IT kDV 24 013

Degree project 15 credits

June 2024

Validation of HOL4's Formal Floating-Point Model

Hugo Eidmann

Bachelor's Programme in Computer Science

UPPSALA
UNIVERSITET

Validation of HOL4's Formal Floating-Point Model

Hugo Eidmann

Abstract

The interactive proof assistant HOL4 offers an implementation of floating-point arithmetic in formal logic, which facilitates reasoning about the behaviour of floating-point operations. The aim of this thesis was to test this model, using given test vectors, to determine whether it correctly implements the floating-point semantics for the general arithmetic and comparison operations as specified by the IEEE standard for floating-point arithmetic. The results showed that HOL4 provides a correct implementation of all general-computational arithmetic and comparison operations that were tested, with the exception of the square root operation and the two rounding modes roundTowardPositive and roundTowardNegative, which were unable to be validated using the test vectors. The results did not show that their implementation was incorrect, but rather that they lacked the support necessary for evaluating expressions in which they occurred. Although they cannot be properly evaluated, they may still facilitate reasoning about their behaviour in HOL4's formal logic, which is the main purpose of the floating-point model.

Faculty of Science and Technology

Uppsala University, Uppsala

Supervisor: Tjark Weber Subject reader: Justin Pearson

Examiner: Johannes Borgström

Contents

Chapter 1

Introduction

This thesis aimed to determine whether the formal model of floating-point arithmetic implemented in the interactive theorem prover HOL4 correctly implements the IEEE floating-point semantics for the common 32 and 64-bit arithmetic and comparison operations. This was done by using generated test vectors from the TestFloat library [13], which were parsed, fed into HOL, and evaluated. We found that most of the operations were correctly implemented. The only exception was the square root operation and two of the four required rounding modes, roundTowardPositive and roundTowardNegative which were unable to be validated using the test vectors. Additionally, we found that the 32-bit formats were substantially faster to evaluate than the 64-bit formats, although the reason for this could not be conclusively determined.

1.1 Thesis Outline

We begin by introducing the concept of floating-point arithmetic and the challenges that come with its complicated nature. We then provide an overview of the IEEE standard for floating-point arithmetic in chapter 2, which defines the formats, encodings, operations, and rounding modes that are tested in this project. In chapter 3 we describe the proof assistant HOL4 and its formal floating-point model. Chapter 4 details how test vectors were generated and evaluated in HOL4. The results are presented and discussed in chapter 5, followed by a chapter on related work. Finally, we conclude the thesis and discuss future work in chapter 7.

1.2 Introduction to Floating-Point Arithmetic

Representing real numbers has long been a challenge for computers. With limited memory that can only store a finite number of bits, it is impossible to store all real numbers with infinite precision. Therefore, computers have to make trade-offs between accuracy and performance. One way of doing this is by using fixed-point arithmetic, which represents numbers by storing a fixed number of digits after the decimal point [25]. However, this method does not offer a lot of flexibility when dealing with very large or very small numbers. For example, a signed 16-bit fixed-point number that uses 10 bits for the integer part and 6 bits for the fractional part can only represent linearly spaced numbers between -512 and 511.984375 at intervals of 0.015625 (2^{-6}).

A more common way of representing real numbers is by using floating-point numbers, which use a fixed number of bits to represent the significand and the exponent of a number in scientific notation. This allows for more flexibility and precision in representing a wide range of numbers because the position of the decimal point can be adjusted by changing the exponent. A signed 16-bit floating-point number that uses 1 bit for the sign, 5 bits for the exponent, and 10 bits for the significand can represent numbers as small as $5.96 \cdot 10^{-8}$ [1] and as large as 65504. However, this method introduces a new set of challenges, such as rounding errors and loss of precision when performing arithmetic operations. It requires a way of handling results that cannot be represented exactly in the given number of bits. Take the operation $1 \div 5$, for example. Both 1 and 5 can be represented exactly as 16-bit floating-point numbers —0111110000000000 and 0100010100000000 respectively — but the result of the division, 0.2, cannot be represented exactly. This is because 0.2 in binary is a repeating fraction, $0.001100110011\ldots$, much like how $\frac{1}{3}$ cannot be represented exactly in decimal. Therefore, the result of the division has to be rounded to some approximate number that can be represented in the given floating-point format, which in this case is 0.199951171875.

To ensure that the behaviour of floating-point arithmetic is well-defined and consistent across different platforms, the Institute of Electrical and Electronics Engineers (IEEE) has standardised the representation of floating-point numbers in the IEEE 754 standard [16]. This standard defines the format of floating-point numbers, as well as the rules for arithmetic operations, rounding, and exceptions. It has been adopted by most modern computers and programming languages, including Java and C++ [1; 24].

Floating-point arithmetic can be implemented in dedicated hardware accel-

[1] $6.10 \cdot 10^{-5}$ for normalised numbers.

erators, such as the Floating-Point Unit (FPU) in modern CPUs, or in software libraries that emulate the behaviour of floating-point numbers. The complicated nature of floating-point arithmetic implementations gives rise to the need for reasoning about the behaviour of floating-point operations. This is especially important in safety-critical applications, such as avionics, automotive, and medical devices, where incorrect results can have catastrophic consequences. One way of doing so is by using interactive proof assistants.

Interactive proof assistants, such as Coq [22], Isabelle [23], and HOL4 [20], are software tools used in the field of formal verification and theorem proving. They provide a platform for users to express mathematical formulas in a formal language and construct proofs for those formulas in a logical calculus, which can be used to ensure the correctness of software and hardware systems. HOL4 offers a software implementation of floating-point arithmetic in formal logic. This facilitates reasoning about the behaviour of floating-point operations; for instance, to prove that $x+y = y+x$ for all floating-point numbers x and y. This can be useful for proving the correctness of floating-point algorithms, such as numerical solvers, signal processing algorithms, and scientific simulations, as well as the correctness of other algorithms or programs which require a correctly implemented floating-point model. However, it requires a correctly implemented floating-point model to confidently trust the conclusion of that proof. That is why we aim to test HOL4's floating-point model in this project.

Chapter 2

The IEEE Standard for Floating-Point Arithmetic

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard that provides guidelines for floating-point implementations. It specifies decimal and binary formats at different levels of precision, encoding of floating-point values, methods for computing operations and the behaviour of these operations, and the handling of exception conditions. It was first published in 1985 [15] and has since been revised several times. The most recent version, IEEE 754-2019 [16], is used as reference in this thesis.

2.1 Formats

The IEEE standard defines several floating-point formats. A format is specified by a set of integer parameters, namely, a radix b, a precision p, and an exponent range $[emin, emax]$. Together, these parameters determine the widths of the sign, exponent, and significand fields in a floating-point number.

The radix b is the base of the number system. The standard provides definitions for binary formats with $b = 2$ and decimal formats with $b = 10$. This project focuses solely on binary formats since HOL4 does not have an implementation for decimal formats. The precision is the number of digits in the significand. $emax$ and $emin$ are the maximum and minimum values of the exponent, respectively. One constraint for all formats is that $emin = 1 - emax$.

The standard defines three basic binary floating-point formats. binary32, binary64, and binary128. Due to time constraints, this project will only test the two most common formats, binary32 and binary64. See table 2.1 for a summary of the parameters and properties of each format.

Additionally, each format shall represent the following floating-point data:

- Signed zero and non-zero floating-point numbers of the form $(-1)^s \times b^e \times m$, where
  - s is 0 or 1 (sign)
  - e is any integer $emin \leq e \leq emax$
  - m is a number represented by a digit string of the form $d_0 \bullet d_1 d_2 \ldots d_{p-1}$ where d_i is an integer digit $0 \leq d_i < b$
- Two infinities, $+\infty$ and $-\infty$
- Two NaNs, qNaN (quiet) and sNaN (signalling)

2.2 Encodings

The standard also defines the encoding for each format. A floating-point value is encoded in k consecutive bits split into three fields, where $k = 32$ for binary32 and $k = 64$ for binary64.

a) 1-bit sign S

b) w-bit biased exponent

c) a $(t = p-1)$-bit trailing significand field digits string $T = d_1 d_2 \ldots d_{p-1}$; the leading bit of the significand, d_0, is implicitly encoded in the biased exponent E.

