therefor SMT solvers use different techniques, in the case of bit-vectors these techniques are *bit-blasting* (Sec. 2.4.1) or *term-rewriting*. CNF is a special type of propositional logic formula. When a SAT solver receives a CNF like this

$$(a \vee \neg b) \wedge (c \vee \neg d).$$

It checks if it satisfiable by looking for a scenario, where every clause is satisfiable. In this case a possible solution a SAT solver can return is $a = true$, $b = true$, $c = true$ and $d = true$. That means the solver returns *sat*. If there is no assignment of the literals, to make the formula *true*, the solver returns *unsat*.

2.4.1 Bit blasting

The most used theory in this thesis is fixed-sized bit-vectors. To be able to solve it, a SMT solver needs to transfer the bit-vectors to propositional logic. The most commonly used decision procedure is called *eager bit-blasting* (sometimes called *flattening*)[8]. The algorithm of *bit-blasting* is able to compute an equisatisfiable propositional formula $\mathcal{B}$ for a given bit-vector arithmetic formula φ, which is then passed to a SAT solver [8]. In the beginning, the algorithm is adding a Boolean variable for each bit of each sub-expression (term). That means having an 8-bit sized bit-vector, results in a vector with 8 Boolean variables. For every term and atom in φ the algorithm returns a constraint that as is added as a conjunct to $\mathcal{B}$ [8]. An easy example is the bit-blasting of $a + b$. Consider a and b are 1-bit bit vectors, all we need is a simple full adder (Fig. 1).

The outputs o and s are calculated as follows

$$\begin{aligned} s &\equiv (a + b + i) mod 2 \equiv a \oplus b \oplus i, \\ 0 &\equiv (a + b + i) div 2 \equiv a \cdot b + a \cdot i + b \cdot i. \end{aligned}$$

This is then transformed into CNF.

$$\begin{aligned} &(a \vee b \vee \neg o) \wedge (a \vee \neg b \vee i \vee \neg o) \wedge (a \vee \neg b \vee \neg i \vee o) \wedge \\ &(\neg a \vee b \vee i \vee \neg o) \wedge (\neg a \vee b \vee \neg i \vee \neg o) \wedge (\neg a \vee \neg b \vee o) \end{aligned}$$

This is easy to solve for a SMT solver. It is still manageable to solve when you add longer bit-vectors. Addition scales linear with the number of bits [8]. Other arithmetic

Figure 1: Picture of a 1-bit full adder with the inputs a, b and i and the outputs s and o.

operations, like multiplications, are not that easy to solve. A simple Multiplication of two 32-bit bit-vectors has more than 11 000 different Boolean variables, that an SMT solver needs to check. That is not solvable in a reasonable time frame [8].

2.4.2 SMT-Lib2

SMT-Lib2 was created to be a cross solver language, for example for benchmarks [6]. The benchmarks in this thesis are also written in SMT-Lib2. The first proposal to initiate the language SMT-Lib was made in 2003 [9]. Today the newest version, the one used in this thesis, is version 2.6 [6]. Its syntax is similar to the syntax of the LISP programming language [10]. Below (List. 2.1) is an example of an SMT-Lib2 algorithm in which two 32 bit bit-vectors are added.

Listing 2.1: SMT-Lib2 implementation for a simple 32-bit addition

```
1  (set-option :produce-models true)
2  (set-logic QF_BV)
3
4  (declare-const x (_ BitVec 32))
5  (declare-const y (_ BitVec 32))
6
7  (define-fun z () (_ BitVec 32) (bvadd x y))
8
9  (assert (= z #x11111111))
10 (check-sat)
11 (get-model)
```

At the beginning of the input file (List. 2.1), you can set various options. Here, the option to produce models was set true.This command makes it possible for the solver to return a model, if the formula is satisfiable. It is possible to declare one or more logics, that are used in the algorithm. QF_BV, the logic declared in line 2, means *quantifier free bit-vectors*. After that you can start declaring constants, functions and setting assertions.
In line 4 and 5 the fixed-sized bit-vectors x and y with the length of 32-bits are declared. In line 7 they are added together to the 32-bit long bit-vector z.The interesting part of the algorithm is line 9. In this line we tell the SMT solver that the solution of the addition of x and y must be $\#x11111111$ (hexadecimal value). After using the command *(check-sat)* the solver is checking for satisfiability. If the algorithm is *sat*, the solver will return *sat* and a model of a possible solution because of the command *(get-model)*. The output will look like:

Listing 2.2: Output from Z3 for List. 2.1

```
1      sat
2  (model
```

```
3     (define-fun y () (_ BitVec 32)
4       #x00000000)
5     (define-fun x () (_ BitVec 32)
6       #x11111111)
7   )
```

The SMT solver, in this case Z3, was able to find a solution and returned *sat* (line 1 of List. 2.2). To achieve that z is $\#x11111111$, Z3 suggests that y must be $\#x00000000$ (line 4) and x $\#x11111111$ (line 6).
The output can vary between different SMT solvers. Z3 returns bit-vector values as a hexadecimal number, whereas others return binary or decimal numbers.

3 Generation of the benchmarks

When considering floating-point arithmetic, most arithmetic properties from the arithmetics on real numbers are lost, like, for example associativity and distributivity. Only when the arithmetic operations are correctly rounded, floating-point addition and multiplication remain commutative [1]. The algorithm proposed in this thesis is creating different benchmarks for floating-point addition. Especially regarding commutativity of floating-point numbers. The main goal of the benchmarks is to show, that floating-point addition can be hard. The algorithm is written in C++. It takes the size used for the exponent and for the stored significand as input. The algorithm then returns a benchmark written in SMT-Lib2, which is a language explicitly developed for benchmarks on SMT solvers [6].
First, we implemented a simple floating-point adder, by only using the *quantifier free bit-vector* (QF_BV) logic. Then we extend the algorithm for a second addition so we can calculate $a + b = a + b$. The last step is to change the algorithm, so we calculate $a + b = b + a$. All of this benchmarks are tested on six different SMT solvers. The last one will also be directly tested on Kissat, a CDCL SAT-solver [11], by converting the SMT-Lib2 benchmarks to CNF files.
The first four SMT solvers, we use to test, are the all-purpose SMT solvers named Z3 by Microsoft Research [12], CVC4 and cvc5 by the University of Stanford and the University of Iowa [13] and Yices 2 by SRI International [14]. These solvers are made to function with most of the theories and logics, which the SMT-Lib2 language supports. Second, we are testing two solvers specialized on bit-vectors. First, Boolector an efficient SMT solver for Bit-vectors and arrays [15], and Bitwuzla a SMT solver for bit-vectors, floating-points, arrays and uninterpreted functions [16].

3.1 Main Algorithm

This section is about the main idea behind the algorithm. An abstract overview over the algorithm is illustrated in Figure 2. First, the algorithm written in C++ receives an input from the user. The input is the number of bits the user wants to use for his floating-point numbers. The input is given as two integer numbers, one for the number of bits stored for the exponent and the second is the number of bits stored for the significand. These two numbers plus the one bit for the sign-bit will determine the size of the floating-point number, for example the input 8 and 23 is resulting

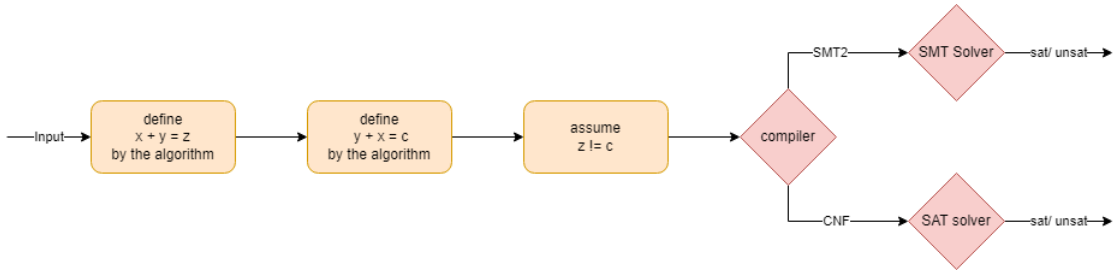

Figure 2: A flow graph of the basic parts of the final algorithm.

in the floating-point type *binary32*[1]. Next the algorithm builds a benchmark that contains two floating-point additions using GRS, for the floating-point type the user has declared in the input. In the end of the algorithm the assertion that $x+y \neq y+x$ is made so the solvers hopefully get challenged. As output the algorithm generates an .smt2 and a .CNF file. These files can be tested with state of the art SMT- and SAT-solvers.

