Generating word-level floating-point benchmarks multiplier

Bachelor's Thesis

for submission in

Chair of Computer Architecture

at the university

Albert-Ludwigs-University Freiburg

from

Benny Wennberg

19/01/2023

Processing period	19/10/2023 - 19/01/2024
Matriculation number	4949782
Chair	Chair of Computer Architecture
Supervisor	Prof. Dr. Armin Biere
Reviewer	Dr. Mathias Fleury

Contents

List of source codes

List of Figures

Abstract

The field of Satisfiability Modulo Theories (SMT) deals with determining whether logical formulas are satisfiable within various first-order theories. One of these theories involves the use of bit-vectors to represent floating-point arithmetic in hardware.

This thesis focuses on exploring the complexities of floating-point multiplication, including its commutativity. To achieve this, we have designed an algorithm to generate SMT solver benchmarks that emphasize the difficulty of multiplying floating-point numbers. We tested this algorithm on six different SMT solvers and found that even basic multiplications of floating-point numbers can be challenging.

We further extend our algorithm to highlight the intricacies introduced by the non-commutative nature of floating-point numbers. Our comprehensive exploration deepens our understanding of the complexities of hardware-implemented floating-point arithmetic, specifically emphasizing the nature of multiplication and its implications for the broader domain of SMT solving.

1 Introduction

Electronic devices have become an indispensable part of our lives, making it crucial to ensure the reliability and correctness of both software and hardware. For example, when it comes to safety-critical systems, like the control software for self-driving drones, it becomes essential to verify the accuracy of the software. This involves dealing with complex logical constraints, including real-time constraints and algorithms governing collision avoidance. To ensure the veracity of such complex logical formulas that embody a myriad of first-order theories, Satisfiability Modulo Theories Solvers (SMT solvers) are used.

The SMT solver expands the Boolean satisfiability problem (SAT) to handle more complicated formulas expressed in first-order logic, accommodating various first-order theories such as arrays, strings, bit vectors, and floating-point numbers. The SMT solver's effectiveness in automating the verification of hardware and software designs makes it a critical tool for ensuring safety and reliability.

This thesis focuses on the implementation of floating-point arithmetic in hardware, with a specific emphasis on the multiplication of floating-point numbers using the GRS technique to round intermediate results. The multiplication is performed on a word-level, not on bit-level. The representation of floating-point numbers leverages fixed-sized bit-vectors for the sign, exponent, and significand.

The thesis introduces diverse benchmark sets, emphasizing the commutativity of floating-point numbers, to challenge state-of-the-art bit-vector SMT solvers like Z3, Bitwuzla, and Boolector. Given the inherent difficulty in checking integer multiplication, the complexity of floating-point multiplication is similarly challenging.

The thesis first provides background information and definitions on the theoretical aspects, including floating-point numbers and SMT solvers. It then elucidates the algorithm developed for floating-point multiplication. Finally, the thesis presents results and conclusions, offering insights into the challenges posed by floating-point arithmetic, particularly in multiplication operations.

2 Theoretical part

2.1 Floating-point

Floating-point arithmetic is a method used by computer systems to represent real numbers. It uses a finite set of binary digits to achieve this. To understand it better, we need to break down the individual components that make up a floating-point number. Each of these components plays a vital role in determining the precision and defining the numerical range.

2.1.1 Floating-point notation

Real numbers in floating-point arithmetic are represented by a sign bit, exponent, and significand. These three components collectively define the representation of real numbers.
The floating-point number x is notated by:

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

This notation shows how a floating-point number is constructed, using the following variables:

Sign Bit (s): determines the sign of the number. It is a single bit, representing either 1 for negative numbers or 0 for positive numbers.

Exponent (e): represents the scale factor applied to the significand to define the actual value's magnitude. In the floating-point format, the exponent is stored with a bias, which adjusts the range of representable exponents.

Significand (m): denotes the precision of the floating-point number. It comprises an integer with a fixed number of bits, typically normalized to include one digit before the radix point and 'p-1' digits after it (where 'p' defines the precision).

Base (b): signifies the radix number used in the floating-point representation. In this context, the base is typically 10, indicating a decimal representation.

Understanding the structure and representation of floating-point numbers is essential for accurate numerical computation and represnetation. [1]

In the subsequent sections, we will explore the IEEE 754 standard, which defines the formats for representing floating-point numbers in most modern computer systems, addressing the structure of these numbers and the methods for interpreting and performing arithmetic operations on them.

2.2 IEEE 754: floating point

The IEEE 754 standard is a crucial aspect of modern computing. It provides a set of guidelines for representing and processing floating-point arithmetic. The standard defines two primary data formats for binary floating-point numbers: 32-bit (single precision) and 64-bit (double precision). It also includes two extended format types, such as 128-bit and 256-bit.
By following these guidelines of this section, the standard ensures that rounding methods are consistent and that overflows and underflows are handled correctly, reducing errors in floating-point calculations.

2.2.1 IEE754 representation

The IEEE 754 standard defines various floating-point formats, including single (32-bit) and double (64-bit) precision formats. Each format has a fixed bit size for the sign, significand, and exponent. [2]

Name	Common Name	Sign	Exponent	Significand
binary16	Half precision	1	5	10
binary32	Single precision	1	8	23
binary64	Double precision	1	11	52
binary128	Octuple precision	1	15	112
binary256	Quadruple precision	1	19	236

Table 2: IEEE754 formats

To represent a floating point number, it is broken down into three bit sequences:

- **S**: A sign bit (1 bit) to indicate whether the number is positive or negative
- **M**: A significand (p bits) for the significand
- **E**: An exponent (r bits) to determine the magnitude of the number

The sign bit, denoted as 'S', is set to 0 for positive numbers and 1 for negative numbers. Using this bit, we can determine the sign of the number using the formula:

$$s = (-1)^S$$

The exponent, denoted as 'E', is a non-negative binary value that is determined by adding a fixed bias value 'B' to the actual exponent 'e'. $'B' : E = e + B$.
The bias value is calculated as $2^{(r-1)}-1$, and is used to enable the representation of negative exponents using an unsigned number. The actual exponent 'e' can be found using the formula:

$$e = E - B$$

The significand, denoted as 'm', is a value derived from the p mantissa bits with a value 'M', calculated as $m = 1 + \frac{M}{2^p}$ and represents the fractional part of the number. To obtain the significand, we use the formula:

$$m = 1.M = 1 + \frac{M}{2^p}$$

This formula works because normalization (Sect. 2.3.3) ensures that $1 \leq m < 2$ for all representable numbers. This means that the first bit of the mantissa is always 1, so it doesn't need to be stored. By removing this redundant bit, we can gain an additional bit of precision.