				Significand	**Exponent**		
name	**common name**	**k, bits**	**b, radix**	**p, bits**	**w, bits**	**min**	**max**
binary32	Single precision	32	2	24	8	−126	+127
binary64	Double precision	64	2	53	11	−1022	+1023

Table 2.1: Properties of floating-point formats

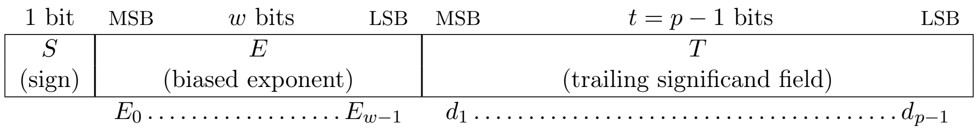


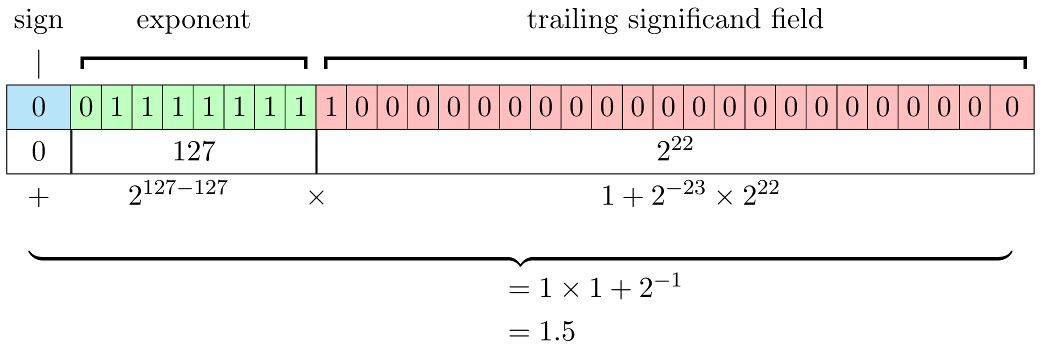
Figure 2.1: Binary interchange floating-point format [16, fig. 3.1]

To avoid storing the exponent as a negative integer but still be able to represent both large and small values, a biased exponent is used. This means that the value of the exponent e is calculated as $e = E + bias$, where $bias = -(b^{w-1} - 1)$, i.e., $e = E - 127$ for binary32 and $e = E - 1023$ for binary64. The value v of a floating-point number is calculated as follows:

$$v = \begin{cases} (-1)^s \times 2^{E-bias} \times (1 + 2^{1-p} \times T) & \text{if } 1 \leq E \leq 2^w - 2 \\ (-1)^s \times 2^{emin} \times (0 + 2^{1-p} \times T) & \text{if } E = 0 \text{ and } T \neq 0 \end{cases}$$

The latter is for *subnormal* (denormalized) numbers, which are numbers with a magnitude smaller than 2^{emin}. Subnormal numbers are represented by setting the biased exponent to 0 and the trailing significand field to a non-zero value. They are distinct from normal numbers because they are intrinsically less precise [16, chap. 2.1].

For example, the decimal number 1.5 is represented as a 32-bit binary floating-point number in the following way:



2.3 Infinity and NaNs

Some bit configurations are reserved for the special values $+\infty$, $-\infty$, qNaN (quiet Not a Number), and sNaN (signalling Not a Number). For both infinity and NaN, the biased exponent E is set to all ones, i.e., $E = 2^w - 1$, and they are differentiated by the trailing significand field (see table 2.2). Infinities have a trailing significand field of all zeroes ($T = 0$) and use the sign bit to indicate the sign of the infinity.

The value NaN is used when the result of an arithmetic operation is undefined, such as $1 \div 0$ or $\sqrt{-1}$. A quiet NaN is propagated from an operation without signalling exceptions (see 2.4) while a signalling NaN will signal exception flags along with the result. NaNs have a trailing significand field $T \neq 0$ and are differentiated by setting the most significant bit (MSB) of the trailing significand field to 1 for qNaN and 0 for sNaN. If the MSB of the trailing significand field is set to 0, at least one other bit must be set to 1, otherwise it would be an infinity. The remaining bits of the trailing significand field can be used to encode a payload containing additional diagnostic information.

Value	Sign	Biased exponent					Trailing significand field				
$+\infty$	0	1	1	1	...	1	0	0	0	...	0
$-\infty$	1	1	1	1	...	1	0	0	0	...	0
qNaN	0 or 1	1	1	1	...	1	1	d_2	d_3	...	d_{p-1}
sNaN	0 or 1	1	1	1	...	1	0	d_2	d_3	...	d_{p-1}

Table 2.2: Encodings for special values

2.4 Exceptions and Flags

This section details the five kinds of exceptions specified and required by the IEEE standard.

In some cases, operations cannot be performed as expected; for example, when dividing by zero or taking the square root of a negative number. In mathematics we may be satisfied with saying that the result is undefined or expressing it in terms of i, but a computer needs a way of handling these cases. In other cases, the infinitely precise result of an operation may not be representable in the given floating-point format, due to the limited number of bits available. Therefore, the IEEE standard defines five exceptions that can be signalled when an operation cannot be performed as expected, as well as the default result to be delivered in such cases [16, chap. 7].

Invalid operation

Invalid operation is an exception that is signalled when an operation is executed with an invalid operand, meaning that there is no usefully definable result. Any arithmetic operation on a signalling NaN will signal this exception. Additionally, there are some cases in which an invalid operation exception is signalled without having a signalling NaN as one of its operands. These are:

- Multiplication of 0 and ∞: $0 \times \infty$ or $\infty \times 0$
- fused multiply-addition containing the multiplication of 0 and ∞: $0 \times \infty + c$ or $\infty \times 0 + c$.

  An exception could be made when the third operand, c, is a quiet NaN. The IEEE standard states that whether to signal an invalid operation exception for such cases is implementation defined.
- Addition, subtraction, or fused multiply-addition containing magnitude subtraction of infinities. For example: $+\infty - (+\infty)$ or $+\infty \times 1 + (-\infty)$
- $\infty \div \infty$ or $0 \div 0$. Division by zero where the dividend is non-zero is not covered by the invalid operation exception. It instead has its own exception (see 2.4).
- Square root of a negative operand, such as $\sqrt{-1}$ or $\sqrt{-\infty}$.

If an operation that signals the invalid operation exception is expected to produce a result in a floating-point format, such as for the operations above, then the result of that operation is a quiet NaN. There are additional cases for when this exception is signalled, such as for conversion and comparison operations, which are not covered in the scope of this project.

Division by zero

The divideByZero exception is signalled for divisions where the dividend is a finite non-zero number and the divisor is 0, such as $1 \div 0$ or $-1 \div 0$. For these cases, the result of the operation is $+\infty$ if both operands have the same sign and $-\infty$ if they have different signs.

This exception is also signalled for logarithmic operations where the operand is 0, such as $log_2(0)$. The result in this case is $-\infty$ regardless of the sign of the operand.

Overflow

Overflow exception is an exception that is signalled when the result of the operation is of greater magnitude than the largest finite number representable in the destination format. The result of an operation that signals an overflow exception is determined by the sign and the rounding mode that is used for the operation.

Underflow

An underflow exception occurs when a non-zero result of an operation is too small to be representable in the destination format. That is, if the result computed with infinite precision would lie strictly between $\pm b^{emin}$, where b is the radix of the destination format and *emin* is the minimum value of the exponent in the destination format. A result of this kind is called a *tiny* result. The IEEE standard states that the detection of *tininess* can be done either before or after rounding. The same rounded result would be delivered for both methods. The only difference in behaviour between the two methods is when a result computed with infinite precision lies strictly between $\pm b^{emin}$ but would be rounded to $+b^{emin}$ or $-b^{emin}$. In that case, only the before-rounding method would signal underflow. HOL4 provides an implementation for both detection methods and uses separate flags for each case.

Inexact

The inexact exception is signalled when the result of an operation differs from what would be the result if it was computed with infinite precision. The difference between the inexact exception and the underflow exception is that the inexact exception does not only apply to tiny numbers. As mentioned in chapter 1, for example, the decimal number 0.2 cannot be precisely represented as a finite floating-point number. Thus, the operation $1 \div 5$ would result in an inexact exception. In these cases, the rounded result is delivered in the destination format.

2.5 Rounding Modes

Rounding modes, formally known as rounding-direction attributes, are used to determine how the result of an operation should be rounded when it cannot be represented exactly in the given floating-point format. Before rounding, the result of an operation is considered infinitely precise and with unbounded range. If the result is not representable in the destination format, it must be rounded to the nearest representable value and, if necessary, signal the inexact, underflow, or overflow exception (see 2.4). The

IEEE standard provides five rounding modes: roundTiesToAway, roundTiesToEven, roundTowardZero, roundTowardPositive, and roundTowardNegative [16, chap. 4.3]. However, roundTiesToAway is not required for a conforming implementation and is therefore not implemented in HOL4.