3.2 Simple floating-point adder

To create a benchmark that focuses on the behavior of the commutativity of floating-point numbers using only quantifier free bit-vectors (QF_BV) logic the first step is to implement a simple floating-point adder. By using only QF_BV logic, we restrict ourselves to fixed sized bit-vectors only and cannot use any implemented floating-point data types, provided by an SMT solver. The adder is coded in SMT-Lib2 so we can use the same code on different SMT solvers [6].

3.2.1 Implementation

To implement the algorithm, we have used C++. The algorithm first asks the user, which floating-point type they would like to add. The four possible floating-point types are *binary16*, *binary32*, *binary64* and *binary128*. Allowing only fixed floating-point types, made it easier to debug (Section 3.2.2). After declaring the type, the C++ algorithm codes an SMT-Lib2 file. In the next paragraphs the structure of the SMT-Lib2 code, generated by the C++ algorithm is described.
First of all, the header of the smt2 file needs to be declared (List. 3.1). The first command gives basic information about the file and what it is purpose is. The second command forces the solver to produce a model of possible values of the declared variables. The solver only provides a model if the algorithm is satisfiable, otherwise an error occurs. The third command sets the logic. By setting a logic, the SMT solver directly knows which theories are used and can use the optimal theory solver to solve it.

Listing 3.1: Header of the SMT-Lib2 file

```
1 (set-info :source |Robin Trueby Bachelorthesis|)
2 (set-option :produce-models true)
3 (set-logic QF_BV)
```

After setting the header, the main algorithm can start. A brief overview of the algorithm as a pseudo-code is illustrated in List. 3.2. In the SMT-Lib2 algorithm, the two random numbers generated by C++ are asserted to x and y. The numbers x and y are then separated into three parts, the sign, the exponent and the significand. All three are stored as a bit-vector. The SMT-Lib2 algorithm then checks if one of the numbers is a special number, like for example +/- infinity (inf) or not-a-number (NaN). To do so, the algorithm looks at the exponent of the two numbers and compares it to predefined constant values. If the exponent of *number1* consists of only ones, the algorithm knows that number1 is either *inf* or *NaN*. A variable is set to true if one of the numbers is *inf* or *NaN*, so in the end the algorithm can decide the right output of the addition. To add the number properly, the hidden-bit needs to be added to the trailing significand. The algorithm checks if the number is a normal or subnormal number by looking at the exponent. Depending on the result it either adds a 1 or a 0 in front of the stored significand.

Also, an important step is the declaration of the bigger number, and how many bits the smaller number needs to shift his significand, so the numbers can get added. To declare the bigger number and the difference between the two exponents, the exponents of x and y are compared and the smaller one gets subtracted from the bigger. Then the significand of the smaller number gets shifted. After the shift the length oft the significand is three bigger as before, because the GRS-bits are added. Now that the significands have been adjusted, the algorithm decides if the numbers are added or subtracted (List. 3.2 line 9), to be added. Because floating-point numbers are signed numbers, you need to subtract them if they have different sign-bits, so you can add them. The algorithm checks if the sign-bits are the same, in order to add the two significands together, or if they are unequal, so that it can subtract the smaller significand from the bigger one. To add, the bit-vectors for the significands are added with the command *bvadd*. The algorithm does not add the bits one by one, instead it uses the already implemented function. This is why, it is a word-level and not a bit-level benchmark.
After the addition the algorithm checks, whether an overflow or an underflow has occurred. Looking for an overflow can be done by checking the first bits of the two numbers and of the result of the addition. If an overflow occurs the significand is shifted to the right. A one is added in front of the significand and the exponent is incremented by one. Checking for an underflow is simple as well. An underflow occurs when the hidden-bit of the result is zero, even though it should be one (always the

case except when adding two subnormal numbers). However, repairing an underflow is quite expensive. The algorithm needs to shift the significand of the result to the left for as long as the first bit is not a one (List. 3.2 line 17). The problem with that is that there are no loops in the SMT-Lib2 language, so it needs to be coded for every bit of the significand. For every shift to the left, the exponent is decreased by one. Next, the algorithm rounds the solution using GRS and checks if one of the numbers was *inf* or *NaN* at the beginning. If one of them was, it returns *inf* or *NaN*, else it returns the sum of both numbers.

Listing 3.2: Pseudo code for the algorithm of adding two floating-point numbers

```
1  GET number1 and number2
2  COMPUTE divide the number into sign, exponent, significa
3  IF one of the numbers is a special value THEN
4          set the value of the special case true
5  ENDIF
6  SET the bigger number to number1
7  SET the smaller number to nummber2
8  COMPUTE shift and round the smaller significand with GRS
9  IF the sign of number1 and number2 equal THEN
10         add the two numbers
11 ELSE
12         subtract number2 from number 1
13 ENDIF
14 IF an overflow occurred THEN
15         increment the exponent shift the significand
16 ENDIF
17 IF an underflow occurred THEN
18         WHILE the hidden bit is zero and
19             the exponent is not zero THEN
20                 decrement the exponent shift the
                       significand
21         ENDWHILE
22 ENDIF
23 COMPUTE round the solution of the calculation
24     of number1 and number2 with GRS
```

3.2.2 Debugging

To verify that the addition of the two floating-point numbers is correct, there are two possible ways. The first is to do it by hand. It is possible to take some edge, or special cases, like, for example both numbers are zero, one is a Inf or NaN, an overflow or an underflow, and see if the result is the same as when calculated by

hand. Testing these cases will eliminate most mistakes in the algorithm. This is a good strategy for the edge-cases, because most of the results are pre-defined and do not need to be calculated. But it is not suitable for checking normal numbers. An addition of binary32 floating-point numbers, is already too big to check a few possible examples in a reasonable time. That is why we need a different strategy. The second possibility is to compare the solution of the adder to the solution the computer gets, when calculating the same floating-point numbers. This is one reason why we use C++. C++ has implemented floating-point types, that are equal to binary32 and binary64 [17], defined in the IEEE-754 standard. An overview of the algorithm is illustrated in figure 3. First, we create two random floating-point numbers using the *rand()* function. Then we assign these two values to x and y in the SMT-Lib2 code.

Listing 3.3: Assignment of the values for x and y in the C++ algorithm

```
1  string bin_num1;
2  bin_num1 = num_gen_str(bin_num1, length - 6);
3  string bin_num2;
4  bin_num2 = num_gen_str(bin_num2, length - 6);
5  write << "(assert (= x #b" << bin_num1 << "))" << endl;
6  write << "(assert (= y #b" << bin_num2 << "))" << endl;
```

The random numbers are created as a string, because that made it easier to embed them in the SMT-Lib2 code. The type of the numbers needed to be changed to float, so C++ can calculate the solution. The type conversion is done by the self-implemented function *float print_num* $(string, int)$. In the end we add the two numbers and compare it with the solution provided by the SMT solver, and check for satisfiability.