2.2.2 Number formats of the IEEE-754 standard

The IEEE 754 standard specifies how floating-point numbers are represented in binary code. It does so by assigned bits for the sign, exponent, and mantissa fields. The standard defines four different representations for binary floating point numbers, namely binary16, binary32, binary64, and binary128. Furthermore, the standard also permits the creation of user-defined extended representations that follow the same principles as the predefined representations. [3]

Name	Bitsize	Values of the exponent	Bias
binary16	16	-14 $\leq$ e $\leq$ 15	15
binary32	32	-126 $\leq$ e $\leq$ 127	127
binary64	64	-1022 $\leq$ e $\leq$ 1023	1023
binary128	128	-16382 $\leq$ e $\leq$ 16383	16383
binary256	256	-262142 $\leq$ e $\leq$ 262143	262143

Table 3: IEEE754 formats

You can see the representations of the IEEE754 in the graphic below(Figure 1.). The size of the memory depends on the IEEE754 standard.

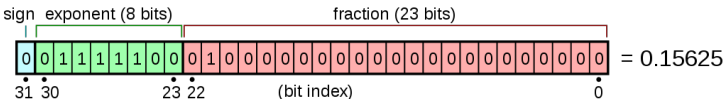

Figure 1: binary32 format [2]

The arrangement with sign - exponent - mantissa in exactly this order puts the displayed floating point values in the same order as the integer values that the same bit pattern can represent. This means that the same operations can be used for comparing floating point numbers as for comparing integers.

2.2.3 Handling Special Cases

The IEEE 754 standard contains provisions for handling special cases in floating-point arithmetic, such as infinity, NaN (Not-a-Number), and denormalized numbers. These cases are necessary to represent floating point numbers exactly and to deal with operations that lead to undefined results, such as multiplication by a denormalized number or other operations that produce undefined results in single precision.[1]

Denormalized Numbers (Subnormal Numbers) Denormalized numbers, or subnormal numbers, occur when the exponent consists of all zeros, while the mantissa holds a non-zero value. This representation accommodates very small numbers that fall below the range of the smallest normalized number, existing across various IEEE 754 formats. The representation of a floating point is no longer ± 1.mantissa, but ± 0.mantissa

Infinity is a state in which the exponent is all ones and the mantissa is all zeros. This state is crucial in handling scenarios where a computed value goes beyond the maximum representable limit within the available formats. It comes into play when there is an overflow, that is, when a computed value exceeds the maximum representable limit across all IEEE 754 formats.

NaN (Not-a-Number) represents an undefined or error state in mathematical operations. It occurs when the exponent has all ones and the mantissa has some non-zero digits. NaN is a specific value returned as a result of certain invalid operations such as $\frac{0}{0}$, $\infty \cdot 0$ or $\sqrt{-1}$).

In general, NaNs are propagated, meaning that most operations involving a NaN result in a NaN.

There are two types of NaN:

- silent NaN (NaNq - quiet)
- signaling NaN (NaNs - signaling)

Both explicitly do not represent numerical values. [4]

Signaling NaN can be used to fill uninitialized memory in the computer, whereby any use of an uninitialized variable automatically triggers an exception and its highest bit of the mantissa is set (1).
Silent NaN makes it possible to handle calculations that cannot produce a valid result, for example, the division of zero by zero and its highest bit of the mantissa is not set (0).
Additional information can be contained in the remaining mantissa bits, for example, to indicate the cause of the NaN.

Datum	Sign	Exponent	Significand
+0	0	00000000	00000000000000000000000
-0	1	00000000	00000000000000000000000
Denormalized numbers	0/1	00000000	nonzero
$+\infty$	0	11111111	00000000000000000000000
$+\infty$	0	11111111	00000000000000000000000
qNaN	0	11111111	1xxxxxxxxxxxxxxxxxxxxxx
sNaN	0	11111111	0 nonezero

Table 4: Special cases in single precision

In the subsequent sections, we will explore the practical implications of the IEEE 754 standard in floating-point multiplication and its implementation in computational algorithms.

2.3 Floating Point Multiplication

We will be discussing the basics of multiplying floating point numbers in this section. We will examine the rules and special cases of floating point multiplication, which is a crucial operation in computational algorithms and is used in various scientific and engineering applications.

Floating-point multiplication refers to the process of multiplying two numbers that are represented in floating-point format. This operation is performed based on the guidelines defined by the IEEE 754 standard (refer to section 2.2.1). To perform the multiplication, the sign, exponent, and mantissa of the numbers are taken into account, and the result is calculated.
Let's say we have two floating point numbers - A and B. These numbers can be represented as follows:
$A = (-1)^{S_a} \cdot M_a \cdot 2^{E_a}$ and $B = (-1)^{S_b} \cdot M_b \cdot 2^{E_b}$
Using this formula, we can multiply these two given floating-point numbers.

$$S_a.E_a.M_a \times S_b.E_b.M_b = (S_a \oplus S_b).(E_a + E_b) - bias.M_b \times M_a.$$

We use a multiplier that XORs the two signs, multiplies the mantissa of the two given numbers, and adds the two exponents, which are subtracted with the bias of the given format.[5]

2.3.1 Calculation Rules

Multiplying floating-point numbers follows specific rules that are governed by the IEEE 754 standard. These rules dictate how to manipulate the exponent and mantissa to compute the product, considering normalization, scaling, and rounding procedures to ensure accuracy within the limitations of the floating-point representation.

Sign Floating-point numbers have a sign bit indicating whether the number is positive or negative. In multiplication, the signs of the multiplicands determine the sign of the product as XOR.
Here are the sign rules:

- Positive · positive = positive
- Negative · negative = positive

- Positive · negative = negative
- Negative · positive = negative

Mantissa When multiplying two floating-point numbers, their significands are multiplied.
To ensure that the significands are multiplied correctly, there are several guidelines to follow:

- The significands are typically in normalized form, where the leading bit is assumed to be 1 and is not explicitly stored.
- The product of the significands results in a new significand, which may need to be normalized to the proper form.
- Multiplying the significands might lead to an overflow or underflow, which may require adjusting the exponent.

Exponent The exponents of the floating-point numbers being multiplied are added to determine the exponent of the result. The values of E_a and E_b are added together. The result of the addition is subtracted with the bias value.

2.3.2 Normalization

When a number is stored in a computer memory, normalization involves representing it in a standardized form. This form should have the leading bit of the mantissa set to 1. If the result of a mathematical operation isn't in normalized form, it needs to be shifted by adjusting the exponent. This is done to set the leading bit of the mantissa to 1.
For example, the result of an operation might be in the form $10.101 \cdot 2^3$. To normalize it, the exponent needs to be incremented until the leading bit of the mantissa becomes 1. This would look like $1.0101 \cdot 2^4$. [5]

2.3.3 Rounding

When performing arithmetic operations like addition or multiplication, the resulting number may not always fit precisely within the precision of the floating point format. In such cases, the number needs to be rounded to meet the specific precision or format requirements of the floating point representation. The rounding

process involves the application of specific rounding modes.
There are three methods for rounding numbers: [2]

- **Round towards Zero:** In this method, the number is always rounded towards zero, regardless of whether the decimal digits are greater or less than 5.
- **Round towards Positive Infinity/Negative Infinity:** In this method, the number is rounded towards positive or negative infinity, depending on whether it is positive or negative.
- **Round to Nearest:** In this method, the number is rounded to the nearest representable number, either up or down, depending on which side is closer.