The default rounding mode for binary formats is roundTiesToEven, which rounds the infinitely precise result r to the nearest representable value. If the result is exactly halfway between two representable values, it is rounded to the value with an even least significant bit. The roundTiesToAway mode rounds the result to the nearest representable value, but if the result is exactly halfway between two representable values, it is rounded to the one with larger magnitude, i.e., negative values round down towards negative infinity and positive values round up towards positive infinity. For both of these modes, if the infinitely precise result is at least $b^{emax} \times (b - \frac{1}{2}b^{1-p})$, it will round to $+\infty$ or $-\infty$ depending on the sign of the result.

The last three rounding modes are called *directed* rounding attributes. They round the result in a specific direction; they do not default to rounding the result to the nearest representable value.

- roundTowardZero rounds the result to the closest representable value that is not greater in magnitude than the infinitely precise result:

  $$|roundTowardZero(r)| \leq |r|$$

- roundTowardPositive rounds the result to the closest representable value that is not less than the infinitely precise result:

  $$roundTowardPositive(r) \geq r$$

  If the infinitely precise result is greater than the largest representable result, it will round to $+\infty$.

- roundTowardNegative rounds the result to the closest representable value that is not greater than the infinitely precise result:

  $$roundTowardNegative(r) \leq r$$

  If the infinitely precise result is greater in magnitude than the largest representable result, it will round to $-\infty$.

Chapter 3

HOL4

HOL4 is a proof assistant used for interactive theorem proving in higher order logic (HOL) [21]. It is built on top of the general-purpose programming language Standard Meta-Language (SML) [12] which acts as an interface for the formal logic. In some sense, HOL4 is a library of SML with its own kernel, syntax, logic and parser implemented in SML. This way of mechanising formal proofs was first introduced by a system called LCF (logic for computable functions) developed by Robin Milner in the 1970's [10]. LCF is the basis on which many of the modern proof assistants are built, including HOL4.

In the 1980's, Mike Gordon at Cambridge University started developing HOL, with its first release in 1988 called HOL88 [9]. The development of HOL continued through the 1990's with releases of HOL90 in 1990 and HOL98 in 1998. HOL4 is the latest version of HOL and is a direct descendant of HOL98. The logic implemented in HOL4 has remained the same since the release of HOL88. It is very similar to Church's Simple Type Theory which was introduced in 1940 [5; 20].

The way you interact with HOL4 is through its interactive interpreter which runs on Poly/ML [18] or Moscow ML [19]. The interpreter is a fully functioning ML interpreter with added support for parsing and pretty-printing HOL expressions [21].

3.1 Datatypes

The three main ML datatypes implemented in HOL are terms (`:term`), theorems (`:thm`), and HOL types (`:hol_type`). The type `term` is practically equivalent to Church's simply typed lambda calculus and it represents the set of HOL terms. In this sense, `term` is an abstract type and may be a variable, a constant, an application of one term to another, or a lambda abstraction. Although terms can be created by calling pure ML functions, a more common way is to write terms inside ` ``double backticks`` `. This is a special kind of syntax which essentially tells the parser to parse the expression written inside it. They can also be written inside `` `single backticks` ``, in which case they are not immediately parsed by the parser. [21]

Terms and HOL types will henceforth be written inside backticks to distinguish them from regular ML. Note that HOL types are prefixed with a colon (':').

The type `hol_type` is used to distinguish between different types of terms. For example, both ` ``14.5`` ` and ` ``sqrt`` ` are of the ML type `term`, but one is a function (application) that has the HOL type (`hol_type`) ` ``:real -> real`` ` and the other is a constant number that has the HOL type ` ``:real`` `. To extract the type of a term one can use the standard library function `type_of`. For example

```
>   type_of ``14.5``;
val it = ``:real``: hol_type
```

One can define new types using the `Datatype` keyword. The syntax is very similar to ML's datatype definition and looks like this:

```
Datatype`ident = specification`
```

Where `ident` is the name (identifier) of the datatype. This is what you see when you invoke `type_of`. `specification` can be a series of clauses, separated by a vertical bar (|), each consisting of an identifier followed by zero or more types. For example, the datatype for float values is defined as follows:

```
Datatype `float_value = Float real | Infinity | NaN`
```

An instance of `` `float_value` `` can be either of these clauses but never more than one at the same time.

The specification could also be a record on the form:

```
<|ident : hol_type; ident : hol_type; ... |>
```

For example, the `` `float` `` datatype is defined as a record:

```
Datatype`float =
  <|Sign : word1; Exponent : 'w word; Significand : 't word|>`
```

Here we see the recursive nature of datatype definitions. The type ``` ``:float`` ``` is partly defined by the `word` datatype, where ``` ``:α word`` ``` denotes a word of α bits, and `'w` and `'t` are the sizes for the exponent and trailing significand field respectively. Since the float type uses type variables (`'w` and `'t`), an instance of a float has the type ``` ``:('t, 'w) float`` ```. For example, the 32-bit floating-point number with decimal value 42 is constructed as follows:

```
``<|Sign := 0w; Exponent:=132w; Significand:=0x280000w|> :(23,8)float``
```

3.1.1 Theorems

The type `:thm` represents a proven theorem consisting of a set of hypotheses and a conclusion, both of type ``` ``:bool`` ```, separated by the turnstile symbol ($\vdash$). For example, the most basic theorem, which has no hypotheses, is

```
⊢ T
```

where `T` denotes the boolean value 'true'. Another theorem could be

```
⊢ ∀x. x + y = y + x
```

One can also define term applications as theorems with the syntax:

```
Definition ident₁: ident₂ χ₁ ... χₙ = t₁ ··· tₘ
End
```

or equivalently

```
val ident₁ = Define `ident₂ χ₁ ... χₙ = t₁ ··· tₘ`;
```

Where $ident_1$ is the name of the ML variable holding the definition, $ident_2$ is the identifier of the term application, $\chi_1 \ldots \chi_n$ are the parameters, and $t_1 \cdots t_m$ are the terms specifying the application. This yields the following theorem:

$$\vdash \ \forall \chi_1 \ \ldots \ \chi_n . \ ident_2 \ \chi_1 \ \ldots \ \chi_n \ \Leftrightarrow t_1 \ \cdots \ t_m$$

For example, let us define the congruency operation. We say that two integers a and b are congruent modulo m if their difference is a multiple of m, where $m \geq 1$. That is,

$$a - b = km, \text{ for some } k \in \mathbb{Z}$$

This is usually denoted as

$$a \equiv b \mod m$$

The following is an example of how this operation can be defined in HOL. Lines starting with '>' and subsequent indented lines are user input, the other lines are output from the HOL interpreter.

```
>   Definition congruent_def:
    congruent a b c = ∃x. a - b = x * c
    End
val congruent_def =
    ⊢ ∀a b c. congruent a b c ⇔ ∃ x. a - b = x * c: thm
```

Now the ML variable `congruent_def` holds the theorem that defines the congruency operation. We can then construct a theorem that uses this definition. For example, we want to prove the following theorem

```
⊢ ∀a. congruent a 0 a
```

Which states that any integer is congruent with 0 modulo itself. We add this to the goalstack by invoking the function `g`:

```
>   g `∀a. congruent a 0 a`;
val it =
    Proof manager status: 1 proof.
    1. Incomplete goalstack:
        Initial goal:
        ∀a. congruent a 0 a
    : proofs
```

The proof manager echoes our goal, i.e., the theorem we aim to prove. To prove the theorem, we can supply the proof manager with a series of *tactics*. A tactic is essentially an ML function that reduces the goal to a list of subgoals. In this case, we expand an invocation of `rw`, which **rew**rites (and simplifies) our goal using our definition theorem.

```
>   expand (rw[congruent_def]);
OK..
1 subgoal:
val it =
    ∃x. x = 1 ∨ a = 0
    : proof
```

We then get a subgoal. This goal looks much simpler than our original goal; we only have to prove that there exists an integer x whose value is equal to 1. This can be proven by simply providing an example of such an integer, thereby proving that it exists. We do that by using `qexists_tac`, which reduces an existentially quantified goal to a goal involving a specific witness. In our case, the witness is the integer 1.

```
>   expand (qexists_tac `1`);
OK..
1 subgoal:
val it =
    1 = 1 ∨ a = 0
    : proof
```

We get a new subgoal which can be simplified using basic rules of inference, such as

$$\varphi \Rightarrow \varphi \vee \psi$$

We do not need to supply any theorems for this simplification because these basic rules are implicit in the definition of `rw`.

```
>   expand (rw[]);
OK..

Goal proved.
⊢ 1 = 1 ∨ a = 0

Goal proved.
⊢ ∃x. x = 1 ∨ a = 0
val it =
    Initial goal proved.
    ⊢ ∀a. congruent a 0 a: proof
```

The proof manager tells us that the subgoal is proven, which in turn means that we have proven our initial theorem.