Listing 3.4: The comparison of the solution calculated by the algorithm and C++

```
1  float num3 = num1 + num2;
2  string num3_str;
3  num3_str = encode(num3);
4  text << num3_str << endl;
5  write << "(assert (= z #b" << num3_str<< "))" << endl;
6  write << "(check-sat)" << endl;
7  write << "(get-model)" << endl;
8  write << "(exit)" << endl;
```

If, in this example, the SMT solver returns *sat*, the algorithm is good . If it returns *unsat*, then the algorithm is wrong. To debug, we removed the line where we assigned the value to *z*, and ran the code again to get a model of the solution. With the model we got from the SMT solver, we can check all the implemented steps and fix the algorithm. When the main function is executed, it runs the algorithm 1000 times and checks the created SMT-Lib2 file on an SMT solver directly. The result and

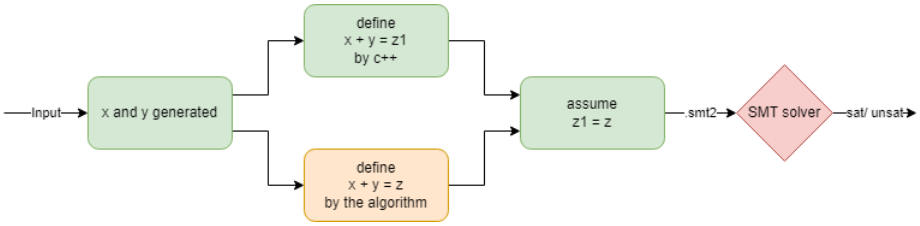

Figure 3: A flow graph of the simple adder and it´s verification.

the created numbers are stored separately in the file *zusammenfassung.txt*. After executing the main function, the *zusammenfassung.txt* file can be checked to see if an example returned *unsat*. The main algorithm was executed a few times with completely random numbers, to debug the algorithm. Subnormal floating-point numbers are handled different. To check the algorithm is valid for subnormal numbers too, the generating of the random was adjusted, so it only returned random subnormal numbers. The algorithm got executed a few more times, to debug the algorithm.

3.3 Double floating-point adder the same operation

After successfully creating a floating-point adder, the next step is to modify the algorithm, so it can calculate two additions. C++ is still used to get the SMT-Lib2 benchmarks. Unlike the simple adder, the algorithm is not asking for a floating-point type, it is asking for the number of bits used for the exponent and significand. For example, the input 8 and 23 results in *binary32* [1]. The sign is not an extra input, because it is always only one bit. The significand input is without the hidden-bit. The opportunity to create floating-point numbers by the size of exponent and significand, makes it easier to check if the solver has more struggles with bigger exponents, or bigger significands, compared to fixed types. The size of the exponent of the floating-point types, defined in the IEEE-754 standard, vary only in the range from 5 to 15, even though the size of the number gets eight-times bigger [1] (binary16 to binary128).

3.3.1 Implementation

To get an adder that calculates two additions, the simple adder is just duplicated. In the simple adder, the bit-vector x and y are added to z. In the new algorithm the addition of a and b to c is added. The structure of the algorithm is illustrated in figure 4. The bit-vectors x, y, a and b do not get an explicit value, compared to the simple adder. That also means the results z and c are not assigned with a value. Instead of fixed values assertions are used to create constraints between the bit-vectors. In the first part this assertion is used:

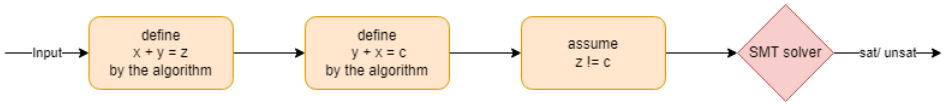

Figure 4: A flow graph of the double floating-point adder.

Listing 3.5: Variable assertion in the algorithm

```
1 (assert (= x a))
2 (assert (= y b))
3 (assert (distinct z c))
```

The assertion mean that x and a have the same value and y and b too. The third assertions means z is not equal as c. This results in the equation

$$\begin{aligned} a &= x \\ b &= y \\ a + b &\neq x + y. \end{aligned}$$

Obviously, this can not be true, as long as the implementation of the adder is correct. The benefit of intentionally setting it to false, is that in the worst case the SMT solver checks every possibility, until it returns *unsat*. If we would assert that z is equal to c, the SMT solver will mostly return the model of x, y, a, b being 0 and will return *sat*, which is indeed correct, but not *hard* for the SMT solver.
After the algorithm finishes, it returns a output file. The output file is the SMT-Lib2 algorithm, that can be run on different SMT solver. The name of the file for the input of 8 23, is code8_23.smt2.

3.3.2 Debugging

When setting $z \neq c$, the SMT solver should return *unsat*. Assigning explicit values to the fixed size bit-vectors x, y, a, and b, the solver returned *unsat*. However leaving the bit-vector unassigned, made the solver return *sat*. In the first case where $x = a, y = b$ got assigned, the mistake was found easily. At some point, some variables got assigned when they should be true, but not when they should be false.

Listing 3.6: The algorithm before debugging

```
1     (assert (ite (= m13 #b0) (= underflow1 #b1) (= none #b1)
          ))
```

In this case, when $m13 = 1$ the SMT solver has no assignment for *underflow1*. The goal for every SMT solver is to show that the algorithm is *sat*. In this case, the solver was able to find a way that the result for c and z differ. In the first addition of $x + y$ the SMT solver assigned *underflow1* to true. In the second addition, the addition of $a + b$ the SMT solver assigned the variable *underflow1_2* to false. That means in the first addition, the algorithm thinks an underflow has occurred, even though there is no underflow. That is why, the result of the addition differ. Changing the code to the following version solved it and the SMT solver returned *unsat*.

Listing 3.7: The algorithm after debugging

```
1       (assert (ite (= m13 #b0) (= underflow1 #b1) (=
            underflow1 #b0)))
```

3.4 Double floating-point addition commutative

3.4.1 Implementation

The algorithm, again written in C++, takes two numbers as input. One for the exponent and the other for the stored significand. Like the previous algorithm, the algorithm consists of two simple adder. The difference to the previous algorithm is the assertion in the end. The assertion got changed to:

Listing 3.8: Variable assertion in the algorithm

```
1 (assert (= x b))
2 (assert (= y a))
3 (assert (distinct z c))
```

The assertion means that x and b have the same value, as well as y and a. The third assertion means that z is not equal to c. This results in the equation:

$$\begin{aligned} a &= y \\ b &= x \\ a + b &\neq x + y \end{aligned}$$

Like the other example, this is also implemented, so the SMT solver returns *unsat* when it is implemented correctly. After the algorithm finishes, it returns an output file. The output file is the SMT-Lib2 algorithm that can be used, to run the algorithm on different SMT solver. The name of the file for the input of 8 23, is code8_23.smt2. Additionally to the SMT-Lib2 file, the algorithm also returns a code8_23.CNF file,

to check the algorithm directly on a SAT solver. To run the algorithm it is important that boolector and aiger are installed. The finished algorithm is illustrated in figure 2.

3.4.2 Debugging

Changing the assignment to $x = b, y = a$, the same mistake as before occurs. The SMT solver returned *sat* even though it should return *unsat*. This time the mistake was that the algorithm had problems deciding which number is bigger. We assumed it would be enough to check which number has the bigger exponent. To assign the sign-bit for z, we used the command *bvuge* (bit-vector unsigned greater or equal). This is valid as long the numbers x and y have different exponents or the same sign-bit.

Listing 3.9: The algorithm before debugging

```
1 (assert (ite (bvuge x_expo y_expo) (= z_sign x_sign) (=
    z_sign y_sign)))
```

The problem occurs when the numbers have the same exponent but different sign-bits. In the first addition, the algorithm would assume that x is bigger. In the second addition, the algorithm will think that a ($a = y$) is bigger. When x and y have different sign-bits, that means z and c have different sign-bits too. This results in the satisfiability of the algorithm. To solve the problem, the algorithm checks, additionally to the exponent, if the significand is bigger.