In this thesis, the rounding mode employed is **Round to Nearest**. To guarantee precision during rounding, the significand is rounded using a method called "Guard, Round, and Sticky" (GRS), which involves three bits: the guard, round, and sticky bits. [6]
For bfloat16, the final significand of the result is limited to 10 bits (see 2.2.1). If the initial significand exceeds this specified bit size, the GRS mechanism adapts by taking into account the specified bit size along with an additional 3 bits.

$$1.0001111111\overbrace{1}^{G}\overbrace{1}^{R}\overbrace{000000000}^{S}$$

$$1.0001111111\overbrace{1}^{G}\overbrace{1}^{R}\overbrace{0}^{S}\cancel{00000000}$$

The purpose of using the GRS mechanism is to improve the accuracy of rounding operations. The three bits (guard, round, and sticky) make the rounding process more specific. The "guard" bit acts as a sentinel, determining whether rounding is necessary based on the discarded bits. The "round" bit indicates whether rounding should occur exactly at the midpoint, and the "sticky" bit captures any remaining bits beyond the designated precision.
The GRS method is especially beneficial in cases where precision is essential, ensuring that rounding operations maintain accuracy while complying to the bit limitations imposed by the bfloat16 format.

To decide whether we round up or down a closer look at the GRS-bits is required.

Significand	G R S	rounded	result
1.0001111111	0 1 1	down	1.0001111111
1.0001111110	1 0 0	down	1.0001111110
1.0001111111	1 0 0	up	1.0010000000
1.0001111110	1 1 0	up	1.0001111111
1.0001100110	1 1 1	up	1.0001100111

Table 5: GRS

In the case mentioned, where the significand is 1.0001111111 and the GRS bits are G=1, R=1, and S=0, according to the GRS method, this configuration signifies rounding up.

1.0010000000

The "guard" bit being 1 indicates that there were '1' bits that were discarded beyond the precision of the target format (bfloat16). In addition, the "round" bit being 1 means that the original number is closer to the next higher representable value. Therefore, considering the GRS bits, especially the guard and round bits both being '1,' it suggests that the rounding operation should be upwards to ensure the closest representation within the limited bits available.

2.3.4 Over- And Underflow

It's important to handle over- and underflow scenarios in computational operations to maintain accurate results. When a result falls outside the range that can be represented by a floating-point format, an overflow or underflow may occur. [7]

- Overflow occurs when the resulting exponent exceeds the maximum value displayable in the IEEE 754 standard. For Binary32 format, this means the exponent field is outside the range of +127, and the number is represented as "infinity".
- Underflow happens when the exponent after the computation becomes smaller than the minimum value that can be displayed according to the IEEE 754 format (see table 3). In such cases, the resulting number is often displayed as zero since it is slightly above zero but too small to be displayed.

Correct handling of overflow and underflow is crucial to maintain numerical accuracy. It ensures that values outside the range are handled according to floating-point specifications.

2.3.5 Dealing with Special Cases

Multiplication of floating-point numbers encounters various special cases, including handling zero, infinity, denormalized numbers, and NaN values. Several special cases can occur during multiplication:

1. **Zero Multiplication** A normal number (0) multiplied by any other number (positive or negative) always results in 0.
2. **Multiplication by Infinity**
   - A normal number multiplied by Infinity (either positive or negative) results in either Infinity (with the same sign) or Negative Infinity (with different signs), depending on the signs of the involved numbers.
   - Multiplying 0 by Infinity results in NaN.
3. **Multiplication by NaN** Any number multiplied by NaN (Not-a-Number) always results in NaN.

These special cases demand specific treatment to ensure accurate results during multiplication operations involving floating-point numbers.[1]

2.3.6 Visualization and Analysis of Floating-Point Multiplication

Let's consider an example to visualize how floating-point numbers are multiplied. We have two bfloat16 numbers, x and y. The format of these numbers is as follows:

x = 0 | 01111 | 1111111111
y = 1 | 00001 | 0000000001

To begin with, we take the sign of both numbers and apply the XOR operation to them.

$$0 \oplus 1 = 1$$

Next, we calculate the exponent. We add the exponents of x and y and then subtract the bias (see table 4) from the sum.

$$01111 + 00001 - 01111 = 00001$$

After multiplying the mantissas of x and y, we expand the width of the resulting mantissa by 2 and extract the first half for rounding.

$$1.1111111111 \cdot 1.0000000001 = 10.00111111101110000000$$

Since the leading bit of the mantissa isn't 1, we shift the mantissa to normalize it. We increment the exponent until the leading bit of the mantissa becomes 1.

$$10.00111111101110000000 \rightarrow 1.00011111101110000000 + 1 \text{ exponent}$$

After we shift the mantissa, we add one to the exponent.

$$00001 + 1 = 00010$$

To ensure the accuracy of floating-point calculations, we round the mantissa based on the GRS method.

$$1.0001111111 \overbrace{0}^{\text{G}} \overbrace{1}^{\text{R}} \overbrace{110000000}^{\text{S}}$$

G = 0
R = 1
S = 1
Mantissa = 0001111111

As a result, the guard bit is set with a 0, the round bit is set with the value 1, and the sticky bit is set with the value 1. Therefore, the mantissa is not rounded up.

Finally, we combine the sign, exponent and mantissa to get the result of the multiplication of x and y.

0 01111 1111111111 · 1 00001 0000000001 = 1 00010 0001111111

Floating-point multiplication plays a significant role in numerical algorithms and scientific computations. [8]

However, it also introduces challenges due to finite precision and potential loss of accuracy. In the subsequent sections, we will explore practical implementations and algorithms related to floating-point multiplication, providing insights into optimizing and handling these computations in real-world scenarios.

2.4 Satisfiability Modulo Theories Solver

An SMT solver, which stands for Satisfiability Modulo Theories Solver, is a software that specializes in determining whether logical formulas can be satisfied within a specific theoretical context. Essentially, an SMT solver checks whether a given logical formula or set of conditions can be satisfied in a particular theory, taking into account the rules and constraints of that theory. Unlike the Boolean Satisfiability Problem (SAT), which deals only with Boolean variables, SMT solvers are designed to handle more complex formulas expressed in first-order logic. These solvers can handle different first-order theories, including integer numbers, arrays, strings, bit vectors, and floating-point numbers. [9]
SMT solvers, such as Z3, are equipped with specialized algorithms and reasoning methodologies that allow them to navigate complex logical formulas. These solvers are particularly useful in handling complex constraints and traversing multiple theories.