Theorems can be useful for many reasons. Often proving a theorem involves using already existing theorems. They can also be used as rules for converting and evaluating terms.

3.1.2 Conversions and the EVAL Function

Theorems can also be used as rules for converting and evaluating terms. A conversion, usually indicated by the suffix '`_CONV`', is a function that takes a term `t` as an argument and finds another term `t'` which is equal to `t` and returns a theorem stating that equality. The process of finding such a term often involves applying conversions to the term constants occurring in `t` by using underlying theorems and rules of inference. For example, the conversion `ADD_CONV` calculates the sum of two numerals. Applying `ADD_CONV` to the term ``` ``5+5`` ``` returns $\vdash 5{+}5 = 10$, i.e., a theorem stating that ``` ``5+5`` ``` is equal to the term ``` ``10`` ```. However, applying a conversion to a term does not necessarily guarantee that it will find a desired equal term. It might, for example, lack the theorems needed to rewrite one of the term constants occurring in `t`. In those cases, a conversion could return $\vdash$ `t = t`.

One important conversion is the `EVAL` function which evaluates a term by deduction. It does so by rewriting the term using a set of simplification rules (theorems) and applying the defining equations for each constant occurring in its argument. These equations are automatically added to a mutable datastructure when a theorem is defined using the `Define` keyword (see 3.1.1). Although `EVAL` can be applied to any term, in this section we will focus on how floating-point values and operations are evaluated using `EVAL`.

The real ($\mathbb{R}$) value v of a float ``` ``f`` ``` is obtained by applying `EVAL` to a term on the form ``` ``float_value f`` ```. To do this, it uses a theorem that defines the ``` ``float_value`` ``` operation which includes the logic for evaluating the floating-point value described in sections 2.2 and 2.3. This will return a theorem on the form

$$\vdash \texttt{float\_value f} = \texttt{f'}$$

where `f'` is either `Infinity`, `NaN`, or `Float` v.

All of the arithmetic operations use ``` ``float_value`` ``` for evaluation. The general method for evaluating arithmetic operations is that the real number result of an operation is computed by using the `EVAL` on the real values of its operands. From this, we get the expected result by rounding and converting it to a float. The expected result then acts as a witness when proving that the witness floating-point result is correct.

For example, say we want to evaluate the term

```
``float_add roundTiesToEven ^f3 ^f4``
```

using `EVAL`, where `f3` and `f4` are the following variables which hold the floating-point numbers with real values 3 and 4 respectively:

```
val f3 = ``<|Sign := 0w; Exponent := 128w; Significand := 0x400000w|>``
val f4 = ``<|Sign := 0w; Exponent := 129w; Significand := 0w|>``
```

After a series of conversions, we get the term

```
``float_round_with_flags roundTiesToEven F (3+4)``
```

Here we can see that the two operands have been converted into numerals (of type ``` ``:real`` ```). After another series of conversions, disregarding the flags part, this will then be converted to ``` ``round roundTiesToEven 7`` ```. The real value result has been computed outside of the floating-point logic and the next step is to round it to a float again to get the witness

```
``<|Sign := 0w; Exponent := 129w; Significand := 0x600000w|>``
```

The witness is then plugged into the theorem `round_roundTiesToEven`:

```
⊢ ∀yxr.
    float_value y = Float r ∧
    (y.Significand = 0w ∧ y.Exponent ≠ 1w ⇒ abs r ≤ abs x) ∧
    2 * abs (r - x) ≤ ULP (y.Exponent,(:τ)) ∧
    (2 * abs (r - x) = ULP (y.Exponent,(:τ)) ⇒
    ¬ word_lsb y.Significand) ∧ ulp (:τ # χ) < 2 * abs x ∧
    abs x < threshold (:τ # χ) ⇒
    round roundTiesToEven x = y
```

Which states the conditions for a witness x being correct, based on its proximity to the real number result r. In this case, all the conditions are true, so we get

```
⊢ T ⇒
    round roundTiesToEven 7 =
    <|Sign := 0w; Exponent := 129w; Significand := 0x600000w|>
```

The final result from the evaluation is a tuple consisting of the exception flags and the floating-point result:

```
⊢ float_add roundTiesToEven
    <|Sign := 0w; Exponent := 128w; Significand := 0x400000w|>
    <|Sign := 0w; Exponent := 129w; Significand := 0w|> =
    (<|DivideByZero := F; InvalidOp := F; Overflow := F;
    Precision := F; Underflow_BeforeRounding := F;
    Underflow_AfterRounding := F|>,
    <|Sign := 0w; Exponent := 129w; Significand := 0x600000w|>)
```

Chapter 4

Implementation

This chapter provides a detailed description of how the test vectors are generated, processed, parsed and evaluated.

The main part of the implementation is divided into two SML files; FP_test_parser.sml contains the logic for parsing the tests and FP_test_base.sml contains the logic for evaluating tests, as well as some general utility functions. The base of the code, including preprocessing, was created by Peter Lammich[1] who originally developed it for testing the formal floating-point model for Isabelle [26]. The complete implementation, along with the TestFloat library and generated test vectors are publicly available on github[2].

4.1 Berkeley TestFloat Library

Test vectors were generated by the Berkeley TestFloat library version 3e, released in 2018 [13]. This library provides a collection of programs for testing binary floating-point arithmetic. The library supports testing all operations required by the original 1985 version of the IEEE 754 Standard [16], except for conversions to and from decimal which are not implemented in HOL4. The 2019 version of IEEE 754 was used as reference in this project. Therefore, all discrepancies between the versions have been considered when comparing the results.

The TestFloat library generates a large number of test vectors designed to compare the behaviour of an implementation of floating-point arithmetic to

[1]https://www21.in.tum.de/~lammich/
[2]https://github.com/hugeid/HOL_FP_testing

its internal reference implementation. The output format can vary slightly depending on the operation that is tested and the supplied arguments to the program. For this project, the output format for a test vector is a single line of text consisting of the operands and result as hexadecimal numbers (in little-endian order) and the expected exception flags encoded in hexadecimal with one bit per flag (see table 4.3 for reference). The following is an example of a test vector for 32-bit binary floating-point addition.

$$\begin{array}{ccc} \overbrace{\texttt{0x8683F7FF 0xC07F3FFF}}^{\text{operands}} & \overset{\text{expected result}}{\texttt{0xC07F3FFF}} & \overset{\text{exception flags}}{\texttt{01}} \end{array}$$

Figure 4.1: Example of a TestFloat test vector for 32-bit binary floating-point addition

4.2 Preprocessing

Before a test vector, such as that shown in figure 4.1 is parsed, some additional information is added. Originally, each test vector consists of the operands, result, and exception flags, but to run the test we also need to know the

a) source format

b) operation

c) rounding mode

Firstly, a field containing the format followed by the operation is added. The first three characters of this field denote the format and the rest denotes the operation. Source and destination format is the same for all arithmetic operations but can differ for other operations, such as conversions to and from integer formats [16, chap. 5.8]. The destination format for comparison operations is always a boolean. Table 4.1 shows a full list of tested operations along with their corresponding names in HOL4 and IEEE 754.

Secondly, a field containing the rounding mode is added. Rounding modes are encoding according to table 4.2.

The addition of the format, operation, and rounding mode fields is accomplished by using the shell script shown in appendix 7.1. It generates test vectors using the TestFloat library and then adds the relevant fields to each line.

Thirdly, the TestFloat library can only generate cases for one of the underflow detection methods at a time but HOL4 supports both detection methods simultaneously. Therefore, two instances of test case for each operation and format were generated and saved in separate files, one that used the "after rounding" detection method and one that used the "before rounding" detection method. The two files were then combined into one which had a separates flags for the two methods. This was done by the Python script shown in appendix 7.1. See table 4.3 for the exception flags encoding.