Listing 3.10: The algorithm after debugging

```
1 (assert (ite(bvuge x_expo y_expo) (ite(= x_expo y_expo)
    (ite(bvuge x_man y_man) (= z_sign x_sign)
2 (= z_sign y_sign))(= z_sign x_sign))(= z_sign y_sign)))
```

After this change the SMT solver correctly returned *unsat*.

3.5 Using the QF_BVFP logic

So far, we have only used QF_BV logic to implement floating-point addition. In this example we use the QF_BVFP logic, it has the advantage of an already implemented floating-point type. In the first part we compare the result of the implemented simple adder, to the result of the addition of floating-point number in the QF_BVFP logic. In the second part, we created benchmarks using only the floating-point implementation of the SMT solver.

3.5.1 QF_BV and QF_BVFP logic

In this section we are using the implemented floating-point type from the SMT solver. In the first addition we are using the self implemented algorithm to add two floating-point numbers. In the second addition we are using the floating-point type implemented in the QF_BVFP logic from the SMT solver. In the end we compare the results from the addition of my algorithm and the result of the implemented floating-point type. The numbers of the second addition are switched, so we are testing the commutativity of these calculation. The benchmark is implemented the way, it returns *unsat*, when the solution of our implementation and the SMT solver implementation is the same.

$$x + y \neq y + x$$

The approach to compare our implementation with the SMT solvers floating-point implementation, is inspired by Brain et al [18]. After testing the results against the floating-point data types from C++, we wanted to verify that the self implemented floating-point addition will return the same solutions as the SMT solver.
In the first part of the algorithm, we used the simple floating-point adder from Section 3.2. For the second addition we used the floating-point data type from the QF_BVFP logic. In the following the additional part of the floating-point logic is shown.

Listing 3.11: The floating-point functions used in this algorithm

```
1  (declare-const a (_ FloatingPoint 8 24))
2  (declare-const b (_ FloatingPoint 8 24))
3  (declare-const c (_ FloatingPoint 8 24))
4  (declare-const xs (_ FloatingPoint 8 24))
5  (declare-const ys (_ FloatingPoint 8 24))
6
7  (assert(= true (fp.isNormal a)))
8  (assert(= true (fp.isPositive a)))
9  (assert(= true (fp.isNormal b)))
10 (assert(= true (fp.isPositive b)))
11
12 (assert (= c (fp.add RNE b a)))
13
14 (assert (= xs ((_ to_fp 8 24) x)))
15 (assert (= ys ((_ to_fp 8 24) y)))
16
17 (assert (= ys a))
18 (assert (= xs b))
19 (assert (distinct ((_ to_fp 8 24) z) c))
```

In the first five lines of the provided algorithm part, we are declaring the floating-point

variables. The 8 and 24 in the *(declare-const a (_ FloatingPoint 8 24))* command, is declaring a binary32 number. Eight bits for the exponent and 24 bits significand, therefore 23 explicitly stored. The last bit missing is the sign-bit. In the line seven to then we excluded some edge-cases, like, subnormal-numbers. In this example we are only comparing positive numbers. When converting the bit-vectors to floating-point the sign gets lost, so we decided to only look at positive numbers. Negative numbers have been tested as well, but we provide only benchmarks for the positive numbers. In line 14, 15 and 19 the algorithm is converting the bit-vectors to floating-point numbers. In line 12 the two floating-point numbers b and a are added with the *round to nearest even* (RNE) mode.

3.5.2 Only QF_BVFP logic

In this set of benchmarks, only the floating-point implementation of the SMT solver is used. In List. 3.12 is the benchmark. First, the four floating-point numbers x, y, a and b are declared. Then they are added with the RNE rounding mode. In the end the variables are asserted. The algorithm checks if the different SMT solvers have problems with commutativity of floating-point numbers, even in the self implemented data type.

Listing 3.12: The algorithm for two additions of floating-point numbers using the QF_BVFP logic

```
1  (set-option :produce-models true)
2  (set-logic QF_BVFP)
3
4  (declare-const x (_ FloatingPoint 8 23))
5  (declare-const y (_ FloatingPoint 8 23))
6  (declare-const a (_ FloatingPoint 8 23))
7  (declare-const b (_ FloatingPoint 8 23))
8
9  (define-fun z () (_ FloatingPoint 8 23) (fp.add RNE x y))
10 (define-fun c () (_ FloatingPoint 8 23) (fp.add RNE b a))
11
12 (assert (= x a))
13 (assert (= y b))
14 (assert (distinct z c))
15 (check-sat)
16 (get-model)
```

4 Evaluation

This chapter provides a detailed evaluation on different sets of benchmarks. The evaluation is separated in five sections. The first part is the introduction that describes the experimental setup (Sect. 4.1). The second section is about the first pair of crafted benchmarks with the assertion of $x + y \neq x + y$ (Sect. 4.2). Third, the evaluation is done on the next set of crafted benchmarks with the assertion of $x + y \neq y + x$ (Sect. 4.3). In the fourth part, is a set of benchmarks evaluating the impact of bigger exponents and significands (Sect. 4.4). The last part, evaluates the benchmarks including the QF_BVFP logic (Sect. 4.5).

4.1 Experimental Setup

All experiments are done on the Intel i7-10510U CPU with 16 GB of Memory. To run the different SMT solvers the Linux subsystem on windows (WSL2) was used. All benchmarks, if possible have been tested on six different SMT solvers and on one SAT solver. The six SMT solver and their version used are:

- Z3: Version 4.8.7 - 64 bit
- CVC4: Version 1.6
- cvc5: Version 1.0.3-dev.172.1729c50f2
- Yices 2: Version 2.6.4
- Boolector: Version 3.2.2
- Bitwuzla 1.0-prerelease

The SAT solver used is Kissat with the version 3.0.0. If not different mentioned, all benchmarks are run with the command ‘time timeout 1000 xy-solver file‘.

4.2 Double floating-point adder the same operation

In this section the benchmarks from section 3.3 are tested, and evaluated. This benchmarks have the assertion of $x = a,\ y = b$. The evaluation is done on all six SMT solvers. First the algorithm, with relational *ITE* statements, is tested and evaluated. In the second part we changed the relational *ITE* commands to a functional one and tested and evaluated it again.

4.2.1 The algorithm with relational ITE

With the algorithm described in section 3.3 a set of five different benchmarks is generated. The first four are testing the four floating-point data-types described in the IEEE 754-2008 standard, *binary16*, *binary32*, *binary64* and *binary128*. The last benchmark is testing a octuple-precision floating-point format (*binary 256*). The octuple-precision floating-point format uses 1-bit for the sign, 19-bits for the exponent and 237-bits for the significand precision (236 explicitly stored) [19]. All benchmarks are tested on Z3, cvc5, CVC4, Yices 2, Boolector and Bitwuzla.

The results of the testing can be seen in the Table 4 and a visualization is in Figure 5. The benchmark for two consecutive additions of the same numbers in *binary16*, could be solved by every SMT solver under seven seconds. Z3, Boolector and Bitwuzla seems to be the fastest option for the small format. All three solved it in under two seconds. CVC4 and cvc5 are the slowest, taking approximately 6 seconds to solve it. Regarding the 32-bit benchmarks CVC4 and cvc5 remaining the slowest taking almost a minute to solve it. For bigger data-types CVC4 could not finish before the timeout of 1000 s appealed. cvc5 is slightly faster than CVC4. cvc5 is more as twice as slow as the next slowest one, Yices 2. Even Z3 was pretty fast for the small data-type, it is getting slower for bigger data-types compared to Boolector and Bitwuzla. Z3 needed more time to solve the 64-bit addition, than Boolector and Bitwuzla needed for the addition of the 128-bit data-type. Boolector and Bitwuzla are the only solver that managed to solve the addition of *binary128* number in under 1000 seconds. Bitwuzla was the fastest solver for all solvable data-types except the first one. It was over 20 percent faster than Boolector for the *binary128* addition. None of the solver could solve the addition of the *binary256* floating-point numbers.