One example of an SMT solver is Z3, developed by Microsoft, which is a cross-platform solver that supports a wide range of theories, including equality and uninterpreted functions (EUF), linear arithmetic, arrays, bit-vectors, algebraic datatypes, sequences, and strings.
Z3 supports a variety of input languages, including its own native input language, SMT-LIB v2.6 script language, and APIs for C++, C, Python, Java, .Net, and Ocaml.

2.4.1 SMT-LIB2

SMT-LIB (Satisfiability Modulo Theories Library) is a standardized format and a collection of rules that define a common language and syntax for communicating with Satisfiability Modulo Theories (SMT) solvers. SMT-LIB2 refers specifically to the second version of this library.

SMT-LIB2 provides a set of standardized input and output formats for expressing logical formulas and theories that SMT solvers can process. It offers a language interface, which allows users to specify logical problems and interact with SMT solvers. This standardized format includes provisions for defining logical constraints, specifying theories (such as arithmetic, arrays, bit-vectors, etc.), asserting formulas, querying for satisfiability, and retrieving solver results. [10]

To create benchmarks for a project in SMT-Lib2, you need to use specific line

commands in a cpp file. These line commands will help you create an SMT-Lib2 file, which an SMT solver can then use to solve the solvability. Here's an example of how you can convert an SMT-Lib2 file using a cpp file to make it solvable by an SMT solver.

```
1  (set-info :source |Benny Wennberg Bachelor thesis|)
2  (set-option :produce-models true )
3  (set-logic QF_BV)
4
5  ------------
6  ;;; bfloat16 multiplication binary
7  # Code
8  -----------
9  (check-sat)
10 (get-model)
11 (exit)
```

Listing 1: SMT File

The SMT2 file is first initiated within a C++ file, where its name and source information are defined. Different options can be set, as in this case, where the 'produce-models' option is set to true. This option instructs the SMT solver to generate models when a feasible solution exists for the provided problem.
The logic declared at line 3, 'QF_BV,' stands for "Quantifier-Free Bit-Vector Logic." It denotes logical formulas based on the bit vector utilized within the corresponding code block.
At the end of an SMT file, the 'check-sat' command is used to check the existence of a solution for the problem. If a solution is feasible, the solver responds with 'sat'; otherwise, it indicates 'unsat.' Finally, the 'get-model' command retrieves the possible solution and its associated variables from the code.

Following this setup, the subsequent code demonstrates the output of an SMT process represented a floating-point multiplication of 2 floating-point numbers: 'x · y = z':

```
sat
(model
    (define-fun x () (_ BitVec 16)
    #xBC01)
    (define-fun y () (_ BitVec 16)
    #x7B807)
    (define-fun z () (_ BitVec 16)
    #x3808)
)
```

Listing 2: SMT Multiplication Output

2.4.2 SMT logic

SMT logic revolves around determining the satisfiability of logical formulas while considering various mathematical theories. These theories cover a wide range of domains, including arithmetic, arrays, bit-vectors, and many others, providing a framework for solving complex computational problems.

In the context of SMT, each logic represents a specific area of mathematical knowledge, defining a set of rules, constraints, and operations. [11]

QF_BV (Quantifier-Free Bit-Vector Logic) is a logic that specializes in dealing with bit-vectors. This logic allows performing arithmetic operations such as addition, subtraction, multiplication, and bitwise operations like AND, OR, and XOR on bit-vectors.

QF_BVFP (Quantifier-Free Bit-Vector Floating Point Logic) extends QF_BV logic to include floating-point numbers. Unlike bit-vectors with fixed size, floating-point numbers represent values with variable precision and exponent ranges.

These logics are quantifier-free, meaning there are no existential or universal quantifiers in the statements.

3 Generation of the benchmarks

When dealing with floating-point arithmetic, some mathematical properties that hold in real number arithmetic may not be preserved. For example, properties like associativity and distributivity may not necessarily hold in floating-point arithmetic. Only when arithmetic operations are correctly rounded, do certain properties like commutativity in floating-point multiplication remain valid.
The thesis describes an algorithm that generates unique benchmarks designed for floating-point multiplication. The focus is on exploring the commutative properties of floating-point arithmetic. These benchmarks aim to showcase the complexity of floating-point multiplication. They are designed to evaluate the capability of various solvers in handling commutativity in floating-point multiplication operations.

Implemented in C++, the algorithm accepts inputs denoting the sizes of both the sing and the exponent and also the stored significand, which defines the format of floating-point numbers. Subsequently, the algorithm generates benchmarks in the SMT-Lib2 language, specifically designed for compatibility with SMT solvers.

The first four SMT solvers selected for testing include solvers that are widely recognized for their support of different theories and logics in the SMT-Lib2 language. These solvers are Z3 developed by Microsoft Research, CVC4 and cvc5 from Stanford University and the University of Iowa, and Yices 2 from SRI International. These solvers offer broad functionality across multiple theories and logics, making them suitable for evaluating floating point arithmetic benchmarks. In addition to the evaluation of general SMT solvers, the evaluation in this study also includes two specialized solvers that are specifically designed for handling bit-vectors. The first solver is Boolector, which is known for its ability to efficiently handle bit-vectors and arrays. The Bituwzla is then considered, which is efficient in managing bit-vectors, floating-points, arrays, and uninterpreted functions.

The comparison involves subjecting all benchmarks to thorough testing across these six SMT solvers, examining their capabilities, efficiency, and accuracy in handling floating-point multiplication benchmarks, especially in contexts related to commutativity.

3.1 Main Algorithm

In this section, the main algorithm of the thesis is described. Firstly, a simple floating-point multiplication algorithm is created, which exclusively uses the syntax inherent to quantifier-free bit-vector logic (QF_BV). Afterward, the algorithm is tested for its ability to establish the commutative property $a \cdot b = b \cdot a$, and then modified accordingly. Additionally, the algorithm's compatibility with floating-point operations is evaluated using the QF_BVFP logic.

This evaluation aims to demonstrate the algorithm's capacity to establish commutativity in floating-point multiplication using different logics and solvers.

The first step of the algorithm, written in C++, is to prompt the user to input the number of bits they want for their floating-point numbers. The user needs to provide an integer value that represents the number of bits they want to use. It checks which format should be used for IEEE 754 calculations. Depending on the input provided by the user, which can be 8, 16, 32, or 64 bits, the algorithm defines the fixed sign, mantissa, and exponent sizes (Table 3). This process ensures that computations are performed in the appropriate format, in line with the IEEE 754 standard.
The algorithm produces a .smt2 file as its output, which is then used with SMT solvers for evaluation and analysis. In cases where a comparison is required between Floating Point (FP) and Bit-Vector (BV), the algorithm ensures that the result obtained from two floating-point numbers is identical, regardless of whether it was generated using FP's syntax or BV's syntax. The generated .smt2 file is then checked and tested by the SMT solver for further evaluation and verification.