The following is an example of a test vector after preprocessing.

operation	rounding mode	operands	result	exception flags
`b32+`	`0`	$\overbrace{\texttt{0x8683F7FF 0xC07F3FFF}}$	`0xC07F3FFF`	`01`

Figure 4.2: Example of a test vector for 32-bit binary addition with rounding mode roundTowardZero, operands -4.96411×10^{-35} and -3.98828, expected result -3.98828, and expected exception flag inexact

4.3 Parsing

For this project, parsing consists of

1. splitting each test vector into its components, i.e., operation, rounding mode, operands, expected result, and flags.

Format & Operation	HOL4 Reference	IEEE 754 Reference
`b32+`, `b64+`	`float_add`	*formatOf*-**addition**(*source1*, *source2*)
`b32-`, `b64-`	`float_sub`	*formatOf*-**subtraction**(*source1*, *source2*)
`b32*`, `b64*`	`float_mul`	*formatOf*-**multiplication**(*source1*, *source2*)
`b32/`, `b64/`	`float_div`	*formatOf*-**division**(*source1*, *source2*)
`b32V`, `b64V`	`float_sqrt`	*formatOf*-**squareRoot**(*source1*)
`b32*+`, `b64*+`	`float_mul_add`	*formatOf*-**fusedMultiplyAdd**(*source1*, *source2*, *source3*)
`b32eq`, `b64eq`	`float_equal`	*boolean* **compareQuietEqual**(*source1*, *source2*)
`b32lt`, `b64lt`	`float_less_than`	*boolean* **compareQuietLess**(*source1*, *source2*)
`b32le`, `b64le`	`float_less_equal`	*boolean* **compareQuietLessEqual**(*source1*, *source2*)

Table 4.1: Tested operations and their corresponding names in HOL4 and IEEE 754. *formatOf* is either binary32 or binary64.

2. translating each component from its encoding to a SML/HOL object.

3. mapping each operation to its evaluation function.

4.3.1 Operands

The operands for all tests in this project were floating-point numbers encoded in hexadecimal. When converted to binary, each of these hexadecimal numbers contains the sign, biased exponent, and trailing significand field as a continuous sequence of bits, in the order shown in figure 2.1. Conversion from a hexadecimal string to a HOL float was done in three steps.

1. First, the hexadecimal string is converted into its corresponding binary string of fixed length, either 32 or 64 depending on the format being tested, where the first character is the most significant bit. Thus, the binary number 1, for example, would have 31 or 63 leading zeros.

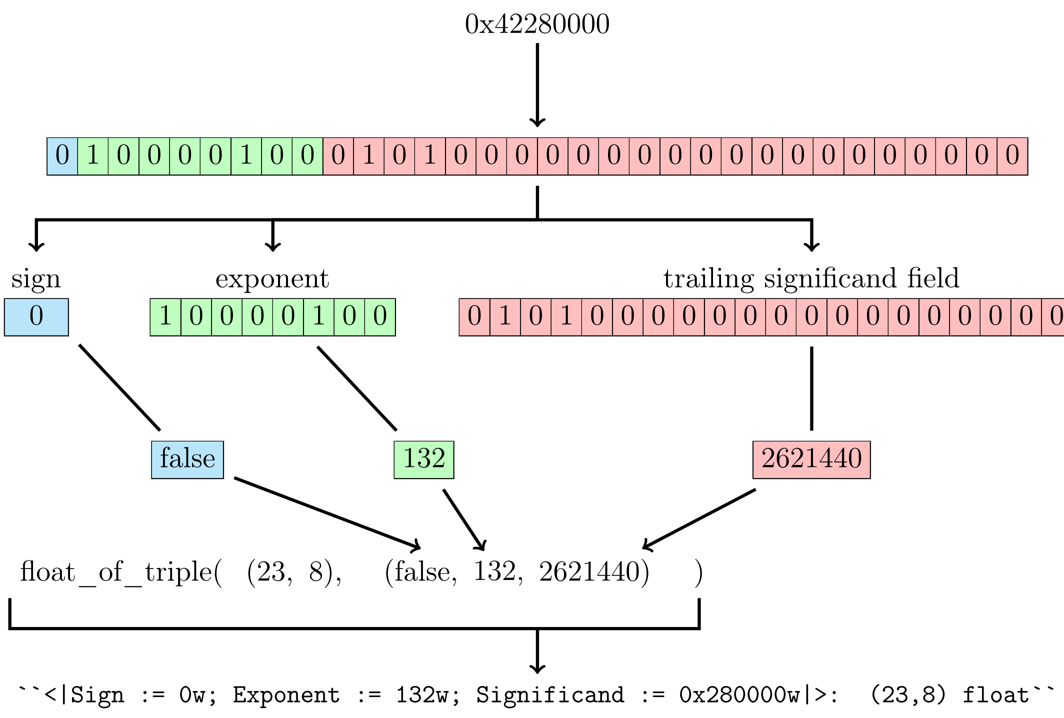
2. Second, the string is split up into the three substrings, one for each component, of lengths 1, $p-1$, and w (see table 2.1).

3. Third, the sign string is converted to the type `bool` and the other two strings are are converted to the integer-like type `num` and passed as arguments to the float constructor function `float_of_triple` along with the bit-sizes for the exponent and significand, which returns the corresponding floating-point number as the HOL type ``` ``:(p,w) float`` ```.

Below is an example of how the hexadecimal string "0x42280000" is parsed and converted into the 32-bit binary floating-point number 42.

Rounding mode	HOL4 & IEEE 754 Reference
=0	`roundTiesToEven`
0	`roundTowardZero`
>	`roundTowardPositive`
<	`roundTowardNegative`

Table 4.2: Tested rounding modes and their corresponding names in HOL4 and IEEE 754



4.3.2 Expected result

The expected results for the arithmetic operations is a floating-point number encoded in hexadecimal and the process of converting it into a float in HOL is the same as for the operands.

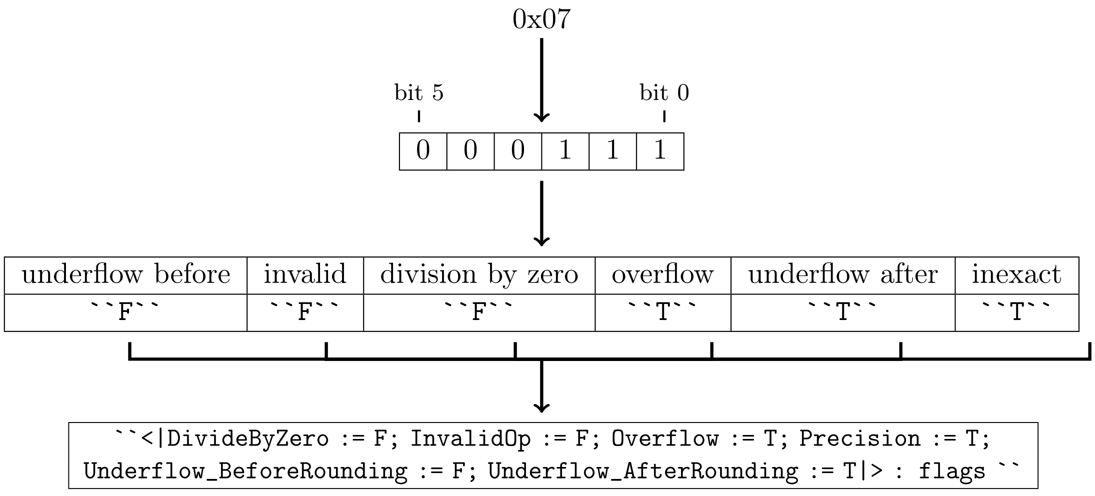
For comparison operations the expected result is a boolean value encoded in hexadecimal as either 0x0 (false) or 0x1 (true). The conversion is simply a mapping of the hexadecimal values to the HOL boolean terms. It should be noted that these terms refer to the HOL-type ``:bool``, which can be either ``F`` or ``T``, and not the SML type `bool`, which can be either `true` or `false`.

bit 0	*inexact* exception
bit 1	*underflow after rounding* exception
bit 2	*overflow* exception
bit 3	*infinite* exception (“divide by zero”)
bit 4	*invalid* exception
bit 5	*underflow before rounding* exception

Table 4.3: Encoding for exception flags in test vectors

4.3.3 Flags

The exception flags are also encoded in hexadecimal. The hexadecimal string is first converted into a list of its corresponding binary digits. Subsequently, each bit is mapped to its boolean counterpart which are then used to create a ``:flags`` object. Below is a diagram showing this process[3].



```
``<|DivideByZero := F; InvalidOp := F; Overflow := T; Precision := T;
Underflow_BeforeRounding := F; Underflow_AfterRounding := T|> : flags ``
```

4.3.4 Rounding Mode

Similarly to the boolean results, rounding modes are parsed by simply mapping their encoded values to the corresponding HOL terms of type ``:rounding`` (see table 4.2).