That the solvers Boolector and Bitwuzla are the fastest to solve benchmarks regarding the quantifier free bit-vector logic was expected, because both of the are specialized in solving bit-vector. But still no one did realize that the benchmark is just the same addition of floating-point numbers duplicated.

Solver	binary16[s]	binary32[s]	binary64[s]	binary128[s]	binary256[s]
Z3	1.56	18.64	279.48	1000.00	1000.00
cvc5	6.66	58.25	685.53	1000.00	1000.00
CVC4	5.56	55.80	1000.00	1000.00	1000.00
Yices 2	2.27	38.73	318.47	1000.00	1000.00
Boolector	1.35	6.25	27.41	265.29	1000.00
Bitwuzla	1.46	4.67	23.49	194.45	1000.00

Table 4: Time needed for the SMT solvers to solve the benchmarks of the algorithm for the same addition twice, with the relational *ITE*.

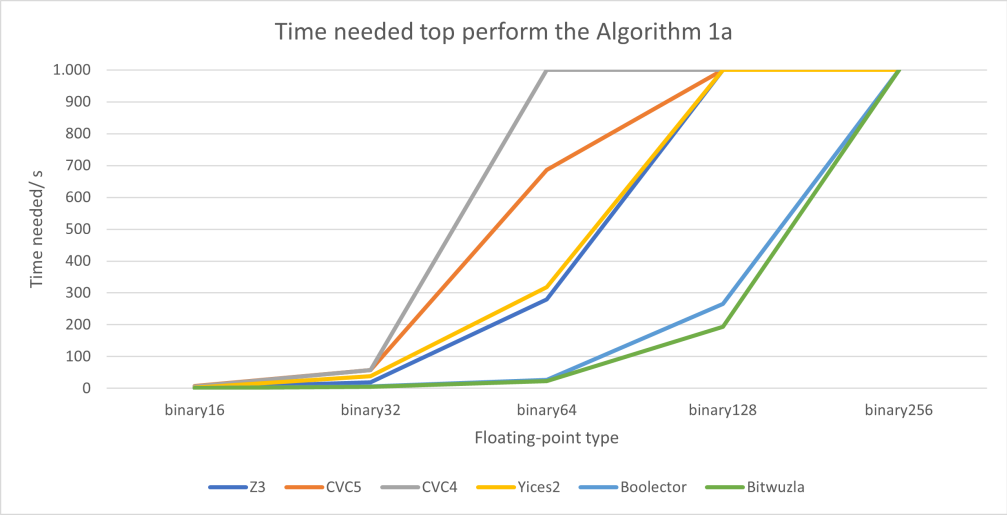

Figure 5: Time needed for the SMT solvers to solve the benchmarks of the algorithm for the same addition twice, with the relational *ITE*.

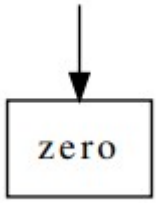

Figure 6: The DAG of the algorithm with the functional *ITE*.

4.2.2 The algorithm with functional ITE

Testing the algorithm for the addition of two floating-point numbers with the assertion $x = a, y = b$ on all six SMT solver returned interesting results. To visualize it on hardware level we used the *-db code.smt2|btorvis.sh* command on Boolector. The command returned, even for a 3 bit long floating-point number, a big complicated hardware implementation (fig. 7). An idea to make it easier for the solvers, was to change the algorithm, whenever a *ITE* was involved. This changed the *ITE* statements from an relational to an functional representation. So we changed the whole algorithm this way.

Listing 4.1: The algorithm before the change

```
1 (assert (ite(bvuge x_expo y_expo) (ite(= x_expo y_expo)
    (ite(bvuge x_man y_man) (= z_sign x_sign)
2     (= z_sign y_sign))(= z_sign x_sign))(= z_sign
        y_sign)))
```

Listing 4.2: The algorithm after the change

```
1 (assert (= z_sign ((ite(bvuge x_expo y_expo) (ite(=
    x_expo y_expo) (ite(bvuge x_man y_man) x_sign
2     y_sign)x_sign)y_sign)))
```

The same set of benchmarks, but this time with the functional representation of the *ITE*, are used on the six SMT solvers. The results of the benchmarks can be seen in the Table 5 and a visualization is in Figure 8. The results completely differ from the previous one. The change in the algorithm appear to made a big difference. cvc5, CVC4, Yices 2, Boolector and Bitwuzla now realize that the algorithm is basically just the repeated addition of the same two numbers. When adding the same numbers twice and compare the result of the addition, the result should be the same, so the solvers directly know that the algorithm is unsatisfiable, because we asserted that the results must differ. The solvers need for all data-types, even for the *binary256*, less than a second to solve it. The only solver that appears to have problems with

Solver	binary16[s]	binary32[s]	binary64[s]	binary128[s]	binary256[s]
Z3	5.82	733.58	1000.00	1000.00	1000.00
cvc5	0.05	0.08	0.14	0.33	1.03
CVC4	0.05	0.07	0.01	0.26	0.86
Yices 2	0.02	0.03	0.02	0.04	0.06
Boolector	0.03	0.02	0.03	0.03	0.03
Bitwuzla	0.02	0.03	0.03	0.03	0.04

Table 5: Time needed for the SMT solvers to solve the benchmarks of the algorithm for the same addition twice, with the functional *ITE*.

the change is Z3, it is five times as slow as before for the *binary16*, and almost 40-times slower for the *binary32* addition.
It is easier for majority of the solvers, because they now can extract the whole statement as one literal, instead of one literal for every $(= z\_sign\ x\_sign)$ assignment. It seems that Z3, the only SMT solver not able to detect this relation, is missing some simplifications.
After this change boolector returns with the same command as previous, a one gate implementation (Fig. 6). In the Figure it is visible that the SMT solver knows, independent on the input it always return *unsat*. This is the right conclusion considering the benchmark tests $x+y \neq x+y$. That realization by the solvers, results in a fast computing time for floating-point additions, even for bigger floating-point types.

4.3 Double-floating-point addition commutative

We have tested which SMT solver can solve two identical floating-point additions the fastest. Now we are testing which one can solve two floating-point additions with switched numbers in the second addition (the algorithm from section 3.4), the fastest. That means we are checking if the solver can actually identify the law of commutativity $x + y = y + x$ that we have implemented for floating-point numbers. Both version of the *ITE* implementation are tested. In the first section the relational *ITE* is evaluated. In the second section the functional *ITE* is evaluated. The algorithm is tested on the six SMT solvers Z3, CVC4, cvc5, Yices 2, Boolector and Bitwuzla. The code is running five times on every solver and the average is calculated. The most common floating-point types including *binary16, binary32, binary64* and *binary128* are tested. Additionally to that, the *binary256* type from the previous section is tested.
The algorithm is also tested on the SAT solver Kissat. Therefore the .smt2 file is converted with Boolector to a .cnf file.