3.1.1 Simple floating-point multplier

To develop a benchmark focusing on multiplying two floating-point numbers using quantifier-free bit-vector (QF_BV) logic, the initial step involves the creation of a simple floating-point multiplier. This multiplier is built exclusively within the constraints of QF_BV logic, limiting the usage to fixed-sized bit-vectors and disallowing the utilization of any preimplemented floating-point data types within an SMT solver. The multiplier itself is coded in C++ and later translated into SMT-Lib2 format, which makes it usable across different SMT solvers. The algorithm prompts the user to specify the type of the IEEE754 formats(Table 2), allowing for the input of parameters such as sign, exponent, and mantissa. After declaring the type, the C++ algorithm codes an SMT-Lib2 file.

To create the SMT2 file, the header of the smt2 file needs to be declared first (Listing 3). The first command within the header provides essential details about the file and its intended purpose. The second command compels the solver to generate a model representing potential values of the declared variables. However, the solver only produces this model solely when the algorithm is satisfiable; otherwise, an error is triggered. Lastly, the third command sets the SMT logic. By setting the logic, the smt-solver knows the theories employed within the code and can use the optimal theory solver to solve it.

```
(set-info :source |Benny Wennberg Bachelorthesis|)
(set-option :produce-models true)
(set-logic QF_BV)
```

Listing 3: Header Declaration

After creating the header, the next step is to code the actual multiplier in the SMT-Lib2 algorithm. In the SMT-Lib2 algorithm, the two random numbers generated by C++ are asserted to x and y. These numbers are then divided into three parts, the sign, the exponent, and the significand, and stored as bit-vectors. The size of the bit-vectors is adjusted depending on the size of the format:

```
;;; Declaration of x
(declare-fun x () (_ BitVec 16))
(declare-fun x_sign () (_ BitVec 1))
(declare-fun x_exponent () (_ BitVec 5))
(declare-fun x_mantissa () (_ BitVec 10))
;;; Declaration of y
(declare-fun y () (_ BitVec 16))
(declare-fun y_sign () (_ BitVec 1))
(declare-fun y_exponent () (_ BitVec 5))
(declare-fun y_mantissa () (_ BitVec 10))
```

Listing 4: Varibale Declaration of bfloat16

The following code contains calculation rules, explained in section 2.3, for a basic floating-point multiplication operation that doesn't take any special conditions into account.

- **Z_sign calculation:** performs an XOR operation (bvxor) on the variables x_sign and y_sign, and assigns the result to z_sign.
- **Z_exponent calculation** calculates the exponent of the result z. It first adds the exponents x_exponent and y_exponent using bvadd, then subtracts the bias using bvsub.
- **Z_mantissa calculation** computes the significand of the result z by performing a simple multiplication operation (bvmul) on the mantissas x_mantissa and y_mantissa.

```
;;; Calculation of  z_sign
(assert (= z_sign (bvxor x_sign y_sign)))
;;; Calculation of z_exponent
(assert (= z_exponent(bvsub (bvadd x_exponent y_exponent) bias)))
;;; Calculation of z_mantissa
(assert (= z_mantissa_extra (bvmul x_mantissa_extra y_mantissa_extra)))
```

Listing 5: Calculation of sign, exponent, and significand

To ensure accurate calculations of floating point numbers, special considerations are taken into account, These considerations include handling denormalized numbers, normalization, rounding with GRS, and dealing with over- and underflow. In the case of a denormalized number, the first bit that is set is 0 instead of 1, as it is with a normal number. If the multiplication results in a mantissa that does not match the corresponding form of normalization, the number is shifted to the left until the first representing number is a 1.

Depending on the mantissa, the exponent is incremented by one if the z_mantissa is left-shifted. After ensuring that the first bit is 1, we check if any of the inputs are equal to zero. If this is the case, we modify the remaining calculation steps to ensure that the result is not a normal number, but rather a denormalized number. The occurrence of infinity and NaN can also result in multiplication by zero. Checking the exponent will help determine if either of these cases has occurred.

```
;;; Denormalized numbers change the form of the given significand of x and y
(assert (= x_mantissa_extra  (ite (= x_exponent denormalized_number)(concat zero x_mantissa) (concat one x_mantissa))))

;;; Normalization by controlling the first Bit before mantissa*2
(assert (= z_mantissa (ite (= extra_bit null) ((_ extract 20 11) z_mantissa_extra) ((_ extract 19 10) z_mantissa_extra))))

;;; If normalization is necessary, then add expo by one
(assert (= z_exponent_GRS (ite (and (or (and (= extra_bit_2 one)(= extra_bit zero))(and (= extra_bit_2 one)(= extra_bit zero)))(distinct z_exponent bias_inf_compare)) (bvadd z_exponent add_expo) z_exponent)))

;;; Controlling if one of the inputs is zero
(assert (= z_exponent_zero (ite (or (and (= x_exponent comp_expo) (= x_mantissa_extra comp_mant )) (and (= y_exponent comp_expo) (= y_mantissa_extra comp_mant))) comp_expo z_exponent_GRS)))

;;; checks if overflow occurs
(assert (= z_exponent_overflow (ite (or (= x_exponent comp_expo_1) (= y_exponent comp_expo_1)(= z_exponent_GRS comp_expo_1) (bvugt z_exponent bias_compare_overflow)) comp_expo_1 z_exponent_zero)))
```

Listing 6: Denormalized Numbers, Normalization & Overflow in bfloat16

In case the value of a number is too small to be represented, it leads to an underflow. In such situations, the exponent is modified to 0, and the mantissa must be shifted to the right. To keep track of the number of right shifts in z_mantissa, we use a variable called "count_underflow". We calculate the difference between the lowest possible exponent value (Table 3) and the calculated value of the z_exponent (Listing 5). This helps us ensure that the mantissa is shifted towards

zero depending on the count_underflow value. To achieve this, we use a for-loop to calculate each step until we reach the shifted mantissa value. If an overflow occurs, we manipulate the z_mantissa depending on the inputs of x and y, as well as the calculated value of z_mantissa that is listed in Listing 5.

```
1  ;;; If an underflow occurs
2  (assert (= count_underflow (ite (bvugt bias_compare_underflow
   ↪  z_exponent_extra_help ) (bvsub bias_extra z_exponent_extra_help)
   ↪  (concat null comp_expo))))
3
4  ;;; count the difference between the lowest possible exponent value and
   ↪  the calculated value of the z_exponent
5  (assert (= count_underflow (ite (bvugt bias_compare_underflow  z_exponent
   ↪  ) (bvsub bias_compare_underflow z_exponent) (concat zero
   ↪  comp_expo))))
6
7  ;;; manipulate z_mantissa depending on the value of count_underflow and
   ↪  z_exponent
8  (assert (= z_mantissa_underflow_0 (ite (= z_exponent_underflow comp_expo)
   ↪  (bvlshr
9  z_mantissa_overflow shift_mant) z_mantissa_overflow)))
10 ;;; first step of the loop if an underflow occurs
11 (assert (= z_mantissa_underflow_1 (ite  (and (= z_exponent_underflow
   ↪  comp_expo)(bvugt count_underflow shift_time_underflow_0)) (bvlshr
   ↪  z_mantissa_underflow_0 shift_mant) z_mantissa_underflow_0)))
12
```