4.3.5 Operation

The operation is the last to be parsed. Each operation is mapped to its corresponding constructor function, for example `mk_float_add` or `mk_float_equal`, which takes the rounding mode and the operands as arguments and returns a term of type ``: flags # ($p$, $w$) float`` for arithmetic operations or ``:bool`` for comparison operations, where p is the number of bits in the significand and w is the number of bits in the exponent. These constructors are not format specific so there is only one function for each operation which constructs both 32- and 64-bit operations (and any other format).

4.4 Evaluation

The `EVAL` function is used to obtain the result and exception flags of an operation. This will return a theorem on the form

[3]The inexact exception flag is labelled `precision` in the HOL4 implementation

```
``⊢ operation = (flags, result)``.
```

from which the flags and result are extracted and evaluated separately.

The result and the expected test result for arithmetic operations are not compared directly. This is because HOL's internal representation of infinity and NaN differ slightly from what is generated in the test vectors. Instead, both values are converted to a ``` ``:float_value`` ``` before they are compared.

The actual test evaluation is done by using the `aconv` function which takes two terms as arguments and tests if they are alpha-convertible, returning a boolean. Two terms are alpha-convertible if they are syntactically equal and differ only in the names of the variables to which they are bound [8]. The main reasons for using `aconv` is that it works for any two terms and that it avoids using something like ``` ``float_equal`` ``` which is one of the operations being tested. The final result of a test is the logical AND of comparing the expected result with the obtained result and expected flags with the obtained flags for equality.

Chapter 5

Results

This chapter presents and explains the results of the tests. A complete table of results is shown in table A.1. It is important to note that a failed test for a particular operation or rounding mode does not necessarily prove that the formal floating-point model implemented in HOL4 is nonconforming to the IEEE standard with respect to that operation or rounding mode. It could, for example, be because the TestFloat library assumes certain properties of the implementation which according to the IEEE standard are implementation defined. It could also be because the internal representation of certain values or operations differ from how they are represented in the test vectors. This is discussed in greater detail in the individual sections for the operators that had failed tests.

As can be seen in A.1, the two rounding modes roundTowardPositive and roundTowardNegative were only tested on 32-bit binary addition, for which almost all tests failed. The reason for this is that they were found to lack the rounding theorems necessary for evaluating operations to a rounded floating-point number. The only tests that passed were those which rounded to a special value (NaN or $\pm\infty$). For this reason, all subsequent arithmetic operations were only tested with the remaining two rounding modes, roundTiesToEven and roundTowardZero. Excluding these two rounding modes, the only operations for which there were failing tests were square root and fused multiply-add. All tests for the comparison operators passed.

Although these rounding modes were unable to be validated by the tests because they could not be evaluated properly, their behaviour was clearly defined in the definition of ``` ``round`` ``` (see 7.1). It is therefore possible that the formal model is conforming to the IEEE standard with respect to these two rounding modes and that you can still use them to reason about their behaviour. However, to validate the current implementation, one has to use some other method that does not include evaluating the operations with `EVAL`.

5.1 Square Root

For the square root operator, about 50% of all tests failed. The test results showed that only the tests where the expected result was a whole number, NaN, or $+\infty$ would evaluate to the expected result. In cases where the expected result was a fraction or $-\infty$, the tests always failed. There are two reasons for why these tests failed.

Firstly, as explained in section 3.1.2, the evaluation of arithmetic operations involves computing the real number result outside the floating-point logic before rounding it to a floating-point result. This means that the evaluation of the floating-point operation relies on the evaluation of the equivalent operation for the type ``` ``:real`` ```. In the case of square root, the current implementation of ``` ``sqrt:real -> real`` ``` will not yield a real result when evaluated if the operand is not the square of an integer, i.e., it cannot be evaluated to a whole number. Specifically, the conversion cannot find an equal term other than the original term and therefore returns the unchanged original term. For example,

```
EVAL ``sqrt 4``
```

returns

```
⊢ sqrt 4 = 2
```

because $\sqrt{4} \in \mathbb{N}$, but

```
EVAL ``sqrt 5``
```

returns

```
⊢ sqrt 5 = sqrt 5
```

because $\sqrt{5} \notin \mathbb{N}$.

Furthermore, since the real number result cannot be computed, neither can the rounded floating-point result nor the exception flags. Thus, these tests fail. Computing the square root of the special values does not rely on computing a real number result and therefore avoids this issue.

Secondly, square root operations involving $-\infty$ could not be evaluated because the definition of ``` ``float_sqrt`` ``` (see appendix 7.1) requires an equality comparison of $-\infty$ and -0 (``` ``NEG0`` ```). This however could not be evaluated to a boolean and instead returns the original term, i.e.,

```
EVAL `` float_minus_infinity (:23 # 8) =
    <|Sign := 1w; Exponent := 0w; Significand := 0w|>``
```

returns

```
⊢ float_minus_infinity (:23 # 8) =
    <|Sign := 1w; Exponent := 0w; Significand := 0w|> ⇔
    float_minus_infinity (:23 # 8) =
    <|Sign := 1w; Exponent := 0w; Significand := 0w|>
```

Note that this refers to the general equality operator ``` ``=`` ``` and not the floating-point operator ``` ``float_equal`` ```, which is different and passed all tests.

Similarly to the rounding modes, this does not necessarily mean that the formal model is nonconforming to the IEEE standard with respect to the square root operator. It only means that we were unable to validate it using the tests provided by the TestFloat library.

5.2 Fused Multiply-Add

For the fused multiply-add operation, only 144 of the total 24532992 tests failed. Common for all of these test vectors were that the first two operands were ± 0 and $\pm\infty$ (in any order), the last operand was a quiet NaN, the expected result was quiet NaN, and it expected the flags for invalid operation to be raised. The result it got was a quiet NaN, as expected, but it did not raise the invalid operation flag. These were also the only tests that failed because of incorrect exception flags but with a correct result.

As explained in section 2.4, whether to signal the invalid operation exception in these cases is implementation defined, according to the IEEE standard [16], meaning that it is optional to include. TestFloat, however, does not allow you to specify whether to signal this exception when the last operand is a quiet NaN which is why the tests failed. Therefore, these failed tests should be regarded as successful as they produced the correct result without any exception flags.

5.3 Performance

This section presents and compares the run time of the tests for the different operations and formats. A complete table of performance results can shown in table A.2. It should be noted that not much emphasis should be put on the specific values for the run times and they should not be regarded as consistent across different machines or accurately reproducible. There are numerous factors that have not been considered in this analysis, such as hardware limitations, varying CPU and memory utilisation by unrelated processes, system timer accuracy, and varying overhead from the parser. The purpose of this section is merely to highlight the general differences between the different operations and formats.

The results show that a 64-bit test took substantially longer on average than a 32-bit, by multiples ranging from 4 to 36 for the different operations. For most of the operations, a 64-bit test took on average about four to ten times as long as its 32-bit counterpart. The outlier was the divide operator, which took on average almost 36 times as long for the 64-bit tests as it did for the 32-bit tests.

The comparison operations were the fastest with on average 11 milliseconds (ms) per 32-bit test and 53 ms per 32-bit test. Out of the arithmetic operations, square root was clearly faster than the others. This could be because many of the tests had expressions that did not evaluate fully and therefore failed and returned earlier than those that evaluated fully. The fact that they had only one operand could also contribute to the shorter time spent per test.

Why the 64-bit operations took so much longer than the 32-bit operations is not entirely clear. It is expected that more bits would take longer to compute, but not by the factor that was observed, assuming that there is an approximately linear relationship between the number of bits and the time it takes to compute. The process of evaluating expressions involves tens, possibly hundreds, of conversions before arriving at the final result. It is likely that the extra time added by the additional bits accumulates over the course of the evaluation and it is not due to a single step. It is also possible that it is due to the implementations of the auxiliary theorems used in the evaluation, such as the theorems for evaluating the real number result of the operation.

Chapter 6

Related Work

Any published work related to verifying floating-point models for proof assistants could not be found. However, there have been numerous works on validating floating-point models in other contexts. For example, a RISC-V emulator for AMD64 hardware [11] and a floating-point multiplier for field-programmable gate arrays [27] both used the Berkeley TestFloat library for testing the floating-point arithmetic.