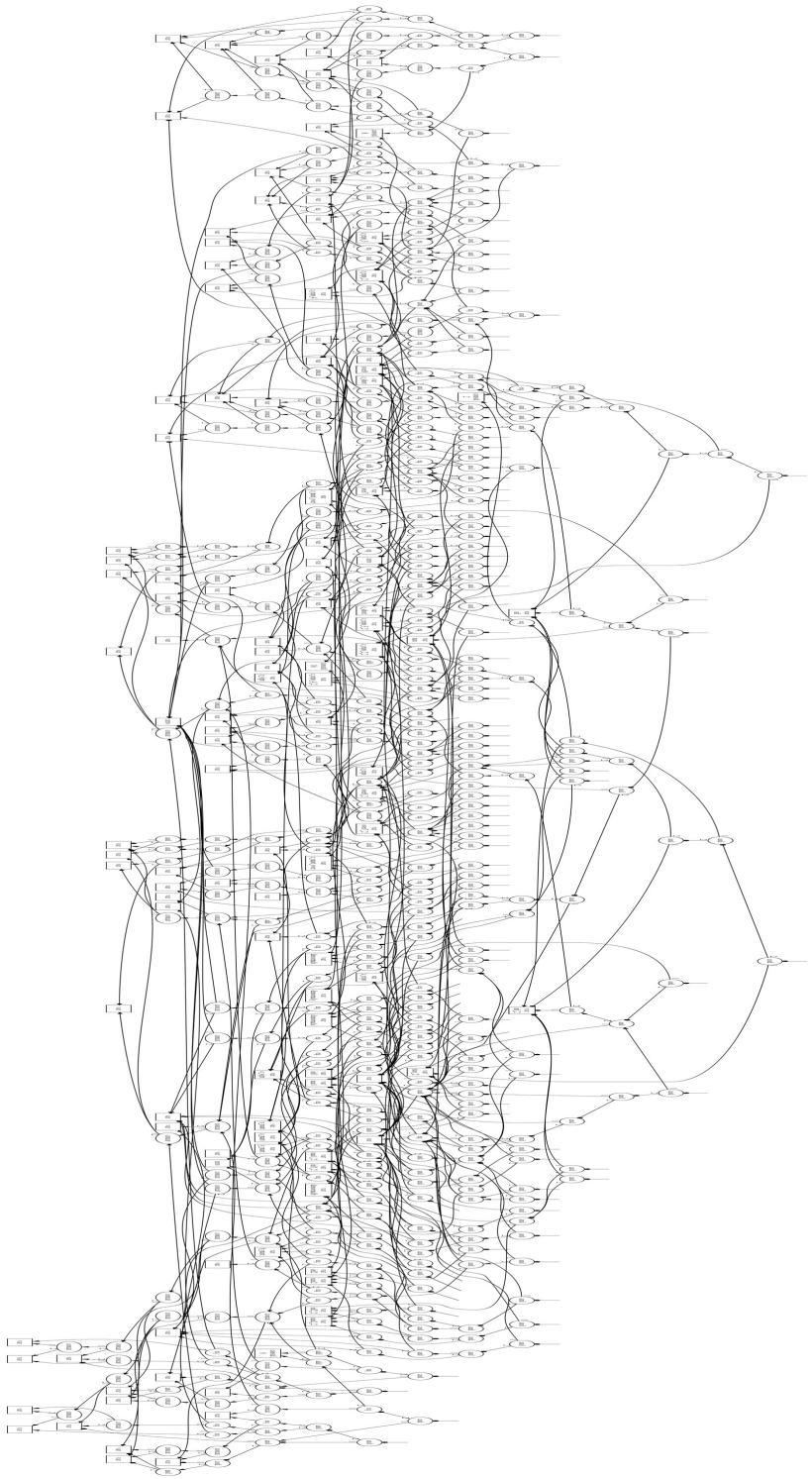

Figure 7: The visualization of two additions of floating-point numbers with the bit length of 3 bit, and the assignment of $x = a, y = b$.

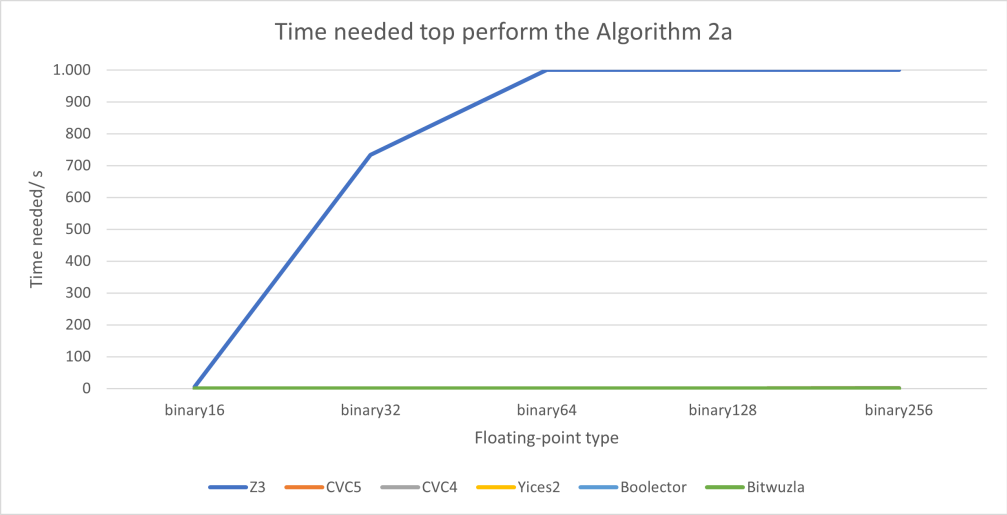

Figure 8: Time needed for the SMT solvers to solve the benchmarks of the algorithm for the same addition twice, with the functional *ITE*.

4.3.1 The algorithm with relational ITE

First, the algorithm with relational *ITE* statements from section 4.2.2 is evaluated. The only difference to the benchmarks from section 4.2.1 is the different assertion in the end. We assigned the variable so that $x = b, y = a$ with $x = a, y = b$. That means we are testing the commutativity of the floating-point numbers. The results of the testing can be seen in the Table 6 and a visualization is in Figure 9.
The results are similar to the results from section 4.2.1. Z3 and CVC4 exactly needed the same amount of time to solve the benchmarks. cvc5, Boolector and Bitwuzla needed about 30 percent longer compared to section 4.2.1. Yices 2 needed almost double the time as before. For the *binary16* addition, the SAT solver Kissat is faster than the SMT solvers. The bigger the data-types are, the slower Kissat gets, compared to Boolector and Bitwuzla. For the *binary128* addition Kissat was double as slow as Boolector and three times as slow as Bitwuzla.

To check the size of the created SMT benchmark, we are looking at the size of the BTOR formula. BTOR is a low-level bit-vector format, which is also a possible input format besides SMT-Lib2 [15] for Boolector. Adding *-db* to the command to execute the benchmarks, Boolector returns the BTOR formula. The different sizes are visible in Table 8.
The number of code lines does not change much, compared to the benchmarks from section 4.2.1. It is constantly added just two more lines. The size of the formula does change with the size of the floating-point data-type. Except the step from *binary32* to *binary64*, where it somehow gets lower, it always double the lines. The time the SMT solver needs, depends not only on the size of the BTOR formula, because boolector

Solver	binary16[s]	binary32[s]	binary64[s]	binary128[s]	binary256[s]
Z3	1.16	16.09	279.40	1000.00	1000.00
cvc5	6.79	69.56	869.44	1000.00	1000.00
CVC4	6.11	57.77	1000.00	1000.00	1000.00
Yices 2	4.45	37.63	616.94	1000.00	1000.00
Boolector	1.37	9.05	39.27	322.54	1000.00
Bitwuzla	1.40	5.85	31.36	246.86	1000.00
Kissat	1.08	11.08	67.06	723.07	1000.00

Table 6: Time needed in seconds for the SMT solvers to solve the benchmarks of the algorithm for the commutative floating-point addition, with the relational *ITE*.