Listing 7: Underflow & Overflow in bfloat16

Once we have calculated the mantissa of the desired result, we then proceed to detect the GRS (Guard, Round, and Sticky) bits in the Mantissa to "Round to Nearest, Ties to Even". We extract the first half of the bit size, which represents the current state of the z_mantissa, and detect the least significant bit (LSB), the guard bit G, the round bit R, and the sticky bit S, as we previously described in section 2.2.3. Using the GRS bits, we can round up or down the z_mantissa,

depending on the values of these bits.

```
;;; Rounding using method GRS
(assert (= z_mantissa_GRS(ite (or (and (= g one) (or (= r one) (distinct
↪  s compare_s)(= add_outshift_number_29 eins))) (and (= g one) (=
↪  guard_bit one))) (bvadd z_mantissa add_mant) z_mantissa)))

;;; concat the results of sign, exponent and mantissa
(assert (= z (concat z_sign z_exponent )))
(assert (= z_finale (concat z z_mantissa)))

```

Listing 8: Rounding & Result in bfloat16

Afterward, we combine the sign, exponent, and mantissa results in the format IEEE754.

3.1.2 Double floating-point multiplier

After defining an algorithm for simple floating-point multiplication, the next step is to prove the commutativity by adding a double floating-point multiplier. Similar to the simple floating-point multiplicator the inputs are still the same by declaring the formats of the IEEE754(Sect. 3.1.1). To get an extra multiplier that calculates a value with two other variables, the declaration of the variable will be duplicated. To avoid overwriting similar variables, the additional variables will be named a and b instead of x and y. This means that a reflects the value of x and b reflects the value of y. The multiplication of both sets of inputs produces z from the multiplication of x and y and c from the multiplication of a and b. To check if z and c are equal, we use the assert function, which tells us if both outputs are solvable. If the output of z and c is the same, the SMT solver returns SAT.
In a variant of the algorithm, the variables x, y, a, and b are not assigned values. Instead, assertions are used to create constraints between the bit-vectors. By assigning a to the value of x and b to the value of y, the algorithm compares the respective results with an assert not operation.

```
// assigning variables
(assert (= x a))
(assert (= y b))

// comparing the outputs
(assert (distinct z c))
```

Listing 9: assigning variables by assertions

When using "assert not," the intention is to create a situation where the SMT solver cannot find a solution. This is achieved by presenting contradicting conditions or assertions which are impossible to be true at the same time. In this case, the solver is expected to recognize that it cannot meet these conditions, and hence, it will output "unsat." The main objective of these manipulations is to test the solver's capabilities and evaluate its behavior when it is faced with unsolvable or contradictory conditions.

If the multiplier's implementation is correct, setting up unsolvable assertions deliberately helps to ensure that the solver responds as expected and identifies the inconsistencies in the provided.

3.1.3 Double floating-point multiplier commutative

This section evaluates the commutativity property of the double floating-point multiplier algorithm. Commutativity in arithmetic operations refers to the property in which the order of operands does not affect the final result.

The double floating-point multiplier algorithm uses two distinct inputs, namely, x and y, and their counterparts a and b, respectively. These inputs undergo multiplication operations to generate z and c as in the previous section.

To verify the commutative property, the algorithm compares the products of x * y and a * b, represented by z and c, respectively, to ensure that they yield identical results as in the previous section. The only difference in this algorithm is the assertion at the end. The assertion confirms that x and b have the same value, and y and a have the same value. The third assertion confirms that z is not

```
// assigning variables
(assert (= x b))
(assert (= y a))

// comparing the outputs
(assert (distinct z c))
```

Listing 10: assigning variables by assertions proofing commutative

equivalent to c. This assertion is implemented so that the SMT solver returns unsat when implemented correctly.

3.2 Floating-point

In this section, we define the floating-point multiplier algorithm using the QF_BVFP logic. The difference between the QF_BVFP and QF_BV logics is that QF_BVFP has an integrated floating point type that makes it easier to multiply two floating-point numbers using operations.

3.2.1 Floating-point multiplier using QF_BVFP logic

To implement this algorithm, we replaced the QF_BV logic with QF_BVFP in the header.

We used the existing implementation of floating-point numbers and declared the variables with the logic. These variables are then multiplied to create a floating-point multiplication.

```
// Declaration of x, y und z with binary32
(declare-const x_fp (_ FloatingPoint 8 24 ))
(declare-const y_fp (_ FloatingPoint 8 24 ))
(declare-const z_fp (_ FloatingPoint 8 24 ))

// floating-point multiplication with round to nearest (RNE)
(assert (= z_fp (fp.mul RNE  x_fp y_fp)))
```

Listing 11: floating-point multiplication by using QF_BVFP in bfloat32

3.2.2 Double floating-point multiplier using QF_BVFP logic

In this benchmark test, we will once again compare the inputs x, y, a, and b, along with the two outputs c and z using the QF_BVFP logic. We will follow the same approach as in sections 3.1.2 and 3.1.3 to prove the commutativity of floating-point by utilizing QF_BVFP logic.

```
;;; double floating-point multiplication with round to nearest (RNE)
(assert (= z_fp (fp.mul RNE  x_fp y_fp)))
(assert (= c_fp (fp.mul RNE  a_fp b_fp)))
;;; assigning variables
(assert (= x_fp b_fp))
(assert (= y_fp a_fp))
;;; comparing the outputs
(assert (distinct z_fp c_fp))
```

Listing 12: double floating-point multiplication by using QF_BVFP

At the end of the algorithm, the values are compared, and the SMT solver checks the solvability, which in this case is unsat, similar to section 3.1.2.

3.2.3 Double floating-point multiplier by using QF_BV and QF_BVFP

In this thesis, we have compared two different logic systems, namely QF_BVFP and QF_BV, by using two double floating-point multipliers. In the next section, we will examine how the SMT solver implements floating-point arithmetic with 2 different SMT-logics and test for commutativity. Firstly, we will perform multiplication using an algorithm specially designed to multiply two floating-point numbers (Sect. 3.1.1). Then, we will conduct a separate multiplication using the floating-point type provided by the SMT solver within the QF_BVFP logic (Sect. 3.2.1). Finally, we will compare the outcomes of these two logics. To test the commutativity between logics, the variables defined with bit vectors are converted to the QF_BVFP logic.

```
;;; BV_variables convert into FP_variables
(assert (= x_bv ((_ to_fp 8 24) x)))
(assert (= y_bv ((_ to_fp 8 24) y)))
;;; assigning variables
(assert (= x_bv y_fp))
(assert (= y_bv x_fp))
;;; comparing the outputs
(assert (distinct ((_ to_fp 8 24) z_finale) z_fp))
```

Listing 13: double floating-point multiplication by using QF_BVFP and QF_BV

4 Evaluation

This section offers a comprehensive evaluation of various benchmarks and divides the results into three sections, including runtime calculations of the SMT solvers. The first section will introduce the experimental setup and describe which solver was used (Sect. 4.1). The second section will focus on the symmetry and commutativity of two identical floating-point multiplication operations (Sect. 4.2). Finally, in the last section, we will evaluate the benchmarks using the QF_VBFP logic (Sect. 4.3).