Becker et al. [4] presents a modular tool, called FloVer, for verifying the error bounds of finite-precision computations. Flover is both implemented and proven correct in HOL4 and Coq.

Abbasi et al. [1] reports on providing floating-point support for the deductive verifier KeY[3], an interactive theorem prover used for verifying Java programs. Similarly to HOL4, this floating-point implementation allows for reasoning and verification of Java floating-point programs.

Fumex, Marché, and Moy [7] introduce an automated approach to verifying floating-point programs with an implementation in the program verifier Why3 and its front-end SPARK. They combine multiple theorem provers and leverage the native support for floating-point arithmetic in the automated solver SMT-LIB.

There also exists a number of different tools for testing floating-point arithmetic with test vectors.

One of the earliest tools, preceding the official adoption of the IEEE 754 standard, was developed by J.T. Coonen in 1984 [6]. It included a large set of format-independent test vectors and a driver program for executing the tests. The so-called floating-point benchmark Paranoia [17] is another tool, introduced in 1985, which tests general aspects of a floating-point implementation such as the range, precision, and behaviour of rounding, overflow, and underflow. Later followed the UCBTEST suite [14], which includes Paranoia as well as other programs for testing difficult edge cases of arithmetic operations.

A more modern alternative to Berkely TestFloat is FPgen [2], a test generation framework for verifying the correctness of floating-point models. Similarly to the Berkeley Testfloat Library, FPgen generates test vectors aimed at testing different aspects of the behaviour of floating-point operations. Moreover, it also allows for verifying the correctness of the intermediate result, i.e., the infinitely precise result before it is rounded to a floating-point value. It was developed in 2002 at the IBM Haifa Research Labs.

Chapter 7

Conclusion

In this thesis, we have shown that HOL4's formal floating-point model provides a conforming implementation of all of the general computational arithmetic operations specified in the IEEE 754 standard with regard to the two rounding modes roundTiesToEven and roundTowardZero, except for the square root operation. Although the square root operation and the remaining required rounding modes, roundTowardPositive and roundTowardNegative, could not be validated, neither could their implementation's nonconformity to the IEEE standard. Instead, we have shown that HOL4's implementation does not provide the necessary theorems needed to fully evaluate operations in which they occur. However, the definitions of their behaviour could still be correct.

Additionally, we have shown that the HOL4 provides a conforming implementation of the quiet comparison operations compareQuietEqual, compareQuietLess, and compareQuietLessEqual.

7.1 Future Work

Although the arithmetic and comparison operations are vital to a complete and correctly implemented floating-point model, there are many more operations and properties that are specified in the IEEE standard which have not been validated in this report. To conclusively determine whether the entirety of HOL4's model provides the required operations and conforms to the IEEE standard, one needs to show that it provides a correct implementation that satisfies each of the specified requirements. Much of this could be done in a similar way, using test vectors, but one could run into the same problem of incomplete evaluation if the necessary theorems are not implemented. In that case, one could explore other methods of validation that do not strictly involve term evaluation.

Bibliography

[1] R. Abbasi, J. Schiffl, E. Darulova, M. Ulbrich, and W. Ahrendt. Combining rule-and SMT-based reasoning for verifying floating-point Java programs in KeY. *International Journal on Software Tools for Technology Transfer*, 25(2):185–204, 2023. doi:10.1007/s10009-022-00691-x.

[2] M. Aharoni, S. Asaf, L. Fournier, A. Koifman, and R. Nagel. FPgen - a test generation framework for datapath floating-point verification. In *Eighth IEEE International High-Level Design Validation and Test Workshop*, pages 17–22, 2003. doi:10.1109/HLDVT.2003.1252469.

[3] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich. *Deductive Software Verification - The KeY Book - From Theory to Practice. LNCS, vol. 10001.* Springer, Cham, 2016. doi:10.1007/978-3-319-49812-6.

[4] H. Becker, N. Zyuzin, R. Monat, E. Darulova, M. O. Myreen, and A. Fox. A Verified Certificate Checker for Finite-Precision Error Bounds in Coq and HOL4. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10, 2018. doi:10.23919/FMCAD.2018.8603019.

[5] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940. doi:10.2307/2266170.

[6] J. T. Coonen. *Contributions to a proposed standard for binary floating-point arithmetic (computer arithmetic).* University of California, Berkeley, 1984.

[7] C. Fumex, C. Marché, and Y. Moy. Automating the Verification of Floating-Point Programs. In A. Paskevich and T. Wies, editors, *Verified Software. Theories, Tools, and Experiments*, pages 102–119, Cham, 2017. Springer International Publishing. ISBN 978-3-319-72308-2.

[8] A. D. Gordon and T. Melham. Five axioms of alpha-conversion. In G. Goos, J. Hartmanis, J. van Leeuwen, J. von Wright, J. Grundy, and

J. Harrison, editors, *Theorem Proving in Higher Order Logics*, pages 173–190, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-70641-0. doi:10.1007/BFb0105404.

[9] M. Gordon. Introduction To The Hol System. In *1991 International Workshop on the HOL Theorem Proving System and Its Applications*, pages 2–3, 1991. doi:10.1109/HOL.1991.596265.

[10] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF A Mechanized Logic of Computation* . Lecture Notes in Computer Science, 78. Springer Berlin Heidelberg, Berlin, Heidelberg, 1st ed. 1979. edition, 1979. ISBN 3-540-38526-6. doi:10.1007/3-540-09724-4.

[11] X. Guo. Dynamic Binary Translator for RISC-V. 2018.

[12] R. Harper, D. MacQueen, and R. Milner. *Standard ml.* Department of Computer Science, University of Edinburgh, 1986.

[13] J. R. Hauser. Berkeley TestFloat Release 3e: General Documentation, Jan 2018. URL `http://www.jhauser.us/arithmetic/TestFloat-3/doc/TestFloat-general.html`. Accessed: 2024-05-15.

[14] D. Hough et al. UCBTEST, a suite of programs for testing certain difficult cases of IEEE 754 floating-point arithmetic. *Restricted public domain software from http://netlib.bell-labs.com/netlib/fp/index.html*, 1988.

[15] IEEE. IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985*, pages 1–20, 1985. doi:10.1109/IEEESTD.1985.82928.

[16] IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. doi:10.1109/IEEESTD.2019.8766229.

[17] R. Karpinski. PARANOIA: a floating-point benchmark. *Byte Magazine*, 10(2):223–235, 1985.

[18] D. Matthews. Poly/ML, Oct 2023. URL `https://www.polyml.org/`. Accessed: 2024-05-21.

[19] Moscow ML. Moscow ML. URL `https://mosml.org/`. Accessed: 2024-05-21.

[20] K. Slind and M. Norrish. A Brief Overview of HOL4. In *Theorem Proving in Higher Order Logics*, pages 28–32, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 9783540710653. doi:10.1007/978-3-540-71067-7_6.

[21] K. Slind and M. Norrish. The HOL System DESCRIPTION, February 2021.

[22] The Coq Development Team. The Coq Proof Assistant. Jan 2019. doi:10.5281/zenodo.2554024.

[23] M. Wenzel, L. C. Paulson, and T. Nipkow. The Isabelle Framework. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 33–38, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-71067-7. doi:10.1007/978-3-540-71067-7_7.

[24] T. Whitney, K. Sharkey, and N. Schonning. IEEE Floating-Point Representation, Aug 2021. URL `https://learn.microsoft.com/en-us/cpp/build/ieee-floating-point-representation?view=msvc-170`.

[25] D. Williamson. Dynamically scaled fixed point arithmetic. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing Conference Proceedings*, pages 315–318 vol.1, 1991. doi:10.1109/PACRIM.1991.160742.

[26] L. Yu. A Formal Model of IEEE Floating Point Arithmetic. *Archive of Formal Proofs*, Jul 2013. ISSN 2150-914x. `https://isa-afp.org/entries/IEEE_Floating_Point.html`, Formal proof development.

[27] H. Zhang, D. Chen, and S.-B. Ko. Area- and power-efficient iterative single/double-precision merged floating-point multiplier on FPGA. *IET Computers & Digital Techniques*, 11(4):149–158, 2017. doi:10.1049/iet-cdt.2016.0100.