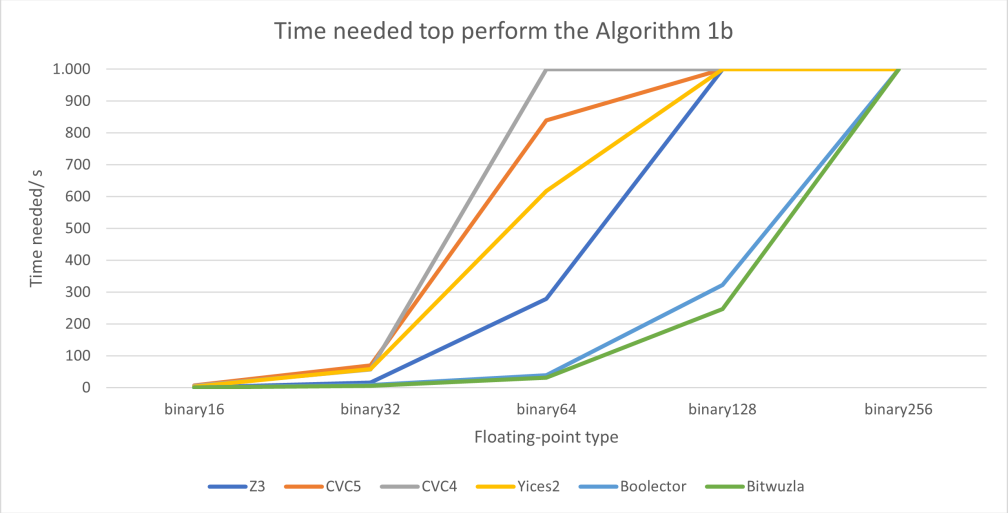

Figure 9: Time needed for the SMT solvers to solve the benchmarks of the algorithm for the commutative floating-point addition, with the relational *ITE*.

still needs more time for the *binary64* addition than for the *binary32* even the 32-bit numbers had 2000 lines more. Kissat is slower than Boolector even though Kissat has one step less. Kissat does not need to extract the CNF file out of the SMT-Lib file. One possible reason is that Boolector is using a different SAT solver with lingeling, that might be faster.

4.3.2 The algorithm with functional ITE

In this section the commutativity of floating-point numbers is tested. The benchmarks used are with the functional *ITE*. The benchmarks are evaluated on all six SMT solvers and on the SAT solver Kissat. The results of the testing can be seen in the Table 7 and a visualization is in Figure 10.

Solver	binary16[s]	binary32[s]	binary64[s]	binary128[s]	binary256[s]
Z3	10.37	1000.00	1000.00	1000.00	1000.00
cvc5	6.66	58.25	685.53	1000.00	1000.00
CVC4	6.33	56.04	1000.00	1000.00	1000.00
Yices 2	2.97	27.15	870.20	1000.00	1000.00
Boolector	0.74	5.70	65.10	409.72	1000.00
Bitwuzla	0.91	6.81	31.32	338.35	1000.00
Kissat	0.83	4.23	38.93	459.59	1000.00

Table 7: Time needed for the SMT solvers to solve the benchmarks of the algorithm for the commutative floating-point addition, with the functional *ITE*.

Once again the two SMT solvers Boolector and Bitwuzla are the fastest regarding the overall performance on this set of benchmarks. For the *binary64* addition Bitwuzla is twice as fast as Boolector. For the addition of 128-bit numbers Boolector is only 25 percent slower than Bitwuzla. Z3 is the slowest. It did not finish the benchmark for 32-bit numbers. cvc5 and CVC4 are approximately equal as fast except the *binary64* benchmark. This benchmark could be finished by cvc5 but not by CVC4. For the *binary64* benchmark cvc5 was even faster than Yices 2, although Yices 2 is mostly two times as fast as cvc5.
For the first two benchmarks, Kissat is faster than Boolector and Bitwuzla. For the 128-bit addition Kissat is slower as them.

For the algorithm with functional *ITE*, that only tested two additions with the same numbers, five out of six SMT solver were able to detect that they are testing the same addition twice. This time none of them could detect it. Even though, the only difference is that the algorithm got changed from $x + y \neq x + y$ to $x + y \neq y + x$. Overall the performance is even worse, then the benchmarks with relational *ITE*.
In Table 8 are the sizes of the BTOR formula of the benchmarks. The BTOR file of Boolector is already the simplified version of the benchmarks. Boolector is applying several simplification and rewriting rules on the SMT-Lib2 benchmarks. After simplifying Boolector returns the BTOR formula. That means the size of the BTOR formula, tells us, how much is later translated into CNF for the internal SAT solver.
The BTOR formulas for these benchmarks are getting bigger the bigger the floating-point data-types are. The number of lines vary from 410 (*binary16*) to 3095 (*binary256*). Compared to the benchmarks with relational *ITE*, the number of lines is at least three times smaller and the execution time is longer. That means, the amount of lines in the BTOR formula is not mandatory comparable to the run time. The change from $x + y \neq x + y$ to $x + y \neq y + x$ and the change of the *ITE* returned small hard benchmarks for the SMT solvers and for the SAT solver.

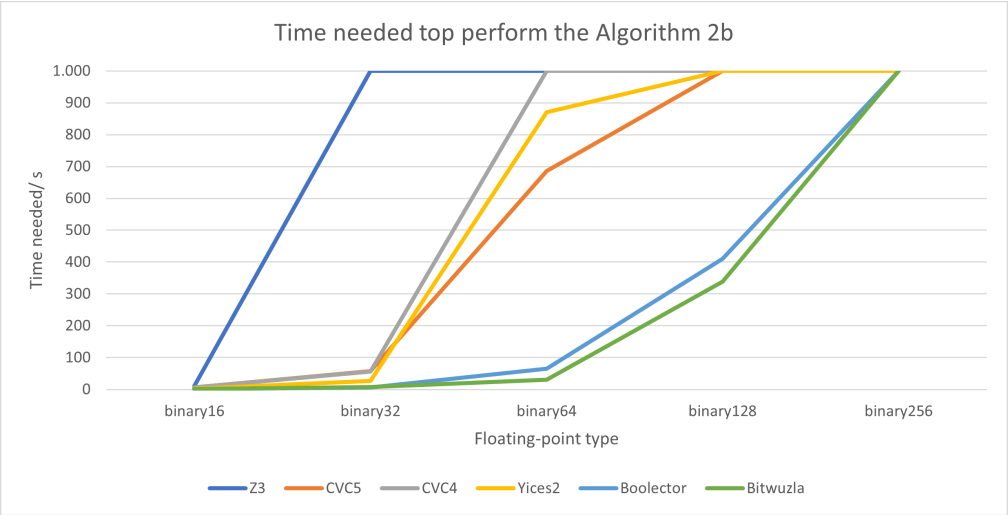

Figure 10: Time needed for the SMT solvers to solve the benchmarks of the algorithm for the commutative floating-point addition, with the functional *ITE*.

Algorithm from Section	binary16	binary32	binary64	binary128	binary256
4.2.1	1760	4620	2879	5399	10607
4.3.1	1762	4622	2881	5401	10609
4.2.2	2	2	2	2	2
4.3.2	410	592	887	1607	3095

Table 8: Number of lines in the BTOR formula returned by boolector.

4.4 Vary the exponent and the significand

So far we only regarded existing floating-point data types and evaluated them. The problem of the existing data-types, regarding the evaluation of the benchmarks, is that the exponent-bit only vary from 5 to 19 bits. Considering the significand-bit vary from 10 to 236, the range of the exponent-bit is small. We are again testing the created benchmarks on all six SMT solvers. We use the algorithm with functional *ITE* statements.

4.4.1 Exponent

This time the first set of benchmarks in Table 9, had a fixed number of significand-bits and the number of exponent-bits got varied. The number of significand-bits is 10 bit. The exponent-bit size ranged from 5 to 160 bits. The SMT solver are getting slower the bigger the exponent gets. The rise of time looks more linear, than exponential. That means the number of exponent-bits is not important, for the benchmarks to be hard.