4.1 Experimental Setup

For the experimental setup, we used an Intel Core i7-1165G7 processor running at a speed of 4 x 2.8 - 4.7 GHz, along with 16 GB of memory. The Linux subsystem on Windows (WSL2) was utilized to test the different SMT solvers. The C program was coded in Ubuntu 22.04.3 LTS and overlaid onto the Linux subsystem. Depending on the benchmark in Section 3, we tested the following six SMT solvers and their respective versions, if possible:

Z3 version 4.8.12 - 64 bit	[12]
Yices 2.6.4	[13]
CVC4 version 1.8	[14]
cvc5 version 11.4.0	[15]
Boolector 3.2.3	[16]
Bitwuzla 0.3.0-dev	[17]

In order to measure the running time of a SMT solver, you can use the command "time timeout 3000 xy-solver .smt2". This command applies the SMT solver named xy-solver to a specific SMT2 file (file.smt2) while measuring execution time. The command sets the time limit for execution to 3000 seconds (timeout 3000), which is equivalent to 50 minutes.

4.2 Double floating-point multiplier using QF_BV logic

In this section, we will assess the benchmarks previously mentioned in section 3.3 across six different SMT solvers: Z3, cvc5, CVC4, Yices 2, Boolector, and

Bitwuzla. The benchmarks consist of an assertion that assigns variables x and y to values a and b respectively, and then measures the runtime for a double floating-point multiplication.

The first part of the evaluation examines how the SMT solvers perform when multiplying two floating-point numbers using the same operation and also test for commutativity.

Finally, we compare the solution time of the two sections and conclude.

4.2.1 Double floating-point multiplier the same operation

The algorithm, as explained in section 3.1.2, is designed to generate four different benchmarks. The size of the input provided by the user, which can be 16, 32, 64, or 128 bits, determines the fixed sign, mantissa, and exponent sizes. The algorithm covers all four floating-point data types that are described in the IEEE 754-2008 standard. The table below shows the running times of each SMT-solver for four given benchmarks.

$$x \cdot y = a \cdot b$$

SMT-Solver [s]	binary16	binary32	binary64	binary128
Z3	0m0.220s	0m1.136s	3m28.680s	> 50m
Yices 2	0m0.009s	0m0.012s	0m0.016s	0m0.030s
CVC4	0m0.106s	0m0.176s	0m0.313s	0m1.897s
cvc5	0m0.364s	0m0.414s	0m0.534s	0m1.576s
Bitwuzla	0m0.013s	0m0.019s	0m0.064s	0m0.127s
Boolector	0m0.010s	0m0.022s	0m0.040s	0m0.161s

In general, the more bits in an input value, the longer it takes for SMT solvers to execute. This is particularly noticeable with Z3 and CVC4 solvers. Z3 solver requires more time for calculations than other solvers, especially for larger binary formats (binary64 and binary128). For binary128 format, the execution time even exceeds the timeout of 50 minutes. Such timeouts may indicate the complexity of calculations or limitations of the SMT solver.

Yices 2 solver consistently has low execution times across all binary formats. The cvc5 solver shows solid performance across all binary formats and delivers consistent execution times.

Bitwuzla and Boolector solvers show comparatively low execution times, particularly for smaller binary formats.

4.2.2 Double-floating-point multiplication commutative

In the previous section, we conducted a test to determine which solver could solve two identical floating-point multiplications the fastest. To add variety to these tests, this section changes the assertion to x = b, y = a (similar to section 3.1.3), in order to calculate the runtime of the given solvers and analyze which one can solve these benchmarks faster.

The results of this section were tested on four floating-point data types described in the IEEE 754-2008 standard: binary16, binary32, binary64 and binary128. All benchmarks were tested on six solvers - Z3, cvc5, CVC4, Yices 2, Boolector, and Bitwuzla.

Commutative Case: $x \cdot y = b \cdot a$

SMT-Solver [s]	binary16	binary32	binary64	binary128
Z3	0m0.242s	0m1.319s	3m7.888s	> 50m
Yices 2	0m0.009s	0m0.012s	0m0.016s	0m0.030s
CVC4	0m0.058s	0m0.112s	0m0.293s	0m0.844s
cvc5	0m0.299s	0m0.380s	0m0.534s	0m1.394s
Bitwuzla	0m0.014s	0m0.022s	0m0.047s	0m0.074s
Boolector	0m0.008s	0m0.018s	0m0.052s	0m0.182s

It is observed that the execution times for a·b=a·b and a·b=b·a are similar since multiplication is commutative.

The overall performance of the SMT solvers remains consistent with Z3 taking more time, Yices 2 performing consistently fast, CVC4 and cvc5 showing solid performances, and Bitwuzla and Boolector showing low execution times. The relationship between bit length and execution times also remains consistent, with longer bit lengths leading to longer execution times, particularly with Z3. However, Z3 still times out at binary128, suggesting that it may struggle with the complexity of calculations in this format.

To summarise, the results of both tables show consistent patterns, with the commutativity of the multiplication retaining the basic structure of the execution times.

4.3 Floating-point multiplication using QF_BVFP logic

In this section, we utilize the quantifier-free bit-vector floating-point theory. This theory already provides a floating-point type that is readily available. The implementation will be found in section 3.2.1.

In the first part, we evaluate how the SMT solvers behave when we multiply two floating-point numbers that are of the data types that the SMT solver already provides. We use the floating-point theory to conduct two multiplications and also test for commutativity.

Lastly, we compare the implementation that uses QF_BV logic with the provided data types from the SMT solver.

4.3.1 Double floating-point multiplier using QF_BVFP logic

In this section, we conducted a benchmark test using the QF_BVFP logic, but excluded Boolector and Yices 2 as they do not support this logic. Our evaluation involved testing the double floating-point multiplier using QF_BVFP logic. The code we implemented for this can be found in section 3.2.2. During the evaluation, we ensured that the solver recognized that the same floating-point multiplier was done twice by switching the variables to proof commutative. Additionally, we compared the results and ran tests to check for any differences in runtime when the numbers in the second multiplier are switched. The results of these tests are presented in the subsequent table.

$$x \cdot y = a \cdot b$$

SMT-Solver [s]	binary16	binary32	binary64	binary128
Z3	23m4.557s	> 50m	> 50m	> 50m
CVC4	0m0.012s	0m0.011s	0m0.017s	0m0.021s
cvc5	0m0.330s	0m0.315s	0m0.344s	0m0.320s
Bitwuzla	0m0.039s	0m0.02s	0m0.02s	0m0.02s

Commutative Case: $x \cdot y = b \cdot a$

SMT-Solver [s]	binary16	binary32	binary 64	binary128
Z3	25m4.557s	> 50m	> 50m	> 50m
CVC4	0m0.008s	0m0.013s	0m0.022s	0m0.055s
cvc5	0m0.319s	0m0.312s	0m0.356s	0m0.348s
Bitwuzla	0m0.003s	0m0.003s	0m0.004s	0m0.003s

Some binary formats in Z3 are causing long solution times of more than 50 minutes in both ordinary and commutative cases. Comparatively, CVC4 and Bitwuzla are much faster than the other two solvers. Additionally, there is not much change in cvc5's performance when controlling commutative. It must be noted that the solver's performance can vary depending on the specific problem at hand. However, Bitwuzla consistently offers the fastest solution times in this case.