Appendices

Appendix A

Results

A.1 Test Results

Operation	**Format**	**Rounding Mode**	**Passed**	**Failed**	**Total**	
add	`b32`	0	46464	0	46464	**185856**
		$= 0$	46464	0	46464	
		$>$	3981	42483	46464	
		$<$	3981	42483	46464	
	`b64`	0	46464	0	46464	**92928**
		$= 0$	46464	0	46464	
subtract	`b32`	0	46464	0	46464	**92928**
		$= 0$	46464	0	46464	
	`b64`	0	46464	0	46464	**92928**
		$= 0$	46464	0	46464	
multiply	`b32`	0	46464	0	46464	**92928**
		$= 0$	46464	0	46464	
	`b64`	0	46464	0	46464	**92928**
		$= 0$	46464	0	46464	
divide	`b32`	0	46464	0	46464	**92928**
		$= 0$	46464	0	46464	
	`b64`	0	46464	0	46464	**92928**
		$= 0$	46464	0	46464	
square root	`b32`	0	5319	4681	10000	**20000**
		$= 0$	5319	4681	10000	
	`b64`	0	5300	4700	10000	**20000**
		$= 0$	5300	4700	10000	

Operation	**Format**	**Rounding Mode**	**Passed**	**Failed**	**Total**	
fused multiply-add	`b32`	0	6133209	39	6133248	**12266496**
		= 0	6133209	39	6133248	
	`b64`	0	6133215	33	6133248	**12266496**
		= 0	6133215	33	6133248	
equal	`b32`		46464	0		**46464**
	`b64`		46464	0		**46464**
less than	`b32`		46464	0		**46464**
	`b64`		46464	0		**46464**
less than or equal	`b32`		46464	0		**46464**
	`b64`		46464	0		**46464**

Table A.1: Test results

A.2 Performance

Operation	**Format**	**Tests**	**Average** (ms)	**Total** (Dd HH:MM:SS)
add	b32	185856	94	04:50:26
	b64	92928	687	17:43:22
subtract	b32	92928	53	01:22:06
	b64	92928	603	15:33:16
multiply	b32	92928	149	03:50:23
	b64	92928	725	18:42:43
divide	b32	92928	107	02:45:08
	b64	92928	3847	4d 03:18:00
square root	b32	20000	11	00:03:42
	b64	20000	93	00:30:57
fused multiply-add	b32	92928	99	14d 00:10:48
	b64	92928	550	78d 00:56:14
equal	b32	46464	11	00:08:49
	b64	46464	55	00:42:58
less than	b32	46464	11	00:08:32
	b64	46464	53	00:40:49
less than or equal	b32	46464	12	00:09:04
	b64	46464	51	00:39:32

Table A.2: Run times for the different operations and formats

Appendix B

HOL Theorems

B.1 ``float_sqrt`` Definition

```
⊢ ∀mode x.
    float_sqrt mode x =
    if x.Sign = 0w then
      case float_value x of
          Float r => float_round_with_flags mode F (sqrt r)
        | Infinity => (clear_flags, float_plus_infinity (:τ # χ))
        | NaN => (check_for_signalling [x], float_some_qnan (FP_Sqrt mode x))
      else if x = NEG0 then (clear_flags,NEG0)
      else (invalidop_flags,float_some_qnan (FP_Sqrt mode x))
```

B.2 The ``round`` Definition

```
⊢ ∀mode  x.
  round mode x =
  case mode  of
    roundTiesToEven =>
      (let
        t = threshold (:τ # χ)
      in
        if x ≤ -t then float_minus_infinity (:τ # χ)
        else if x ≥ t then float_plus_infinity (:τ # χ)
        else
          closest_such (λa. ¬word_lsb a.Significand) float_is_finite x)
  | roundTowardPositive =>
    (let
       t = largest (:τ # χ)
     in
       if x < -t then float_bottom (:τ # χ)
       else if x > t then float_plus_infinity (:τ # χ)
       else closest {a | float_is_finite a ∧ float_to_real a ≥ x} x)
  | roundTowardNegative =>
    (let
       t = largest (:τ # χ)
     in
       if x < -t then float_minus_infinity (:τ # χ)
       else if x > t then float_top (:τ # χ)
       else closest {a | float_is_finite a ∧ float_to_real a ≤ x} x)
  | roundTowardZero =>
    (let
       t = largest (:τ # χ)
     in
       if x < -t then float_bottom (:τ # χ)
       else if x > t then float_top (:τ # χ)
       else
        closest {a | float_is_finite a ∧ abs (float_to_real a) ≤ abs x}
          x)
```

Appendix C

Shell script for generating and processing test vectors

```
#!/bin/bash

set -e
# example: ./mktests_flags.sh -o + -f 64 -t before > tests_add64_before.txt

function error() {
  echo "Error: " $@
  exit 1
}

OPS=('+' '-' '*' '/' '*+')
FORMATS=(32 64)
RMODES=('0' '=0')

NUM_TESTS=""
TININESS="before"
while getopts o:f:r:n:t: flag
do
    case "${flag}" in
        o) OPS=(${OPTARG});;
        f) FORMATS=(${OPTARG});;
        r) RMODES=(${OPTARG});;
        n) N=${OPTARG};;
        t) TININESS=${OPTARG};;
    esac
done

if [ ${N+x} ]; then
  NUM_TESTS="-n $N";
fi
TESTFLOAT=./TestFloat-3e/build/Linux-x86_64-GCC

# Berkeley TestFloat

TFGEN=$TESTFLOAT/testfloat_gen

function xrm() {
  if   [[ $1 == "=0" ]]; then  echo "near_even";
  elif [[ $1 == "0" ]]; then  echo "minMag";
  elif [[ $1 == ">" ]]; then  echo "max"
  elif [[ $1 == "<" ]]; then  echo "min"
  else error "Unknown rounding mode: '$1'"
  fi
}
```

```
function xop() {
  if   [[ $1 == "+" ]]; then echo "add"
  elif [[ $1 == "-" ]]; then  echo "sub"
  elif [[ $1 == "mul" ]]; then  echo "mul"
  elif [[ $1 == "/" ]]; then  echo "div"
  elif [[ $1 == "V" ]]; then  echo "sqrt"
  elif [[ $1 == "*+" ]]; then  echo "mulAdd"
  elif [[ $1 == "eq" ]]; then  echo "eq"
  elif [[ $1 == "lt" ]]; then  echo "lt"
  elif [[ $1 == "le" ]]; then  echo "le"
  else error "Unknown operation: '$1'"
  fi
}

function xtiny() {
  if [[ $1 == "before" ]]; then echo "tininessbefore"
  elif [[ $1 == "after" ]]; then echo "tininessafter"
  else error "Unknown tininess: '$1'"
  fi
}

function gen_tf() {
  ty=$1
  opr=$2
  rmode=$3

#   echo "TFGEN f${ty}_$(xop "$opr")" "-r$(xrm "$rmode")"

  $TFGEN "f${ty}_$(xop "$opr")" "-r$(xrm "$rmode")" "-$(xtiny $TININESS)" $NUM_TESTS \
  | gawk '{ for (i=1;i<NF;++i) $(i) = " 0x" $(i); NF=NF; print }' \
  | sed "s|^|b${ty}${opr} ${rmode} |"
}


for ty in ${FORMATS[@]}; do
  for opr in ${OPS[@]}; do
    for rmode in ${RMODES[@]}; do #for rmode in '0' '=0' '<' '>'; do
      echo "Generating $ty $opr $rmode" >&2
      gen_tf "$ty" "$opr" "$rmode"
    done
  done
done
```

Appendix D

Python script to combine underflow detection methods

```python
import sys

test_file = sys.argv[1]
def to_bin(hex_s):
  return "{0:08b}".format(int(hex_s, 16))


def add_underflow_before(hex_s):
  bin_s = to_bin(hex_s)
  tmp = list(bin_s)
  tmp[2] = "1"
  bin_result = "".join(tmp)
  return "{0:02x}".format(int(bin_result, 2))

def get_flags(test):
  parts = test.split("␣")
  return parts[-1]

def has_underflow(flags):
  bin_s = to_bin(flags)
  return bin_s[6] == "1"


with open(f"tests_{test_file}_before.txt", "r") as f:
  lines = list(map(str.strip, f.readlines()))
  uflow_index = []
  for i, line in enumerate(lines):
    flags = get_flags(line)
    if has_underflow(flags):
      uflow_index.append(i)

with open(f"tests_{test_file}_after.txt", "r") as f:
  with open(f"tests_{test_file}.txt", "w") as wf:
    lines = f.readlines()
    for i, line in enumerate(lines):
      data = line.split("␣")
      flags = data[-1]
      if i in uflow_index:
        data[-1] = add_underflow_before(flags) + "\n"
      wf.write("␣".join(data))
```