4.4.2 Significand

The next set of benchmarks is about testing the behavior of the SMT solvers, when changing the number of significand-bits. This time the number of exponent-bits is static with 10 and the number of significand-bits vary from 5 to 160 bits.
The results are in Table 10. The SMT solvers getting slower the bigger the significand gets. The rise of the time looks exponential. Combined with the test on the exponent it is clear that the expensive part of the benchmarks comes from the number of significand-bits. One reason for that is that the bigger the significand is, the bigger the algorithm is. To check for underflow we needed to code a loop. Since there is no loop command in the SMT-Lib2, we needed to unfold the loop. To check for an

Solver	5[s]	10[s]	20[s]	40[s]	80[s]	160[s]
Z3	10.25	8.32	16.99	14.89	16.33	31.60
cvc5	20.44	26.44	37.15	50.12	1m12.10	2m40.29
CVC4	6.46	6.59	9.36	17.44	56.04	3m44.61
Yices 2	2.99	5.33	8.51	7.78	19.51	49.08
Boolector	0.77	1.00	1.90	2.99	6.71	13.56
Bitwuzla	0.89	1.19	1.76	2.83	6.47	11.38

Table 9: Time needed for the SMT solver to solve the benchmarks with fixed number of significand-bits and different size of exponent-bits.

Solver	5[s]	10[s]	20[s]	40[s]	80[s]	160[s]
Z3	0.68	8.32	268.97s	1000.00s	1000.01s	10000.00s
cvc5	4.75s	26.43s	207.25s	1000.00s	1000.00s	1000.00s
CVC4	2.28s	6.68s	36.51s	610.44s	1000.00s	1000.00s
Yices 2	0.42s	5.33s	27.34s	308.99s	1000.00s	1000.00s
Boolector	0.34s	0.98s	5.54s	23.18s	2m3.73s	937.44s
Bitwuzla	0.32s	1.24s	3.99s	18.80s	122.49s	669.28s

Table 10: Time needed for the SMT solvers to solve the benchmarks with fixed number of exponent-bits and different size of significand-bits.

underflow three lines of code and two new variables are added for every significand-bit. The significand-bits are responsible for the benchmarks to be hard.

4.5 Floating-point addition using QF_BVFP logic

In this section we make use of the *quantifier free bit-vector floating-point theory*. This theory provides an already available floating-point type. In the first part, we are evaluating the behavior, when compare our implementation of floating-point arithmetic against the provided data-types from the SMT solver. In the second part we are using the floating-point theory on both additions.

4.5.1 Using bit-vectors and QF_BVFP logic to add

In the first part, we have evaluated the behavior, of our implementation of floating-point arithmetic against the provided data-types from the SMT solver. It was possible to test it on three different solvers. Only Bitwuzla, Z3 and cvc5 support all commands in this benchmark. CVC4 does support the QF_BVFP logic in general, but does not support the function of transforming bit-vectors to floating-point types in the base kit. We have tested the addition of binary16, binary32, binary64, binary128, and binary256. cvc5 only supported the transforming of binary32 and binary64 bit-vectors to floating-point. The results are in Table 11.

Solver	binary16[s]	binary32[s]	binary64[s]	binary128[s]	binary256[s]
Z3	0.83	12.43	1000.00	1000.00	1000.00
cvc5	-	98.01	1000.00	-	-
Bitwuzla	0.66	2.25	18.81	132.48	1000.00

Table 11: Time needed for the three SMT solvers to solve the benchmark, that contains bit-vectors and floating-point numbers.

Bitwuzla was fastest of all the SMT solver possible to run these type of benchmark. cvc5 was only able to run two benchmarks, and was the slowest for both of them.

4.5.2 Using QF_BVFP logic only

In this part we have tested a set of benchmarks with the QF_BVFP logic. Boolector and Yices 2 do not support this logic, that is why we only tested it on four different SMT solvers. We have evaluated if the solver had any problems recognizing, that the same floating-point addition is done twice and then compared the results. Additionally, we have tested if there are runt-time differences, when the numbers in the second addition are switched. The results are in Table 12. The four SMT solver supporting the QF_BVFP logic had no problems with both of the problems. All solvers needed the same amount of time for both benchmarks.

Solver	a+b = a+b[s]	a+b = b+a[s]
Z3	3.68	3.68
cvc5	0.04	0.04
CVC4	0.03	0.03
Bitwuzla	0.02	0.02

Table 12: Comparison for two 32 Bit floating-point number additions, using the floating-point logic on four different SMT solver.

5 Conclusion

We introduced an algorithm, that creates different sets of benchmarks for the addition of floating-point numbers. We implemented the floating-point numbers as implemented in hardware, using fixed-sized bit-vectors for the exponent and the significand. GRS, a technique to round intermediate results, was implemented as well.
We verified the algorithm, by first comparing the results of the algorithm with C++. Additionally we compared the results of the algorithm with the SMT solver implemented floating-point numbers.
With the first set of benchmarks, we showcased that the SMT solvers could recognize the symmetry of the two floating-point additions, that are repeated. Then we changed the benchmarks to a commutative addition of the floating-point numbers. This change returned a set of small, hard benchmarks. All of the six different SMT solver and Kissat, the SAT solver, were not able to finish the addition of the *binary256* addition before the timeout interrupted the solvers.
The last benchmarks showed against our assumption, that the number of significand-bits is more crucial for the execution time, compared to the exponent-bit. Larger exponents resulted only in linear rise of the run-time. Whereby more bits for the significand resulted in a exponential rise of the run-time.
To sum up we have showed, that not only integer multiplication can be hard, also floating-point addition implemented in hardware is hard.

Bibliography

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

[3] 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.

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

[5] N. Bjorner, L. de Moura, L. Nachmanson, and C. Wintersteiger, "Programming Z3," in *Programming Z3, Microsoft Research*, 2010.

[6] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)." www.SMT-LIB.org, 2016.

[7] "Z3prover." https://github.com/Z3Prover/z3.

[8] D. Kroening and O. Strichman, eds., *Decision Procedures An Algorithmic Point of View*, vol. 2. Springer-Verlag Berlin Heidelberg, 2016.

[9] S. Ranise and C. Tinelli, "The smt-lib format: An initial proposal," in *Proceedings of the 1st Workshop on Pragmatics of Decision Procedures in Automated Reasoning (Miami Beach, USA)*, 2003. Available at www.smt-lib.org.

[10] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," tech. rep., Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.

[11] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions* (T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, eds.), vol. B-2020-1 of *Department of Computer Science Report Series B*, pp. 51–53, University of Helsinki, 2020.

[12] L. M. de Moura and N. Bjørner, "Z3: An efficient smt solver.," in *TACAS* (C. R. Ramakrishnan and J. Rehof, eds.), vol. 4963 of *Lecture Notes in Computer Science*, pp. 337–340, Springer, 2008.

[13] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength SMT solver," in *Proceedings of the* 28^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '22)* (D. Fisman and G. Rosu, eds.), vol. 13243 of *Lecture Notes in Computer Science*, pp. 415–442, Springer, Apr. 2022. Best SCP Tool Paper Award.

[14] B. Dutertre, "Yices 2.2," in *Computer-Aided Verification (CAV'2014)* (A. Biere and R. Bloem, eds.), vol. 8559 of *Lecture Notes in Computer Science*, pp. 737–744, Springer, July 2014.

[15] R. Brummayer and A. Biere in *Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays*, Institute for Formal Models and Verification Johannes Kepler University Linz, Austria.

[16] A. Niemetz and M. Preiner, "Bitwuzla at the SMT-COMP 2020," *CoRR*, vol. abs/2006.01621, 2020.

[17] A. limited, "c Data Types - Handbook." https://os.mbed.com/handbook/C-Data-Types.

[18] M. Brain, F. Schanda, and Y. Sun, "Building better bit-blasting for floating-point problems," in *Tools and Algorithms for the Construction and Analysis of Systems* (T. Vojnar and L. Zhang, eds.), (Cham), pp. 79–98, Springer International Publishing, 2019.

[19] M. George, "Comparative review of floating-point multiplier systems," *International Journal of Hybrid Information Technology*, vol. 12, pp. 21–48, 11 2019.