4.3.2 Double floating-point multiplier by using QF_BV and QF_BVFP

In this section, we assessed our implementation of floating point arithmetic by comparing it with the data types provided by the SMT solver. We conducted the comparison between the logics QF_BV and QF_BVFP by using the first multiplication part with QF_BV logic and adding another multiplication using QF_BVFP logic. Our aim was to examine how the SMT solver implements floating-point arithmetic with two different SMT logics and test for commutativity.

We tested our implementation on three different solvers: Bitwuzla, Z3, and cvc5. However, we found that cvc5 only works with Float32 or Float64 and does not support Float16 and Float128.

$x \cdot y = a \cdot b$

SMT-Solver [s]	binary16	binary32	binary64	binary128
Z3	13m 45.437s	> 50m	> 50m	> 50m
cvc5	-	> 50m	> 50m	-
Bitwuzla	13m 20.548s	> 50m	> 50m	> 50m

Commutative Case: $x \cdot y = b \cdot a$

SMT-Solver	binary16	binary32	binary64	binary128
Z3	25m 2.139s	> 50m	> 50m	> 50m
cvc5	-	> 50m	> 50m	-
Bitwuzla	19m 48.129s	> 50m	> 50m	> 50m

The SMT solvers Z3 and Bitwuzla were successful in handling the binary16 case within a reasonable time frame. However, both Z3 and Bitwuzla solvers encountered significant difficulties and took over 50 minutes to compute the binary32, binary64, and binary128 cases, indicating potential challenges in solving them. cvc5 had trouble with binary32 and binary64, while binary16 and binary128 were not supported.

The commutative case ($x \cdot y = b \cdot a$) displayed similar trends to the non-commutative case.

Bitwuzla performed slightly better than Z3, but it also faced challenges in solving the specified benchmarks.

The evaluation of the double floating-point multiplier using QF_BV and QF_BVFP logic has consistently shown difficulties in accommodating specific data types. This highlights the ongoing challenge in achieving comprehensive support for floating-point arithmetic. The limitations faced by these solvers underscore the challenges in solving double floating-point multiplier benchmarks, particularly for higher-precision types (binary32, binary64, and binary128) and in cases where commutativity is considered. The provided tables demonstrate the challenges faced by the three solvers in solving these benchmarks.

5 Conclusion

In this study, we have developed an algorithm for generating various benchmarks for multiplying floating-point numbers. The algorithm was designed to consider the implementation of floating point numbers as implemented in hardware, which involves using fixed-size bit vectors for the sign, exponent, and significand. We followed the rules of floating point multiplication, including taking into account issues such as underflow and rounding techniques like GRS.

To check the accuracy of the algorithm, we compared the results commutatively and with a floating-point multiplication that was already implemented in hardware. The initial benchmarks showed that SMT solvers can recognize the symmetry of repeating floating point multiplication and check commutativity.

In the second set of benchmarks, we found that not all SMT solvers can interpret the already implemented floating point numbers. Based on this observation, we compared the results of the two variants of floating-point multiplication to determine whether the SMT solvers can recognize the multiplication of two logics.

However, we discovered that the changes made to the algorithm posed several challenges, both for small and more demanding benchmarks, as the SMT solvers had difficulty solving the results before the timeout expired.

In conclusion, this study suggests that applying SMT solvers to the multiplication of floating point numbers presents certain challenges. Overall, this bachelor's thesis provides valuable insights into the application of SMT solvers to floating point multiplication.

References

[1] Ieee standard 754 floating point numbers, 2020. [Online]. Available: https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/.

[2] W. contributors, Floating-point arithmetic — Wikipedia, the free encyclopedia, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Floating-point_arithmetic&oldid=1195817215.

[3] P. Schnabel. "Gleitkommadarstellung / gleitkommazahlen." (), [Online]. Available: https://www.elektronik-kompendium.de/sites/dig/1807231.htm.

[4] Wikipedia contributors, Nan — Wikipedia, the free encyclopedia, 2023. [Online]. Available: https://de.wikipedia.org/wiki/NaN.

[5] 2.1.2 gleitkommazahlen. [Online]. Available: https://agra.informatik.uni-bremen.de/doc/lehrmat/wise0304/ra/kap2.1b.pdf.

[6] R. Trüby, Gleitkommazahlen/ floating-point, 2022. [Online]. Available: https://cca.informatik.uni-freiburg.de/grs-bits/grs.html.

[7] Floating point representation – basics, 2023. [Online]. Available: https://www.geeksforgeeks.org/floating-point-representation-basics/.

[8] Prof. Dr. Edmund Weitz, Ieee 754 calculator, 2023. [Online]. Available: http://weitz.de/ieee/r.

[9] Wikipedia contributors, Satisfiability modulo theories — Wikipedia, the free encyclopedia, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Satisfiability_modulo_theories&oldid=1194894209.

[10] Clark Barrett, Pascal Fontaine, Cesare Tinelli, The smt-lib standard version 2.6, 2021. [Online]. Available: https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2021-05-12.pdf.

[11] Clark Barrett, Pascal Fontaine, Cesare Tinelli, Smt-lib - the satisfiability modulo theories library, 2021. [Online]. Available: https://smtlib.cs.uiowa.edu/language.shtml.

[12] Nikolaj Bjørner, Lev Nachmanson, Z3, 2008. [Online]. Available: https://www.microsoft.com/en-us/research/project/z3-3/publications/.

[13] BrunoDutertre, Yices2, 2021. [Online]. Available: https://github.com/SRI-CSL/yices2.

[14] Clark Barrett, Cvc4, 2021. [Online]. Available: https://cvc4.github.io/documentation.html.

[15] Andrew Reynolds, Cvc5, 2023. [Online]. Available: https://github.com/cvc5/cvc5.

[16] mpreiner, aniemetz, arminbiere, Boolector, 2023. [Online]. Available: https://github.com/Boolector/boolector.

[17] Aina Niemetz, Mathias Preiner, Bitwuzla, 2023. [Online]. Available: https://bitwuzla.github.io/.

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Freiburg, 19/01/2024

Place, date

Signature